Mechanizing Assembler Hazard Checks With Machine Descriptions

David A. Holland

October 28, 2017

Abstract

Many processors exhibit hazards*: combinations of instructions that should not be executed for fear of unwanted behavior. I present a framework for describing these in a crisp, precise way, an analysis that matches these descriptions against machine code, and an implementation of the analysis that generates hazard-checking code for a portable assembler.* **but it's not all done yet**

1 Introduction

At the machine level many processors exhibit *hazards* of one kind or another: sequences of instructions that lead or can lead to unspecified, unpredictable, or undefined behavior. Ensuring that such sequences are not executed is partly the responsibility of the compiler, which should avoid generating them. However, many hazards are associated with system control operations appearing mainly in hand-coded assembly language; it is thus desirable for assemblers to detect and reject (or perhaps mitigate) hazard code. Also, compilers occasionally exhibit code generation bugs, and if one of these results in accidental emission of a hazard sequence it is helpful to have a second line of protection.

Traditionally assemblers are partially or entirely hand-written, and so is any hazard checking code they might include. This has two drawbacks: first, a new machine architecture requires writing a new assembler, including any hazard checking code. This porting effort makes architectural experimentation expensive. It also likely means that during the initial development phase, a new architecture's assembler will have no hazard checking yet. This is, however, the time when machine-dependent kernel code is first being written and protections and cross-checking are most valuable.

Second, as new models of an existing architecture appear, they will usually have different hazard characteristics. (As we will see below, hazards mostly arise where internal implementation details are exposed; those details vary from model to model. Hazards also arise from bugs, which one hopes will *not* simply remain the same from model to model.) Consequently with every new model the hazard checking code grows more complicated; this has all the ingredients of a software maintenance nightmare.

To address this problem I introduce a portable table-driven hazard checker. The hazards themselves are declared in tabular form that can be indexed as needed by model. This makes the hazard declarations simple and orthogonal and thus easy to audit. The actual enforcement code is automatically generated. This is made possible by piggybacking on an existing assembler based on declarative machine descriptions: the hazard declarations for a particular machine architecture augment the description for that architecture, and can be specified in terms of machine properties (instructions, registers, etc.) by referencing the declarations of these things in that description.

In the remainder of this paper **blah blah blah**

2 Background

Some background notes are in order, first on hazards and their properties, then on machine descriptions, and finally on the `bintools` software suite and its specific machine description framework.

2.1 Hazards

In general, a hazard is a state of a processor in which something bad can happen, where "something bad" means an execution state whose semantics are not guaranteed by the processor specification. For example, on early MIPS processors, reading from a control register requires a one-cycle wait before the value becomes available, and trying to use the destination register without waiting gives unpredictable behavior [**?**]. Similarly, on 32-bit SPARC, the three instructions following an explicit load of the FPU control word must not test the FPU condition codes, at risk of seeing an undefined value.

These specific hazards (and many others) are caused by exposed details of the execution pipeline: while the hazard-generating instruction percolates through, more instructions are fetched and executed, and these must not do anything that interacts badly. Hardware designers can, but do not always, provide pipeline interlocks so that such states stall instead of exposing hazards. Sometimes this is because the extra silicon involved would add complexity or delays; but also (especially for system-level operations) the processor can often be made to do useful work while the hazard clears.

A second class of hazard involves an exception happening while a particular instruction is in the pipeline. Ideally an exception would stop the pipeline at a specific point, resulting in a clear self-consistent point to resume execution at after the exception is processed. However, this is not necessarily the case. On the same MIPS models cited above, the multiply/divide unit is not stopped on exception. Thus, if you start a new multiply right after fetching a previous multiply result, an exception at the wrong moment might abort the fetch but not the new multiply. Then the new multiply might complete during exception processing and overwrite the previous result, and on exception return the restarted result fetch will get the wrong value out. To avoid this, one must wait two cycles after fetching a multiply (or divide) result before issuing another multiply or divide instruction [**?**].

In addition to these I also include a category of similar safety considerations where the execution state is not undefined/unpredictable but incorrect: processor bugs. For example, on x86 the `SWAPGS` instruction exchanges the `%gs` segment register with an internal supervisor-only register. (This is intended for the kernel to use to find the in-kernel stack on system call entry via the `SYSCALL` instruction.) However, some AMD Opterons are missing the machinery needed to ensure seeing the latest value of `%gs`. Therefore according to AMD's errata sheet [**?**] an instruction that specifically synchronizes the pipeline must come between a load (via `MOV` or `POP`) of `%gs` and a `SWAPGS`.

Note that instructions that cause exceptions or runtime errors (such as dividing by zero) are not themselves hazards, as the behavior of a processor asked to divide by zero is (in most cases) clearly specified.

There is also a set of hazards that I cannot protect against (absent an oracle for the halting problem) because they are clearly undecidable. For example, on some processors there may be a delay between a TLB or page table update and the new translation taking effect. It is in general undecidable whether the instructions executed during the delay might be affected by the new translation. (Even if they do not themselves touch memory, they still need to be fetched.)

2.2 Machine Descriptions

Compilers and other programs that manipulate machine code need to know detailed properties of the architecture they work with. One strategy for dealing with this material is to collect

it up into a *machine description*; then porting the compiler to another target architecture amounts to producing a description for it. This is a standard concept in compiler backends, and in recent years (under the name "architecture description language" or ADL) has also become a standard concept for machine simulators.

The term, however, has no precise definition. Traditionally a machine description is "whatever my tool happens to need to be given to handle a new target". While in principle machine descriptions are in some way more concise or more self-contained than just writing code, in practice, particularly for older tools, machine descriptions are often code. They also often expose their tool's innards. The machine descriptions used by `gcc` are a particularly extreme example.

Sometimes the term "declarative machine description" is used to mean descriptions that are specifically not (Turing-complete) code, to help distinguish them from the noise.

I wish to approach this in a principled fashion, to maximize the usefulness of the machine description framework for porting. So for `bintools` I adopt the following principles. These are based on Ramsey's recommendations for machine descriptions [**?**], some of which have been (apparently independently) reiterated by Penry [**?**] in the context of machine simulators.

• *Machine descriptions should be declarative.* They should certainly not be Turing-complete; ideally they should not have code-like structure either. This makes them amenable to analysis, and also (by ruling out special-case hacks within individual descriptions) helps ensure the framework is general and flexible.

• *Machine descriptions should be self-contained.* Machine descriptions should be written in absolute terms, not relative to the internals of their associated software. For example, a description of instruction semantics for a compiler can be expressed as transfer functions on processor state; this is entirely self-contained. However, a traditional scheme is to provide, for each construct in a compiler's target-independent internal representation, a sequence of machine-specific instructions that implement that construct. This specifies the semantics of the machine instructions relative to the compiler internals. Exposing program internals to description writers has many fairly obvious disadvantages, of which probably the most important are (a) the description writer has to understand those internals, and (b) there is no hope of reusing/adapting the description for another compiler, or reusing another compiler's descriptions for your compiler.

• *Machine descriptions should be bidirectional.* Machine descriptions as a concept may be applied to many problems; however, in most cases these problems have both a forward and backward direction. For example, descriptions of instruction encoding can be used in one direction to encode instructions in an assembler, and in the other to decode instructions in a simulator. Similarly, descriptions of instruction semantics can be used to issue instructions in a compiler backend, or to execute them in a simulator. In general it is highly desirable to be able to write one description that can be used to drive both directions. This both reduces overall effort and avoids an entire class of bugs caused by inconsistent forward and backward specifications. Usually one direction is in some sense easier than the other; the needs of a bidirectional specification are then driven by the harder direction. For example, describing assembly-level instruction syntax for output is trivial (all it requires is some `printf` formats) so a bidirectional syntax specification is mostly about input parsing.

Note that Turing-complete descriptions are unlikely to be bidirectional.

• *Machine descriptions should be modular.* It is tempting to put all the information about

the machine in one file; e.g. all instructions one after another, with all properties of each instruction attached to that instruction's declaration. This is not the best plan. Ramsey envisions a world where machine descriptions are broadly reusable, in the sense that a wide range of programs from different authors all use the same universal machine description format. In that world (which unfortunately shows no signs of coming to pass) it is obviously desirable to be able to read in only the portions of a description that one cares about. For example, nothing in a conventional assembler cares what instructions do, so an assembler does not need a description of instruction semantics; but compilers obviously care deeply about this. One would be able to get the assembler running without having yet written the semantics description. However, even in the present world where every package or toolset needs to have its own set of machine descriptions, modularity is still useful: it is easy for a computer to combine information about one instruction from multiple inputs (it is just a relational join) but it is annoying and error-prone for a human to go through a single huge input file looking for specific subsets of information. Formats that contain one or two lines of cogent information per instruction expose patterns and structure (and thus, latent mistakes) that disappears when too much material is collected at once.

• *Machine descriptions should be concise but readable.* The description writer should not need to be an expert in the internals of one's program, as noted above; equally, the description writer should not need to be an expert in esoteric description languages. The intended meaning of description language constructs should be as obvious as possible (e.g. by analogy to well-known programming languages) without being laborious to work with. Overemphasis on concision breeds incomprehensibility; but also too much work on making the things one is talking about simple can tend to make the form one writes too long to be easy to work with. In programming languages making this tradeoff historically requires taste; APL and COBOL are both (in retrospect) traditional examples of having missed the target.

• *Machine descriptions should be checkable.* Since machine descriptions are used to generate code, bugs in descriptions lead to bugs in that code; debugging generated code by hand is unpleasant. To the greatest extent possible one would like to validate the descriptions—certainly for self-consistency, preferably directly against the machine being described—before trying to run the code produced from them. The use of types and type safety, for example, offers clear advantages. Furthermore, as already noted, descriptions that are modular are easier to audit; so are legible descriptions. And all analysis is easier when descriptions are not Turing-complete.

2.3 `bintools`

The `bintools` suite is a retargetable assembler, linker, and associated tools (`ar`, `ranlib`, `objdump`, etc.). It is comparable to GNU `binutils` in intent, and is ultimately intended for production use. Thus the operational parts must be written in C due to bootstrapping considerations. Unlike `binutils` and other similar suites it is retargeted entirely via declarative machine descriptions. Porting it to a new target does not require writing Turing-complete code.

From a machine/portability standpoint the interesting parts of `bintools` are handling the bit encoding of instructions, handling the syntax of instructions (and other aspects of the assembler's input syntax), and both generating and applying linker relocations. These issues affect the assembler and disassembler (broadly), and also a small part of the linker. The other tools in the suite are generally portable and need to know only basic things like the target's machine word size. (Like `binutils`, the suite does not include a debugger or machine emulator, which would

constitute separate projects.)

Note that in a production environment one must interoperate with existing compilers and tools. One does not, for example, get to choose the numbering or semantics of linker relocation codes or properties of the input syntax.

The bit encoding descriptions in `bintools` are largely based on Ramsey's SLED descriptions [**?**] with some adjustments and simplifications and some additional material needed to support a production environment. (These differences are discussed in Section 8.3.1.) The assembly syntax and linker relocation descriptions, which in keeping with the design principles in the previous section are separated from the encoding descriptions, are home-grown. There is no description of instruction semantics, as neither an assembler nor a disassembler needs to know what instructions do. (Linkers do not even need to know what instructions are.)

The machine description part of `bintools` was largely a reimplementation effort and was initially done in a rush on a tight deadline. The worst consequences of this have been cleaned up; however, its models and abstractions are not everything they could be and the implementation lags somewhat behind the design. Furthermore, there is no generated disassembler yet.

3 Hazard Model

The abstract model of a hazard I am using is as follows: executing some instruction (the *trigger*) puts the processor into a state that requires special handling (a *hazard state*). While the processor is in this state, certain instructions or operands to instructions must be avoided. This state may last for some period of time, some number of instructions, or until some other instruction is executed, at which point the hazard is *discharged* and no longer relevant. To describe hazard states I use regular expressions, specifically regular expressions over the alphabet of instructions. The regular expression describes the acceptable instruction stream when a hazard is triggered, including the trigger instruction.

Formally a hazard is a pair (t, r) where t is the trigger (an instruction predicate, possibly augmented with binders) and r is a regular expression describing the possible sequences of instructions required to discharge the hazard.

Instruction predicates are ordinary expressions in the description language. They may refer to fields of machine-level instructions by name. (These are treated as pre-bound variables, like elsewhere in the machine descriptions.) For modeling purposes these can be construed as follows:

$$\begin{array}{rcl} t & ::= & x := e; t \mid e \\ e & ::= & k \mid x \mid e_1 = e_2 \mid e_1 \circ e_2 \\ & \mid & \neg e \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \end{array}$$

where $\circ$ are additional uninterpreted operators, k are constants, and x are variables.

The regular expression grammar is:

$$r ::= \epsilon \mid . \mid [e] \mid r_1 r_2 \mid r_1 \nmid r_2 \mid r^*$$

which is conventional except for the use of an instruction predicate where text-matching regular expressions have character classes.

The binders in triggers appear as assignments (rather than let-bindings) because their scope extends past the trigger and includes the regular expression. This allows, for example, predicates within the regular expression to match against a register used in the trigger instruction.

Subsequent text will use h to refer to hazards, and $h.t$ and $h.r$ to refer to h's trigger predicate and acceptable instruction stream regular expression, respectively.

The goal of modeling hazards is to avoid unsafe executions. An execution is unsafe if the instruction sequence executed following a trigger does not match the required pattern. Thus:

Definition 1. An execution is *safe* if, for all hazards h, upon executing an instruction that satisfies $h.t$, some prefix of the continuing instruction sequence (including the satisfying instruction) matches $h.r$.

Definition 2. A program or program fragment is *safe* if all executions of it are safe.

In the next section I shall describe a static analysis that can be used to check this property.

4 Hazard Analysis

The hazards described in the previous section are *independent*, that is, the presence or absence of one does not affect the presence or absence of others. Therefore, it is sufficient to develop an analysis to check for safety with respect to a single hazard h; the implementation can run the same analysis repeatedly (or in parallel) for all declared hazards.

I will first describe a simple form of the analysis on a single standalone basic block, and then generalize it to arbitrary control-flow graphs. This stepwise elaboration clarifies the relationship of the analysis to the original hazard model. An alternate form that overcomes some of the drawbacks (but has drawbacks of its own) follows these.

4.1 Single Block Analysis

Assume that the single basic block has no jump (or branch, or call/return) instructions, and ends with some form of halt instruction. (The halt instruction might or might not be mentioned in $h.r$.) Let p and q be *program points*; program points exist at the beginning of the block and after each successive instruction.

Definition 3. Say that p is *r-safe* for some regular expression r if some prefix of the sequence of instructions beginning after p matches r. Write this formally as $\text{SAFE}(p, r)$.

Definition 4. Say that p is *h-safe* for our hazard h if either (a) the instruction x following p does not satisfy $h.t$, or (b) p is $h.r$-safe. Write this formally as $\text{SAFE}(p, h)$.

Definition 5. Say that our basic block is *h-safe* if for all p in the block, p is h-safe.

Since execution of a single block starts at the beginning and proceeds to the end, it follows readily that this definition of safety is equivalent to the execution-oriented definition given in the previous section.

Notice that every p is always ϵ-safe: the empty regular expression matches the empty prefix of any sequence of instructions whatsoever.

Now, observe that Definition 3 can be incrementalized.

Proposition 1. If x is the instruction between p and q, and r' is the *derivative* [**?**] of r with respect to x, then $\text{SAFE}(p, r) \Leftrightarrow \text{SAFE}(q, r')$.

This follows directly from the definition of a regular expression derivative: r matches the sequence $xy...z$ if and only if r' matches the sequence $y...z$.

We will use this form for the full-graph analysis, because it allows reasoning about matches that span blocks.

4.2 Full Graph Analysis

Generalize the environment to an arbitrary control flow graph: arbitrarily many basic blocks b linked together in an arbitrary structure, where each block has either one or two successor blocks and arbitrarily many predecessor blocks. Each block consists of some number of non-jump instructions and potentially has a jump (or call, or return, or branch) at or near the end that chains to the successor blocks. Note that in some cases (e.g. indirect jumps, returns) the identity of a successor block might be unknowable or require more analysis to identify than is feasible in an assembler.

As before, let p and q be program points; program points exist at the beginning of each block and after each instruction, except that the end of a block with two successors has two program points at the end, one for each successor.

Generalize r-safety to this environment:

Definition 6. Say that p is *r-safe* for some regular expression r if either (a) r is ϵ or (b) x is the instruction between p and a following program point q, and r' is the derivative of r with respect

to x, and for all such q and r', q is r'-safe. Write this formally as $\text{SAFE}(p, r)$.

Proposition 2. If q is a program point at the end of block b_1, and it leads to a successor block b_2, and p is the program point at the beginning of b_2, then $\forall r, \text{SAFE}(q, r) \Leftrightarrow \text{SAFE}(p, r)$.

(This is just a statement of what it means for blocks to follow one another.)

Proposition 3. For any program point p, it is always correct (a conservative approximation) to assume it is not r-safe for any r other than ϵ. (That is, $\forall r, r \neq \epsilon \Rightarrow \neg\text{SAFE}(q, r)$.)

If p is actually r-safe for other r, disallowing reaching it in a situation where r is expected may result in rejecting code that can execute safely, but will not result in accepting code than can execute unsafely. (Recall that all program points are ϵ-safe, which is now by definition but was motivated in the previous section.)

For program points at the ends of blocks where the successor block is not known (or is unavailable for analysis due to separate compilation) this approximation will serve to maintain safety.

Now restate h-safety, which is not materially different in this environment:

Definition 7. Say that p is *h-safe* for our hazard h if either (a) the instruction x following p does not satisfy $h.t$, or (b) p is $h.r$-safe. Write this formally as $\text{SAFE}(p, h)$.

Definition 8. Say that our program is *h-safe* for our hazard h if for all points p in all blocks in the program, p is $h.r$-safe.

This gives us a definition for whether the full program is safe under h, which we can reasonably believe (by reasoning relative to the single-block form) respects the original hazard model. However, we have a slight problem: the r-safety of some program points may be defined in terms of itself. (For example, imagine a hazard that must be discharged with an explicit barrier instruction, and a short loop in the code between the point the hazard is triggered and the barrier. Or, more annoyingly, such a hazard followed by an infinite loop of nops.)

Informally, one can argue that when this happens, the results are still safe: the premise of the safety property is that nothing bad has happened, so if execution cycles back to the same state without anything bad happening, the results are safe. This is unsatisfying; however, the self-referential equivalence loops can be removed formally via coinduction. Unfortunately, the detailed treatment of the coinduction is not ready yet. **blah. how does one say this? one never would in a real paper**

4.3 Alternate Form

Here is an alternate form of the analysis that integrates a coinductive proof technique to eliminate the self-referential equivalences directly.

Extend the regular expression notation with an $\&$ ("and") operator, where $r_1 \,\&\, r_2$ means that both r_1 and r_2 must match the input. For notational clarity call these regular expressions $\hat{r}$ instead of r. Note that $\epsilon \,\&\, \hat{r}$ is equivalent to $\hat{r}$, $\hat{r} \,\&\, \hat{r}$ is equivalent to $\hat{r}$, $\&$ is associative and commutative, and the derivative of $\hat{r}_1 \,\&\, \hat{r}_2$ with respect to x is $\hat{r}'_1 \,\&\, \hat{r}'_2$ where $\hat{r}'_i$ is the derivative of $\hat{r}_i$. If the derivative of either subexpression does not exist, neither does the derivative of the conjunction.

Now use Hoare triple notation to indicate the requirements for safe execution: let $\{\hat{r}\}\ x\ \{\hat{r}'\}$ mean that the instruction x, executed in a context where the (extended) regular expression $\hat{r}$ describes the required instruction stream, executes safely such that $\hat{r}'$ describes the required subsequent instruction stream.

Based on the definitions in the previous section, there are four cases for a single instruction:

$$\{\epsilon\}\ x\ \{\epsilon\}$$
$$\{\epsilon\}\ x\ \{h.r\}$$
$$\{\hat{r}\}\ x\ \{\hat{r}'\}$$
$$\{\hat{r}\}\ x\ \{\hat{r}' \,\&\, h.r\}$$

where $h.r$ appears if x satisfies $h.t$ (that is, the hazard has been triggered) and $\hat{r}'$ appears if $\hat{r}'$ is the derivative of $\hat{r}$ with respect to x. The first two cases are restricted to ϵ on the left because if there is no derivative of $\hat{r}$ with respect to x, either $\hat{r}$ is ϵ (in which case any x is safe) or x does not match $\hat{r}$, in which case x is not safe.

The standard axiom for sequential composition in Hoare logic now allows reasoning about sequences of instructions, up to whole basic blocks. **Margo: shall I state it?** However, caution is needed regarding blocks with two exit points.

For an ordinary branch instruction, there are two results, corresponding to the two output program points, but they must be the same, because the branch instruction itself can only match the regular expression in one way. However, some architectures (including MIPS and SPARC) have so-called "branch-likely" instructions, where the instruction in the delay slot is conditionally executed based on whether the branch is taken. Thus the two exit points of a branching basic block may have different Hoare triples, and it might be important to distinguish them when reasoning about the full control-flow graph. Write these as $\{\hat{r}_1\}\ b.1\ \{\hat{r}'_1\}$ and $\{\hat{r}_2\}\ b.2\ \{\hat{r}'_2\}$ for the first and second output program points respectively.

Now the sequential composition axiom allows combining blocks. Do this as follows:

1. Make two lists of pairs $(b, \hat{r})$, one to remember cases that have already been seen and a worklist to hold cases that have not yet been examined. b is a block and $\hat{r}$ is a context for the block: an (extended) regular expression it must match to execute safely.
2. The list of cases seen starts empty.
3. Initialize the worklist with (b, ϵ) for every block b, or restrict this to some subset of blocks known to be allowable initial entry points.
4. Take an entry $(b, \hat{r})$ off the worklist.
5. If b has one exit point, find the corresponding triple $\{\hat{r}\}\ b\ \{\hat{r}'\}$. If it does not exist, reject the program as it is unsafe.
6. If b has two exit points, find both corresponding triples $\{\hat{r}_1\}\ b.1\ \{\hat{r}'_1\}$ and $\{\hat{r}_2\}\ b.2\ \{\hat{r}'_2\}$. If both do not exist, reject the program as it is unsafe.
7. For each triple found (call it $\{\hat{r}\}\ b\ \{\hat{r}'\}$):
8. Simplify $\hat{r}'$ with respect to $\&$.
9. If the pair $(b', \hat{r}')$ where b' is the corresponding successor block has been seen before, ignore it.
10. Otherwise, remember it (add it to the list of cases seen) and also add it to the worklist.
11. If the corresponding successor block is not known (or not available for analysis) and $\hat{r}'$ is not ϵ, reject the program.
12. Repeat from step 4 until the worklist is empty or the program is rejected.

Proposition 4. If this algorithm terminates, the program is safe.

The Hoare triples have been constructed such that they can only be stated if the block and context they describe is safe. Therefore, if the algorithm sees any context in which a particular block is *not* safe, it will be unable to state a triple and will reject the program. It is therefore sufficient to show that the algorithm sees every block and context that is reachable in the course of execution. Observe first that every block and context added to the worklist is examined. Then proceed by induction: for the base case, the worklist is seeded with every possible initial block and the correct initial context (ϵ); and for the inductive case, every (reachable) block and context found on the worklist is processed and its reachable successor blocks and contexts are added to the worklist exactly once. Thus every reachable block and context appears on the worklist and has been examined for safety.

Proposition 5. This algorithm always terminates.

Since the number of blocks in the program is (presumptively) finite, and the number of possible regular expressions encountered is also finite, in the worst case every block and every possible context will be examined but then no more cases can be added to the worklist. The number of possible regular expressions encountered is finite because they are all derivatives of $h.r$, or combinations of these with $\&$. There are only a finite number of distinct derivatives: all the regular expression constructs besides r^* are structurally decreasing, and while r^* has infinitely many derivatives, only a finite number of distinct ones are generated before returning to r^*. There are also only a finite number of combinations that can be constructed with $\&$ given the basic simplification rules stated when it was introduced above.

The coinduction embedded in this scheme happens when cases that have been seen before are discarded: those are the ones where in the prior formulation the chains of equivalence become cyclic. The prior formulation is reasoning about the total execution after a point, so the equivalences need to eventually reach some ground state, or match a coinduction hypothesis about infinite equivalence chains. This formulation is reasoning only about the execution from one point to another, which allows chains of identical executions to be folded together. Unfortunately it has a drawback of its own, namely the introduction of a custom regular expression operator.

5 Implementation-level Analysis

There are a some of things about the previous analysis that should be refined in an actual implementation.

First, while derivatives of regular expressions are a nice formulation that allows reducing the number of moving parts in the analysis, ultimately the regular expressions should be compiled into state machines and the implementation of the analysis should be done in terms of states. Fortunately the analysis is not materially different when written in terms of states: each possible regular expression corresponds to a state, a derivative is a state transition, etc. The unorthodox $\&$ operator can be compiled into the state machine, and for that matter the triggering of the hazard can as well.

Second, because the assembler does not have a semantic model of the instruction set, it does not know which instructions cause control flow; it has to be told. On architectures with branch delay slots, and particularly architectures where the branch delay slots are conditionally executed, this gets complicated. However, it turns out that the same logic used to model hazards can also be used to model delayed jumps and branches: the jump instruction itself is the trigger, and the jump is taken when the "hazard" discharges. For branches one can simply write two regular expressions, one for each case. "Branch-likely" instructions can be handled with a flag attached to one branch case that causes the instructions to be ignored.

Third, as noted in Section 1, hazards tend to be specific to particular processor models within an architecture. For this reason every hazard declaration has a *guard expression* attached to it that determines whether it is checked. This expression will typically refer to the architecture revision or specific processor model in use, and perhaps also to user-settable assembler modes or command-line options. (These are not mentioned in the previous two sections because there they are an extraneous complication.)

Last and not least, the analysis described is context-sensitive and requires reanalyzing each basic block for every new context. (It also requires a basic block analysis.) For the implementation, I want an analysis that runs in one pass top to bottom on the assembly input, without requiring significant changes to the internal architecture of the assembler. Since the analysis runs

at the machine-instruction level and the assembler does not materialize a representation of the input at this level (but rather processes instructions one at a time and dumps them out into a binary blob) this is a fairly strict requirement.

First, for assembler input it is fairly easy to do without a basic block analysis: the input is a sequence of instructions and labels, and seeing a label or a control flow transfer instruction starts a new block; there is no need to materialize the blocks provided that they can be processed once each in whatever arbitrary order the arrive in.

To avoid having to rescan blocks, I speculatively check every possible context of every new block. Subsequent branches backward can thus be resolved. Note that the possible contexts of a new block are limited to cases where a jump or branch instruction can match in the proper prior position; this will, in general not be all of them. The speculative aspect of this is not particularly complicated; the complications arise when fields from the trigger instruction are matched by the regular expression. When speculating these must be treated as symbolic values, meaning that the safety of a block in a speculative context becomes a boolean expression in terms of these values rather than a constant. These expressions can become complex as the regular expression matching logic can flow into them. Fortunately, however, I need not check them for satisfiability, only evaluate them when given values from a concrete context. Thus the implementation is reasonably straighforward, though not entirely trivial.

Note that at the end of the input, some labels may still not have been seen and their safety is not yet resolved. These labels are (by definition) external, so all nondischarged contexts referencing them are presumptively unsafe. This may result in rejecting the input.

Observe that while this may seem like a lot of speculation, given that a result must be computed and stored for every label, for every hazard, and for (potentially) every state associated with that hazard. However, both the storage requirement and the time overhead are linear in these numbers. (Remember that while each of these results is potentially a boolean expression, the possible size of these expressions is bounded.) Since the expected number of hazards is on the order of tens, and the expected number of states per hazard on the order of ones, and the expected number of labels is on the order of hundreds (or maybe thousands for a large file), this should not constitute an unmanageable amount of overhead.

6 Implementation

The actual implementation in `bintools` is a mixture of C and OCaml code: the assembler itself is written in C and the machine description system is written in OCaml. One additional file is added to the description of each machine architecture: a file that contains specifications of instruction predicates, specifications of hazards, and specifications of jump instructions. (In order to maximize the legibility of the regular expressions in the hazard specification, instruction predicates must be written separately and named.)

The instruction encoding code generated by the description system is augmented to test the instruction predicates and call into the hazard checking code to drive the state machine. It is also augmented to remember jump and branch targets: it turns out that because the analysis runs at the machine-instruction level (with access to fields of instructions, rather than source-level operands) by the time the analysis sees a branch or jump its target has been converted from a symbol to a numeric offset or (worse) an anonymous relocation entry in a relocation table. This problem is handled in a pragmatic way by tagging the target operand of branch instructions in the encoding description and sending it off to the hazard checker at a higher level before the operand is evaluated.

The state machine table is generated and dumped out by the description system; the code

that drives it (which is machine-independent) is handwritten C. The implementation is approximately **1,000,000** lines of C and **1,000,000,000** lines of OCaml. **write more here when there's more implementation**

In order to print useful error messages when rejecting the input, every "live" context remembers the line number where its hazard was triggered and every failed speculative context remembers the line number where an inadmissible instruction was found.

7 Evaluation

There are three aspects to the evaluation of this scheme: first, evaluate the completeness and expressivity of the hazard model; second, evaluate the soundness of the analysis with respect to the hazard model; and third, evaluate the implementation based on its overhead in performance and possibly memory usage.

(Another possible aspect is to compare the hazard description language to the usually confusing and ambiguous English text descriptions of hazards found in processor and architecture manuals; however, this would be hopelessly subjective.)

7.1 Model Completeness

The scope of the model is somewhat modest: it is intended to model a specific type of hazard, namely pipeline hazards with either a finite lifespan or an explicit discharge or barrier instruction. And, as noted up front, it only attempts to model these when conflicting instructions are readily identified.

This is, however, a fairly substantial category; it includes, for example, all the usermode-exposed pipeline hazards of the original MIPS r2000 and r3000, including rules like not placing branches in branch delay slots, and most of the supervisor-mode hazards [**?**], the chief exceptions being undecidable issues like delayed TLB changes, or delayed changes to interrupt enable bits. (If disabling interrupts is delayed for two cycles, how can the assembler tell if the instructions following the change are safe to interrupt?)

I have not had time to conduct a detailed survey of a large number of architectures **is this ok to say?** to try to quantify this. (However, given that hazards tend to be obscure—they are rarely specified clearly, or defined authoritatively in any one manual, and sometimes vary without explanation between manual revisions—it is not clear how useful or accurate such a survey would be.)

As far as processor bugs: of 77 errata described in the AMD Opteron errata sheet referenced above [**?**], the model can capture five and perhaps a sixth. However, many of these are clearly beyond not just the intended scope of the model but the scope of the assembler: some are hardware-level issues, some are internal problems that are mitigated by setting some otherwise-undocumented bit in a model-specific register, some are things mitigated by the operating system using or not using memory in a certain way, etc. Of the errata that might conceivably be handled by code analysis (possibly including a halting oracle, many requiring nontrivial semantic analysis), the model captures 5-6 out of 10-12. These numbers are approximate because several of the descriptions are not particularly clear.

There are some other specific limitations:

- The model cannot express hazards of the form "instruction `xyz` is safe only after instruction `pdq`". One can approximate this with a hazard that triggers on any instruction that is not `pdq` and must be followed by not-`xyz`; but this fails at program entry, on external calls, on computed jumps, and anywhere else that relies on the assumption that all-hazards-discharged is the baseline safe state. (An example: the MIPS r3000 return-from-exception instruction is not a jump but a state-update instruction meant

to sit in the delay slot of a jump. Any use *not* in a jump delay slot is presumptively incorrect.)

- There is no ability to reason directly about time; instruction counts must be used as a proxy for time. This is an adequate approximation for old RISC processors with lots of exposed pipeline behavior. In more modern environments (more or less since the advent of superscalar hardware) instruction timings are not predictable enough that reasoning about either timing or instruction counts is effective, and explicit hazard-clearing instructions are typically required, so this restriction becomes immaterial.

- I intended to also allow reasoning about instruction alignment in memory, because this is often relevant to processor errata; however, it does not fit this model at all and should probably be implemented as a separate facility.

- The implementation cannot reason about instruction fields that are targets of relocations. Arguably this is a bug; however, in general the values of such fields are not knowable prior to link time, and checking them would require a linker-level facility and the ability to leave some form of check records in the output object file for the linker to process. Also, the values of such fields are normally addresses or parts of addresses, and not (for example) identifiers for functional elements of the CPU. They are unlikely to participate in pipeline hazards, and where they do the fact that arbitrary addresses can appear in registers as the result of arbitrary computation will usually make reasoning about such hazards undecidable anyhow.

7.2 Soundness

My goal has been to construct a mechanized soundness proof (of both the abstract and implementation-level versions of the analysis) in Coq. (That is, prove that a Coq model of the analysis respects a Coq version of the hazard model.)

This is underway but not yet complete. **and unlikely to advance before the written article is due**

Sketch proofs, or at least outline-level reasoning, do appear above.

7.3 Implementation

None of this has been done yet since the implementation is not finished. **sigh**

Since assembling an ordinary-sized file is too fast to benchmark well, and assembling a made-up huge file is unrealistic, what I have had in mind to capture the overhead (unless it turns out to be detectable under ordinary circumstances, which seems like it would be a bug) is to run the assembler in an emulator for a slow vintage machine and measure timings there. Possible options include NetBSD on sun3 (m68k) or on pmax (MIPS r3000); perhaps also OS/161 on System/161, although that has various drawbacks.

The benchmark would be to take a variety of fairly large but not unreasonably huge source files (e.g. `vfs_syscalls.c` from NetBSD, which is roughly 5000 lines of C, or use the SPEC benchmark files), run them through gcc up front, and assemble the results with warm cache, using (a) the assembler without the hazard code at all, (b) the assembler with the hazard code present but not enabled, and (c) with hazard checking turned on, and measure the elapsed time.

8 Related Work

There are several categories of related work. The first is work about machine descriptions; the second is work about handling pipeline hazards and similar safety properties; and third is other retargetable assemblers and linkers. Meanwhile the work about machine descriptions can

be broken down further by the material described. (Though note that some description frameworks, like that of bintools, support description of multiple classes of material.)

8.1 Describing Microarchitecture

The hardware and architecture community has long used machine descriptions of one form or another to describe processor implementations. Some of these descriptions are effectively code (e.g. Verilog [**?**]) and others are more abstract; however, they all have in common the fact that microarchitectural implementation and microarchitecture details are irrelevant to bintools, and thus I will not consider them further.

8.2 Describing Instruction Semantics

I'm thinking I should leave this out, or make this section like the previous one. It's mostly here because I had to wade through a lot of this (because a lot of semantics description languages also have encoding descriptions attached) and it was a pain, and I want everyone to know it was a pain. Which (like putting something in a paper because it was hard and you want everyone to know it was hard) is kinda lame.

Describing instruction semantics is the primary issue for machine descriptions powering compiler backends, where (as mentioned above) machine descriptions (and languages for machine descriptions) are a standard concept. Blindell [**?**] writes "...many such languages and related tools have appeared–and then disappeared–over the years" (p. 14) and cites a survey paper from 1969. Blindell's book is a survey of instruction selection methods, not of machine descriptions, and mentions machine descriptions only in passing; but many of the 300-odd publications and systems cited have some machine description concept. Surveying these in detail is a fairly daunting proposition.

As description of instruction semantics is also irrelevant to bintools (though not to the larger project bintools is embedded in) **is it ok to say that?** I will mention only a few recent examples specifically interesting from the point of view of their descriptions.

Register transfer lists, or RTL, are a standard technique for encoding the semantics of machine instructions as transfer functions on concrete processor state. Unfortunately, writing down raw RTL for every processor instruction is akin to implementing complex logic in assembly language – you can specify everything exactly in precise detail but you *must* specify everything exactly in precise detail and there are many opportunities to make mistakes. Ramsey's λ-RTL [**?**] is a scheme for using applicative functions and type inference to build RTLs in a way concise enough to be human-writeable. Dias and Ramsey [**?**, **?**, **?**] use this (as well as other things, such as a description of calling conventions [**?**]) to build a compiler backend fully parameterized by non-Turing-complete descriptions. However, this approach requires that the frontal parts of the compiler emit RTL, or an intermediate representation equivalent to RTL.

Hohenaur [**?**] extends the simulator-oriented LISA description language [**?**] with non-Turing-complete semantics descriptions (the original semantics descriptions for generating simulators are arbitrary C++ code) and uses this material to generate a compiler backend. (However, generating simulators with the new material is not discussed.)

Richards [**?**] offers a description language CISL that describes both instruction encodings and semantics. Rather than specifically using register transfer lists, CISL can be used with any intermediate representation; both the intermediate language and the target machine are described with CISL and the backend generation tools match these up. The results demonstrated include backends for both the `lcc` C compiler and the Jikes JVM.

Machine descriptions have also become a standard technique for implementing simulators. However, in this context the "description" is usu-

ally just code. For example, Wagstaff [**?**] extends the ArchC description language [**?**] with semantic information, but that semantic information is arbitrary C++ code. Similarly, Lockhart's Pydgen [**?**] is a simulator framework that sports "simple architecture descriptions" but these are actually Python code that implements instructions. This code is then fed to PyPy's RPython JIT.

8.3 Describing Instruction Encoding

Since the encoding descriptions in bintools are largely based on Ramsey's SLED descriptions [**?**], I will first describe that relationship before moving on to other description languages.

8.3.1 SLED vs. `bintools`

The encoding descriptions in bintools are simpler and more explicit than SLED descriptions. However, most of the key features of SLED are retained, in particular the use of strong typing and type inference. Instruction fields are strongly typed: numeric fields have explicit size and signedness, and opcode and register fields are enumerations. Distinct register classes cannot be accidentally mixed, symbolic opcodes may only be placed in the field they belong to, and so forth. Type inference keeps the written overhead of all this to a minimum, and parametric polymorphism allows factoring out common material.

The chief simplification is that SLED has a very general (perhaps too general[1]) concept of "patterns", which have been removed entirely from the bintools descriptions. In SLED, many things, such as the association of particular instruction fields with specific instruction formats, are specified directly as patterns or deduced by analyzing patterns. In the bintools descriptions explicit constructions for these things are provided instead.

Instruction formats in bintools descriptions are given explicitly (as "forms", ordered sequences of fields similar to the "formats" in ArchC); this supplies both the association of fields with formats and the positioning of fields. In SLED the positioning of fields must be stated with explicit bit offsets, which are both fragile (that is, easy to mistype) and also subject to numbering order/endianness considerations.

Many of the uses of patterns in SLED descriptions are to enumerate subsets of opcode alternatives; instead there is a specific construction for naming subsets of enumerations and using them in predicates and guard expressions. Another use of patterns in SLED descriptions is the expansion of "macro instructions" (constructs that are one syntactic instruction in the input assembly language, but multiple machine-level instructions in the output); in bintools there is an explicit macro construction for these. One consequence of this is that in bintools the disassembler cannot fold these expansions back to their source form; however, this is not actually desirable behavior in the field, where by the time one is disassembling one usually wants to see the exact primitive instructions executed by the machine.

Another major difference is the interface between the encodings and the assembly-language-level instruction syntax. In SLED the "constructors" (which are mostly machine-level instructions, though they can also be partial instructions) have some limited syntactic information attached to them, but it is not really sufficient for use in a production assembler (which must interoperate with existing compilers and thus accept externally determined syntax) and furthermore there is no real support for cases where the same assembly-level instruction has many entirely unrelated machine-level encodings, such as the x86 `mov` instruction. (These can be *represented* in SLED by giving them each a different name, but not disambiguated or parsed.)

In bintools assembly-language-level instruc-

[1] The paper says: "Studying the details of the [internal] representation is the best way to understand the meanings of patterns and the pattern operators and to understand the utility of patterns in generating encoders and decoders."

tions (that have opcode strings and operand expressions) are a separate concept from machine-level instructions (that have opcode enumerations and field values) and there is an explicit mapping between the two. (Currently the mapping is entirely explicit and has to be written out in longhand for all combinations, which is clearly undesirable; I have a scheme in mind to use a solver to generate it, but this has neither been implemented nor specified in detail yet.)

The bintools assembly-language-level instruction model is that each instruction consists of an opcode string and zero or more operands, each of which has a type. The types of specific input operands are identified at assembler runtime by matching logic belonging to the syntax specification; then the type signature is used to index the mapping to machine-level instructions, each of which has a (possibly parameterized) encoding procedure generated from the encoding descriptions.

8.3.2 Other Encoding Descriptions

There are also a lot of encoding description languages.

XXX

nML? decgen? UPFAST LISA (pees-1999) EEL rosetta (krishna-2001) ISDL DERIVE facile simit-arm HOIST yirr-ma qin-2004 ArchC isildur reshadi-2006 derrico-2006 klein assembler system cotson lim-2009 CISL SGDL rocksalt sepp-2012 HARMLESS walters-2013? fournel-2013 tan-2016

I have at least some actual stuff to put here but I haven't managed to type it in yet

8.4 Describing Assembler Syntax

As mentioned above, SLED descriptions include some syntactic information, but it is not sufficient to generate assemblers that accept the output of existing compilers or assemble the existing assembly source files found in e.g. operating system kernels. (For example, there is nothing to support arithmetic expressions or to indicate where arithmetic expressions are potentially acceptable.)

Many of the simulator-derived description languages have a syntax description, but these are typically output-only and consist only of a collection of `printf` formats.

I have not found anything specifically about describing the syntax of assembly languages. (There is of course a large literature on parsing in general.)

The current scheme in bintools, which is meant to take advantage of the common basic layout of most assembly languages, is a core shared lexer and parser augmented with additional machine-specific tokens and sets of rewrite rules to be applied in specific contexts. I am not satisfied with it, because it feels like a collection of special cases; but on the other hand making the description writer provide a complete LR(1) grammar (much of which will be cut-and-paste for most machines) to feed to `yacc` or similar seems excessively general, and despite assorted attempts to the contrary LR parsing apparently remains hard for the uninitiated.

8.5 Describing Linker Relocations

Since bintools must interoperate with existing linkers and libraries, it must limit itself to existing definitions of linker relocations. This makes work that envisions a better world not very helpful. For example, Ramsey proposed a scheme that uses partially-applied functions and a simple interpreter [**?**] to make linker relocations machine-independent. This looks very attractive, but not very practical.

Abbaspour and Zhu [**?**] wrote a tool to generate ports of the BFD ("binary file descriptor") library, one of the core parts of GNU binutils. This was able to encode several of the basic relocations for i386 and SPARC. Unfortunately it is unclear how general it is; MIPS for example has some relocations that need to be matched in pairs before being applied, and has others whose

semantics differ when applied to local vs. external symbols, and representing these is problematic under the best of circumstances.

Kell, Mulligan, and Sewell [**?**] have modelled (static) linking of ELF files in Isabelle. This includes a specification of at least some relocation codes for each of five architectures (AArch64[2], Power64, both 32-bit and 64-bit x86, and since the paper was published they have added MIPS64) but not all, and most likely not the more complex relocations, which tend to arise in dynamic linking. It is not clear from the paper what their descriptions of these look like; however, visiting their source repository shows that they are code.

The only other related reference I found is that I apparently called this a solved problem in 2004 [**?**]. Oops.

8.6 Hazards

RockSalt [**?**], which is a verified checker for Google Native Client (NaCl) [**?**] code, is a similar tool: given machine code, it verifies that a set of desired safety properties are maintained. The safety properties have a different nature: the main challenge for NaCl verification is checking that indirect jump addresses have been masked properly (they must go only to 32-byte-aligned destinations) which requires substantial semantic analysis and is fundamentally a harder problem. Conversely however the analysis is local to basic blocks and does not involve properties that may carry across jump and branch instructions, as hazards can. **I should probably make one more go at improving this description**

In the compiler community there is a good deal of work on hazard modeling for the use of instruction schedulers. Much of it is about deducing hazard specifications such as the ones I propose from microarchitectural-level pipeline specifications, such as Proebsting and Fraser 1994 [**?**]. **and I need to look at these more**

[2] which is 64-bit ARM

8.7 Other Toolchains

There are quite a few toolchain packages out in the world. (Surprisingly many perhaps, given that in practice everyone uses GNU binutils.)

GNU binutils. Retargeting of binutils is entirely by hand (modulo the Abbaspour and Zhu work mentioned above [**?**], which did not get merged upstream) – some aspects of machine specification are table-driven rather than open-coded, but most of these rely extensively on ways to attach code for cases the basic models fail to cover.

LLVM tools. The LLVM project has been building its own set of assemblers and linkers. These are partly description-driven, but some aspects of each assembler are still hand-coded for each port.

Keystone and Capstone. These are an assembler and disassembler (respectively) based on the LLVM tools with various extensions. These extensions do not seem to include extensions to the retargeting mechanisms.

Elftoolchain. The elftoolchain project was started as a BSD-licensed replacement for binutils, but so far it does not yet have an assembler or linker at all.

Metasm. Metasm is a Ruby toolkit for dealing with machine code, including assembling and disassembling. However, it turns out that its CPU models are arbitrary Ruby code.

Cgen. Cgen is, unlike all the above, specifically architected to use machine descriptions, although their machine descriptions can attach arbitrary Scheme code. However, it is in a very preliminary state (all it actually supports is machine simulator generation for fairly simple embedded CPUs) and its development is not very active.

9 Future Work

Obviously the first article of future work is to finish the implementation and the soundness proofs, and run the desired benchmarks.

Beyond that, one interesting proposition is automatically mitigating hazards instead of rejecting unsafe input: since the assembler does not have a semantic model of instructions, it would need to be given a list of instructions (or instruction sequences) with no effect it can insert freely; but given those it is fairly straightforward to match them against the regular expressions defining hazards to find a shortest mitigation sequence for every state of every hazard. (Or at least where one exists; for hazards that must be discharged with hazard-clearing instructions that also have other effects, it isn't very feasible for the assembler to try to figure out how to insert those instructions safely.)

Another possibility is to give the assembler a semantic model of instructions and add further dataflow analysis, for example to trace the destination of return instructions and perhaps some indirect jumps and branches.

It might also be interesting to investigate stronger constraint systems, such as types [**?**] or outright verification [**?**].

Rewriting the implementation, or at least the description framework part of the implementation, in Coq instead of OCaml and proving the implementation (rather than just a model) sound is another possible direction, and definitely feasible though not trivial. (Proving the assembler itself correct is a bigger job given that it is written in C.)

10 Conclusions

I should have some

References