Fault-tolerant Typed Assembly Language

Frances Perry[†] Lester Mackey[†] George A. Reis[†]
Jay Ligatti[‡] David I. August[†] David Walker[†]

[†]Departments of Computer Science and Electrical Engineering
Princeton University
{frances, lmackey, gareis, august, dpw}@cs.princeton.edu

[‡]Department of Computer Science and Engineering
University of South Florida
ligatti@cse.usf.edu

Princeton University Technical Report TR-776-07

Abstract

A *transient hardware fault* occurs when an energetic particle strikes a transistor, causing it to change state. Although transient faults do not permanently damage the hardware, they may corrupt computations by altering stored values and signal transfers. In this paper, we propose a new scheme for provably safe and reliable computing in the presence of transient hardware faults. In our scheme, software computations are replicated to provide redundancy while special instructions compare the independently computed results to detect errors before writing critical data. In stark contrast to any previous efforts in this area, we have analyzed our fault tolerance scheme from a formal, theoretical perspective. To be specific, first, we provide an operational semantics for our assembly language, which includes a precise formal definition of our fault model. Second, we develop an assembly-level type system designed to detect reliability problems in compiled code. Third, we provide a formal specification for program fault tolerance under the given fault model and prove that all well-typed programs are indeed fault tolerant. In addition to the formal analysis, we evaluate our detection scheme and show that it only takes 34% longer to execute than the unreliable version.

1 Introduction

A *transient fault* or *soft error* is a temporary hardware failure that alters a signal transfer, a register value, or some other processor component. While transient faults are temporary, they corrupt computations and have led to costly failures in high-end systems in recent years. For example, in 2000 there were reports that transient faults caused crashes at a number of Sun's major customer sites, including America Online and eBay [2]. Later, Hewlett Packard admitted multiple problems in the Los Alamos Labs supercomputers due to transient faults [7]. Finally, Cypress Semiconductor has confirmed "The wake-up call came in the end of 2001 with a major customer reporting havoc at a large telephone company. Technically, it was found that a single soft fail. . . was causing an interleaved system farm to crash" [27].

Unfortunately, while soft errors can already cause substantial reliability problems, current trends in hardware design suggest that fault rates will increase in the future. More specifically, faster clock rates, increasing transistor density, decreasing voltages and smaller feature sizes all contribute to increasing fault rates [1, 11, 20]. Due to a combination of these factors, fault rates in modern processors have been increasing at a rate of approximately 8% per generation [3].

These trends are well known in the architecture and compiler communities, and, consequently, many solutions to the threat of soft errors have been proposed. At a high level, all of these solutions involve adding redundancy to computations in one way or another, but the specifics vary substantially. For instance, there are proposals involving hardware-only solutions such as error-correcting codes, watchdog co-processors [6] and redundant hardware threads [4, 9, 15, 24] as well as software-only techniques that use both single and multiple cores [12, 13, 16, 17, 19, 23]. Broadly speaking, if the technique can scale, hardware-only solutions are more efficient for a single, fixed reliability policy, but software-only solutions are more flexible (they may be deployed exactly when, where, and to the degree needed) and less costly in terms of hardware. In an attempt to gain some of the best of both worlds, researchers have also recently proposed hybrid software-hardware solutions involving strong fault tolerance mechanisms implemented in hardware but controlled by the software running on the processor [18].

Software-only and hybrid hardware-software techniques also possess at least one further, little-mentioned drawback — *they may not actually work*. To be fair, many of these techniques appear extremely promising. However, as far as we are aware, the published transient fault-tolerance techniques come with no rigorous proofs that they guarantee any particular reliability properties. In general, researchers satisfy themselves with presenting an algorithm for fault-tolerance and leave the audience to judge for themselves whether or not the algorithm is correct. In fact, the literature does not even precisely define what it might mean for an assembly-level program to be fault tolerant. This paper tackles this gaping hole in the existing literature by defining a new hybrid hardware-software technique for tolerating transient faults, and, unlike any previous work, actually proving it has strong fault-tolerance properties.

The specification and proof of fault tolerance comes in several stages. First, before proving any particular properties, it is necessary to define a fault model precisely. Most of the current literature uses the *Single Event Upset (SEU) Model*, which states that only one fault may occur during execution [15, 18, 25]. However, the details of exactly where and when faults may occur are usually given in English. We also assume the SEU model, but we specify exactly where by including faulty transitions as formal rules in the operational semantics of our assembly language.

Second, it is necessary to state precisely what "fault tolerance" actually means. Abstractly, a program is fault-tolerant if no fault can change the observable behavior of a program. More concretely, we assume our system operates in the presence of a memory-mapped output device, and hence a program is *not* fault-tolerant if a fault can cause a deviation in the sequence of values written to memory. We formalize this property more precisely as a mathematical theorem that relates faulty and non-faulty executions of a program.

Third, it is necessary to provide a technique for actually proving that specific programs are fault tolerant relative to the fault model. Our proof technique is presented in the form of a type system. All well-typed programs satisfy variants of the standard progress and preservation lemmas, even in the presence of transient faults, as well as the stronger fault tolerance property mentioned above. In addition to being theoretically important as a proof technique for fault tolerance, the type system can be used to debug compilers that intend to generate reliable code. If the output from these compilers type check, their code will have strong fault tolerance guarantees. In the past, researchers have proposed testing compiler outputs using fault injection techniques that randomly insert errors into programs. However, using a type checker in this case is a much better idea. In principle, a conventional testing technique would need to test all combinations of features *in conjunction with all combinations of faults*, causing an explosion in the number of test cases, and yet still failing to achieve perfect fault coverage in practice. By using the type checker we have designed, one achieves perfect fault coverage relative to the fault model without needing to increase the compiler test suite.

The rest of this paper presents the details of our hybrid hardware-software fault-tolerance technique. Section 2 presents the syntax and operational semantics of the new, idealized assembly language we have designed for fault tolerance. It is a RISC-based architecture with special instructions to facilitate reliable communication with memory and to detect control-flow faults. Section 3 presents the key principles and formal definitions for the fault-tolerant assembly language type system ($\mathrm{TAL}_{\mathrm{FT}}$ for short). Though the typing rules are specific to our particular setting, the underlying principles are more general; we believe many of these principles will apply to reasoning about related fault-tolerant systems. Our innovative combination of a TAL-like type-theory with concepts from classical Hoare Logics is a particularly general and important technical contribution. Section 4 describes the key theorems we have proven including Progress, Preservation, "No False Positives," and Fault Tolerance. Section 5 provides empirical evidence that our new hybrid solution to fault tolerance is feasible for many applications by measuring performance results on simulated hardware. Related work is discussed in more detail in Section 6.

2 The Faulty Hardware

The faulty hardware is based on a simple RISC architecture, extended with features to support detection of control-flow faults and safe interaction with memory-mapped output devices. Correct use of these features makes it possible to detect all faults that might change a program's observable behavior. Most practical systems also need a fault recovery mechanism of some kind. However, since recovery is largely orthogonal to detection, we omit the former, focusing only on the latter in this paper.

The general strategy of every fault-tolerant program is to maintain two redundant and independent threads of computation, a *green* (G) computation and a *blue* (B) computation. The green computation generally leads slightly, and the blue computation generally trails, though there is a fair amount of flexibility in how the instructions in each computation may be interleaved. Prior to writing data out to a memory-mapped output device, the results of the two computations are checked for equivalence. If the results are not equivalent, the machine will signal that a fault has been detected. The arguments to any control-flow transfer must also be checked for faults. This methodology has been show in the literature as an effective implementation of fault tolerance [13, 17], and we expand on this style of implementation by formalizing the fault model and coverage.

The execution of assembly programs is specified using a small-step operational semantics that maps *machine states* (Σ) to other machine states. These machine states are made up of a number of components. The first component is the machine's *register bank* R, which is a total function that maps register names to the values contained therein. The meta variable a ranges over all sorts of registers, and meta variable r ranges only over general-purpose registers (r_1, r_2, ...). In addition to general-purpose registers, there are two program counter registers (pc_G and pc_B), which contain the same value unless there has been the fault. There is one additional special register, the *destination register*, d. Its role in control-flow checking will be explained later.

To facilitate proofs of certain theorems, the value in each register is tagged with the color (either green or blue) of the computation to which it belongs. However, these tags have no effect on the run-time behavior of programs.[1]

In addition to a register bank, the machine state includes a *code memory* C, which we model as a function mapping integer addresses n to instructions.[2] The machine also has a *value memory* M, which maps addresses to integer values. In between the value memory and the processor is a special *store queue*, Q, which is used to detect faults before data is written to a memory-mapped output device. The store queue is a queue of address-value pairs. We will discuss the role of the queue in greater detail later.

Overall, an abstract machine state (Σ) may have the form *fault*, indicating the hardware has detected a transient fault, or the ordinary state (R, C, M, Q, ir), where the first four components are as discussed above, and ir is either an instruction i to be executed, or "$\cdot$" indicating the next instruction should be fetched from code memory. Figure 1 summarizes the syntax of machine states. Here and elsewhere in the paper, we use overbar notation to indicate a sequence of objects.

2.1 The Fault Model

The operational semantics is designed both to model proper execution of machine instructions and to make perfectly explicit, precise, and transparent all of our assumptions about when and where faults may occur. The central operational judgment has the form $\Sigma_1 \longrightarrow_k^s \Sigma_2$, which expresses a single step transition from state Σ_1 to state Σ_2 while incurring k faults and writing data s to a memory-mapped output device. We

[1] In contrast, the tags on instruction opcodes, to be introduced momentarily, *do* have an effect on evaluation.
[2] Address 0 is not considered a valid code address.

$$
\begin{array}{llcl}
\textit{colors} & c & ::= & G \mid B \\
\textit{colored values} & v & ::= & c\ n \\
\textit{registers} & r & ::= & r_n \\
\textit{general regs} & a & ::= & r \mid d \mid pc_c \\
\textit{register file} & R & ::= & \cdot \mid R, a \rightarrow v \\
\textit{code memory} & C & ::= & \cdot \mid C, n \rightarrow i \\
\textit{value memory} & M & ::= & \cdot \mid M, n \rightarrow n \\
\textit{store queue} & Q & ::= & \overline{(n, n)} \\
\textit{ALU ops} & op & ::= & add \mid sub \mid mul \\
\textit{instructions} & i & ::= & op\ r_d, r_s, r_t \mid op\ r_d, r_s, v \\
 & & \mid & ld_c\ r_d, r_s \mid st_c\ r_d, r_s \mid mov\ \ r_d, v \\
 & & \mid & bz_c\ \ r_z, r_d \mid jmp_c\ \ r_d \\
\textit{inst register} & ir & ::= & i \mid \cdot \\
\textit{state} & \Sigma & ::= & (R, C, M, Q, ir) \mid \textit{fault}
\end{array}
$$

Figure 1: Syntax of instructions and machine states.

will work under the standard assumption of a single upset event and hence k will always be either 0 or 1. The data s is a (possibly empty) sequence of address-value pairs. While the operational semantics models the internal workings of the machine, the only externally observable behavior of the machine is the sequence of writes s to the output device or the signaling of a hardware-detected fault. If faults cause the processor to have drastically different internal behavior, but the externally observable sequence s is unchanged, we consider the program to have executed successfully.

Different fault-tolerance techniques protect different components of machines. In the literature, the protected areas are usually inside the *Sphere of Replication* (SoR) [15]. In our case, we target faults that may occur in data manipulated within the processor. We assume that both code memory C and value memory M are fully protected. This is often the case since error-correcting codes can very efficiently protect memory. To make these assumptions explicit, the following three operational rules specify exactly how faults may occur within our system.

$$
\frac{R(a) = c\ n}{(R, C, M, Q, ir) \longrightarrow_1 (R[a \mapsto c\ n'], C, M, Q, ir)}\ (\textit{reg-zap})
$$

$$
\frac{\begin{array}{c} Q_1 = \overline{(n_1, n_1')}, (m_1, m'), \overline{(n_2, n_2')} \\ Q_2 = \overline{(n_1, n_1')}, (m_2, m'), \overline{(n_2, n_2')} \end{array}}{(R, C, M, Q_1, ir) \longrightarrow_1 (R, C, M, Q_2, ir)}\ (\textit{Q-zap1})
$$

$$
\frac{\begin{array}{c} Q_1 = \overline{(n_1, n_1')}, (m, m_1'), \overline{(n_2, n_2')} \\ Q_2 = \overline{(n_1, n_1')}, (m, m_2'), \overline{(n_2, n_2')} \end{array}}{(R, C, M, Q_1, ir) \longrightarrow_1 (R, C, M, Q_2, ir)}\ (\textit{Q-zap2})
$$

Rule *reg-zap* nondeterministically introduces a fault into any register by replacing the value in that register with some other arbitrary value. There are no restrictions on how the underlying value might be changed. For instance, code pointers can be changed to arbitrary integer values; references may no longer be in bounds. However, the color tag is preserved to facilitate fault-tolerance proofs. Since the color tag is fictional (has no effect on run-time behavior), this poses no limitation on the fault model. Rules Q_1-*zap* and Q_2-*zap* alter the contents of the store queue in similar ways.

Formally, these are the only faults that can occur. However, notice that since the program counters and targets of indirect jumps are susceptible to the *reg-zap* rule, we effectively capture many forms of "control-flow faults" studied previously. Notice also that we do not explicitly consider faults that occur *during* execution of an instruction. However, many such faults may easily be shown equivalent to correct execution of an

Instruction Fetch:

$$\frac{R_{val}(pc_G) = R_{val}(pc_B) \qquad R_{val}(pc_G) \in \ Dom(C)}{(R, C, M, Q, \cdot) \longrightarrow_0 (R, C, M, Q, C(R_{val}(pc_G)))} \ (\mathit{fetch})$$

$$\frac{R_{val}(pc_G) \neq R_{val}(pc_B)}{(R, C, M, Q, \cdot) \longrightarrow_0 \ fault} \ (\mathit{fetch\text{-}fail})$$

Basic Instructions:

$$\frac{}{(R, C, M, Q, op \ \ r_d, r_s, r_t) \longrightarrow_0 (R\texttt{++}[r_d \mapsto R_{col}(r_t) \ (R_{val}(r_s) \ op \ R_{val}(r_t))], C, M, Q, \cdot)} \ (\mathit{op2r})$$

$$\frac{}{(R, C, M, Q, op \ \ r_d, r_s, c \ n) \longrightarrow_0 (R\texttt{++}[r_d \mapsto c \ (R_{val}(r_s) \ op \ n)], C, M, Q, \cdot)} \ (\mathit{op1r})$$

$$\frac{}{(R, C, M, Q, mov \ \ r_d, v) \longrightarrow_0 (R\texttt{++}[r_d \mapsto v], C, M, Q, \cdot)} \ (\mathit{mov})$$

Figure 2: Operational rules for basic instructions

instruction composed with a fault either immediately before or afterwards. For example, consider a simple register-to-register add instruction. Any fault within the adder hardware during execution of the add is equivalent to a correct add followed by a fault in the destination register.

An important benefit of our formal model is that there is actually some precise, concrete specification to analyze. Moreover, if a researcher wants to reason about the consequences of some fault that lives outside the formal model, this may be done by adding a new operational rule to the system and studying its semantic effect.

2.2 Instruction Semantics

The syntax of machine instructions was presented along with the rest of the components of our abstract machine in Figure 1. The semantics is described formally by the inference rules in Figures 2, 3, and 4, and explained informally below. The formal rules use several notational conventions. For instance, if R is a register file then $R(a)$ is the contents of register a and $R[a \mapsto v]$ is the updated register file with register a mapped to v. $R\texttt{++}$ is the register file that results from incrementing both pc_G and pc_B by 1. If $R(a)$ is the colored value $c\ n$, we write $R_{val}(a)$ to denote n and $R_{col}(a)$ to denote c. The function $find(Q, n)$ produces the first pair (n, n') that appears in Q, or () if no pair (n, n') appears in Q.

Instruction Fetch. The machine operates by alternatively fetching an instruction from code memory and executing that instruction. When there is no current instruction to execute (i.e. $ir = \cdot$), the *fetch* rule should fire. This rule tests for equality of the two program counters to check for faults and loads the appropriate instruction from code memory. If pc_G and pc_B are the same but $R_{val}(pc_G)$ is not a valid address in code memory, execution "gets stuck" (no rule fires). Fortunately, however, well-typed programs never get stuck, even when a single fault occurs. On the other hand, a fault can render the two program counters inequivalent. In this case, rule *fetch-fail* fires and causes a transition to the fault state. Abstractly, this transition represents hardware detection of a transient fault. Controlled program termination or perhaps recovery may follow. Fault recovery is an orthogonal issue to fault detection, so we leave it unspecified here. The fault model does not allow for the instruction itself to be corrupted.

Basic Instructions. The arithmetic and move instructions (rules *op2r*, *op1r*, and *mov*) are completely standard. The first arithmetic operation *op* r_d, r_s, r_t performs *op* on the values in r_s and r_t, storing the result in r_d. The second arithmetic operation uses a constant operand v in addition to r_s and r_d. All constants are annotated with the color of the computation they belong to. Likewise, the *mov* instruction loads an annotated constant into a register.

Memory Instructions. Transient faults are problematic only when they change the results of computations and those results are *observed* by an external user. In our model, the only way a result can be observed is for a program to write it to memory, where a memory-mapped output device may read and process it.

Without special hardware it appears *impossible* to guarantee that storage operations guard access to memory properly. No matter what sophisticated software checking is performed just before a conventional store instruction, it will be undone if a fault strikes between the check and execution of the store instruction. This is the conundrum of the *Time-Of-Check-Time-Of-Use* (TOCTOU) fault.

To avoid TOCTOU faults, our machine possesses a modified store buffer (the queue Q), which is similar to the store buffer used in previous hardware [15] and hybrid [18] fault tolerant systems. In addition, there are two special storage instructions, each tagged with a color. The green store instruction $st_G\ r_d, r_s$ places the address-value pair $(R_{val}(r_d), R_{val}(r_s))$ on the front of the queue (rule st_G*-queue*). The blue store instruction $st_B\ r_d, r_s$ retrieves the pair (n_l, n'_l) on the back of the queue, checks that it equals $(R_{val}(r_d), R_{val}(r_s))$, and then stores it in memory (rule st_B*-mem*). If the pairs are different, the hardware signals a fault (rule st_B*-mem-fail*). Since green stores must always come before blue stores, instruction scheduling is somewhat constrained. As we will show later in Section 5, we have evaluated the performance both with and without this scheduling constraint and show that its performance impact is negligible.

As an example, consider the following straight-line sequence:

1 mov r_1, G 5
2 mov r_2, G 256
3 st$_G$ r_2, r_1
4 mov r_3, B 5
5 mov r_4, B 256
6 st$_B$ r_4, r_3

These six instruction have the effect of storing 5 into memory address 256. Moreover, a fault at any point in execution, to either blue or green values or addresses, will be caught by the hardware when the blue store (instruction 6) compares its operands to those in the queue. In addition, our instruction set gives a compiler the freedom to allocate registers however it chooses (*e.g.*, reusing registers 1 and 2 in instructions 4-6 instead of registers 3 and 4) and to change the instruction schedule in various ways (*e.g.*, moving instruction 3 to a position between instructions 5 and 6).

Interestingly, however, not all conventional optimizations are sound, and, of course, this is why type checking generated code can be so helpful in detecting compiler errors. For example, common subexpression elimination might result in the following code:

1 mov r_1, G 5
2 mov r_2, G 256
3 st$_G$ r_2, r_1
4 st$_B$ r_2, r_1

In this case, a fault in r_1 after instruction 1, or a fault in r_2 after instruction 2 will cause both instructions 3 and 4 to manipulate the same, but incorrect, address-value pair. The result would be to store an incorrect value at the correct location or a correct value at an incorrect location. Fortunately, the TAL$_{\mathrm{FT}}$ type system catches reliability errors like this one.

As mentioned in Section 2.1, many "intra-instruction" faults can be modeled by modifying the register file before or after the instruction. However, this is not the case for a fault that occurs during the execution of the st_B*-mem* rule in between the comparisons and the store. The hardware designer must implement structures

$\boxed{\Sigma \longrightarrow_k^s \Sigma'}$

$$\frac{}{(R, C, M, Q, st_G \ \ r_d, r_s) \longrightarrow_0 (R\text{++}, C, M, ((R_{val}(r_d), R_{val}(r_s)), Q), \cdot)} \ (st_G\text{-}queue)$$

$$\frac{R_{val}(r_d) = n_l \qquad R_{val}(r_s) = n_l'}{(R, C, M, (\overline{(n, n')}, (n_l, n_l')), st_B \ \ r_d, r_s) \longrightarrow_0^{(n_l, n_l')} (R\text{++}, C, M[n_l \mapsto n_l'], \overline{(n, n')}, \cdot)} \ (st_B\text{-}mem)$$

$$\frac{R_{val}(r_d) \neq \ n_l \ or \ R_{val}(r_s) \neq \ n_l'}{(R, C, M, (\overline{(n, n')}, (n_l, n_l')), st_B \ \ r_d, r_s) \longrightarrow_0 \ fault} \ (st_B\text{-}mem\text{-}fail)$$

$$\frac{find(Q, R_{val}(r_s)) = (R_{val}(r_s), n)}{(R, C, M, Q, ld_G \ \ r_d, r_s) \longrightarrow_0 (R\text{++}[r_d \mapsto G \ n], C, M, Q, \cdot)} \ (ld_G\text{-}queue)$$

$$\frac{find(Q, R_{val}(r_s)) = () \qquad R_{val}(r_s) \in \ Dom(M)}{(R, C, M, Q, ld_G \ \ r_d, r_s) \longrightarrow_0 (R\text{++}[r_d \mapsto G \ M(R_{val}(r_s))], C, M, Q, \cdot)} \ (ld_G\text{-}mem)$$

$$\frac{find(Q, R_{val}(r_s)) = () \qquad R_{val}(r_s) \notin \ Dom(M)}{(R, C, M, Q, ld_G \ \ r_d, r_s) \longrightarrow_0 \ fault} \ (ld_G\text{-}fail)$$

$$\frac{R_{val}(r_s) \in \ Dom(M)}{(R, C, M, Q, ld_B \ \ r_d, r_s) \longrightarrow_0 (R\text{++}[r_d \mapsto B \ M(R_{val}(r_s))], C, M, Q, \cdot)} \ (ld_B\text{-}mem)$$

$$\frac{R_{val}(r_s) \notin \ Dom(M)}{(R, C, M, Q, ld_B \ \ r_d, r_s) \longrightarrow_0 \ fault} \ (ld_B\text{-}fail)$$

$$\frac{find(Q, R_{val}(r_s)) = () \qquad R_{val}(r_s) \notin \ Dom(M)}{(R, C, M, Q, ld_G \ \ r_d, r_s) \longrightarrow_0 (R\text{++}[r_d \mapsto G \ n], C, M, Q, \cdot)} \ (ld_G\text{-}rand)$$

$$\frac{R_{val}(r_s) \notin \ Dom(M)}{(R, C, M, Q, ld_B \ \ r_d, r_s) \longrightarrow_0 (R\text{++}[r_d \mapsto B \ n], C, M, Q, \cdot)} \ (ld_B\text{-}rand)$$

Figure 3: Selected operational rules for memory instructions.

that detect or mask any faults that occur here. If the hardware designer cannot meet the specification given by the operational semantics, he acknowledges there may be a vulnerability.

The load instructions also come in pairs: ld_B and ld_G. The only difference in their semantics is that ld_G checks for a pending store in the queue before loading its value from memory, whereas ld_B goes directly to memory, ignoring the queue. This wrinkle increases the freedom in instruction scheduling by allowing the green computation to load a value it may have recently stored before the blue computation has necessarily committed the store. Rules ld_G*-queue*, ld_G*-mem*, and ld_B*-mem* specify these behaviors.

Notice that there is no mechanism for verifying the address used in loads. Hence, a fault can result in an invalid address. In practice such a load might induce a hardware exception such as a segmentation fault (rules ld_G*-fail* and ld_B*-fail*) or might result in loading some arbitrary value (rules ld_G*-rand* and ld_B*-rand*).

Control-Flow Instructions. Any change in the control-flow of a program may cause a different sequence of values to be stored and observed by an external user. Consequently, the hardware contains mechanisms to detect faults in addresses that serve as jump targets. Intuitively, these mechanisms mirror the solution to faults in stored data in that execution of a control-flow transfer is accomplished through two instructions. Our solution uses a combination of software and hardware control-flow protection that is similar to watchdog processors [6], but that makes both versions of the control flow explicit as in software-only control flow protection [12, 17].

To achieve an unconditional jump, one executes a jmp_G instruction first and a related jmp_B instruction at some point in the future. A $jmp_G\ r_1$ moves the destination address from r_1 into the special destination register d (rule jmp_G). Like the store queue, the destination register stores a programmer intention, initiated by the green computation. Later, when the blue computation attempts to commit the jump by executing a $jmp_B\ r_2$ instruction, the contents of r_2 are compared to the contents of the destination register and if they are equal, control jumps to that location (rule jmp_B). If the addresses are different, the hardware detects a fault (see rule jmp_B*-fail*). Similar to the constraint for the store queue, forcing green control flow instructions to be executed before the corresponding blue version constrains the instruction schedule. Section 5 will show that this scheduling constraint has only a minimal performance impact.

The following code illustrates a typical control-flow transfer.

$$
\begin{array}{l}
1\ \texttt{ld}_G\ r_1,\ r_2 \\
3\ \texttt{ld}_B\ r_3,\ r_4 \\
2\ \texttt{jmp}_G\ r_1 \\
4\ \texttt{jmp}_B\ r_3
\end{array}
$$

Initially, registers r_2 and r_4 should point to the same memory location, which contains a code pointer to jump to. The example illustrates some of the flexibility in scheduling jump instructions.

Conditional jumps are more complex, but follow the same principles. The green conditional $bz_G\ r_z,\ r_d$ tests r_z and if it is 0, moves the contents of r_d into destination register d (rules *bz-untaken* and bz_G*-taken*). No control-flow transfer occurs until a blue conditional $bz_B\ r'_z,\ r'_d$ tests the contents of its r'_z register. If r'_z is 0 then r'_d must equal the contents of d, and if so, the control flow transfer occurs (rule bz_B*-taken*). If r'_z is not 0, it is not good enough merely to fall through — the contents of r'_z might be faulty. To avoid this possibility, the instruction examines the destination register. If it is 0 (and hence a prior bz_G instruction did not store an address), the fall-through occurs (rule *bz-untaken*). Our metatheory will show that this mechanism suffices to detect faults either in the green computation (registers r_z and r_d) or the blue computation (registers r'_z and r'_d).

3 Typing

The primary goal of the $\mathrm{TAL}_{\mathrm{FT}}$ type system is to ensure that well-typed programs exhibit fail-safe behavior in the presence of transient faults. In other words, well-typed programs must guarantee that a memory-mapped output device can never read a corrupt value and make it visible to a user. We call this property "fault tolerance."

$\boxed{\Sigma \longrightarrow^s_k \Sigma'}$

$$\frac{R_{val}(d) = 0}{(R, C, M, Q, jmp_G \;\; r_d) \longrightarrow_0 (R\text{++}[d \mapsto R(r_d)], C, M, Q, \cdot)} \;(jmp_G)$$

$$\frac{R_{val}(d) \neq \; 0}{(R, C, M, Q, jmp_G \;\; r_d) \longrightarrow_0 \;\; fault} \;(jmp_G\text{-}fail)$$

$$\frac{R_{val}(d) \neq \; 0 \qquad R_{val}(r_d) = R_{val}(d)}{(R, C, M, Q, jmp_B \;\; r_d) \longrightarrow_0 (R[pc_G \mapsto R(d)][pc_B \mapsto R(r_d)][d \mapsto G\ 0], C, M, Q, \cdot)} \;(jmp_B)$$

$$\frac{R_{val}(r_d) \neq R_{val}(d) \;\; or \; R_{val}(d) = 0}{(R, C, M, Q, jmp_B \;\; r_d) \longrightarrow_0 \;\; fault} \;(jmp_B\text{-}fail)$$

$$\frac{R_{val}(d) = 0 \qquad R_{val}(r_z) \neq 0}{(R, C, M, Q, bz_c \;\; r_z, r_d) \longrightarrow_0 (R\text{++}, C, M, Q, \cdot)} \;(bz\text{-}untaken)$$

$$\frac{R_{val}(r_z) \neq \; 0 \qquad R_{val}(d) \neq \; 0}{(R, C, M, Q, bz_c \;\; r_z, r_d) \longrightarrow_0 \;\; fault} \;(bz\text{-}untaken\text{-}fail)$$

$$\frac{R_{val}(d) = 0 \qquad R_{val}(r_z) = 0}{(R, C, M, Q, bz_G \;\; r_z, r_d) \longrightarrow_0 (R\text{++}[d \mapsto R(r_d)], C, M, Q, \cdot)} \;(bz_G\text{-}taken)$$

$$\frac{R_{val}(r_z) = 0 \qquad R_{val}(d) \neq \; 0}{(R, C, M, Q, bz_G \;\; r_z, r_d) \longrightarrow_0 \;\; fault} \;(bz_G\text{-}taken\text{-}fail)$$

$$\frac{R_{val}(d) \neq 0 \qquad R_{val}(r_z) = 0 \qquad R_{val}(r_d) = R_{val}(d)}{(R, C, M, Q, bz_B \;\; r_z, r_d) \longrightarrow_0 (R[pc_G \mapsto R(d)][pc_B \mapsto R(r_d)][d \mapsto G\ 0], C, M, Q, \cdot)} \;(bz_B\text{-}taken)$$

$$\frac{R_{val}(r_z) = 0 \qquad (R_{val}(r_d) \neq R_{val}(d) \;\; or \; R_{val}(d) = 0)}{(R, C, M, Q, bz_B \;\; r_z, r_d) \longrightarrow_0 \;\; fault} \;(bz_B\text{-}taken\text{-}fail)$$

Figure 4: Selected operational rules for control flow instructions.

Static Expressions

$$
\begin{array}{llcl}
\textit{exp kinds} & \kappa & ::= & \kappa_{int} \mid \kappa_{mem} \\
\textit{exp contexts} & \Delta & ::= & \cdot \mid \Delta, x : \kappa \\
\textit{exps} & E & ::= & x \mid n \mid E \ \textit{op} \ E \mid \textit{sel} \ E_m \ E_n \mid \textit{emp} \mid \textit{upd} \ E_m \ E_{n_1} \ E_{n_2} \\
\textit{substitutions} & S & ::= & \cdot \mid S, E/x
\end{array}
$$

Types

$$
\begin{array}{lrcl}
\textit{zap tags} & Z & ::= & \cdot \mid c \\
\textit{basic types} & b & ::= & \textit{int} \mid \Theta \rightarrow \textit{void} \mid b \ \textit{ref} \\
\textit{reg types} & t & ::= & \langle c, b, E \rangle \mid E' = 0 \Rightarrow \langle c, b, E \rangle \\
\textit{reg file types} & \Gamma & ::= & \cdot \mid \Gamma, a \rightarrow t \\
\textit{result types} & RT & ::= & \Theta \mid \textit{void}
\end{array}
$$

Contexts

$$
\begin{array}{llcl}
\textit{heap typing} & \Psi & ::= & \cdot \mid \Psi, n : b \\
\textit{static context} & \Theta & ::= & \Delta; \Gamma; \overline{(E_d, E_s)}; E_m
\end{array}
$$

Figure 5: TAL_{FT} type syntax.

In the following sections, we explain the intuitions and principles behind the various elements of the type system. Throughout the discussion, the reader will notice that our typing rules are not syntax-directed. Of course, as with other sorts of typed assembly language or proof-carrying code, this fact presents no particular difficulty in practice — it is easy for a compiler to generate sufficient "typing hints" to make type reconstruction trivial. For the reader's reference, the objects used in the type system are presented in Figure 5.

3.1 Static Expressions

Our "type system" is actually a combination of two theories, one being a relatively simple type theory for assembly, inspired by previous work on TAL [8], and the second being a Hoare Logic, designed to enforce the more precise invariants required for strong fault tolerance. The latter component requires we define a language of *static expressions* for reasoning about values and storage.

For the purposes of this paper, the static expressions are drawn from the standard theory of arithmetic and arrays used in many classical Hoare Logics (*c.f.*, Necula's thesis [10]). These static expressions are classified as either integers (kind κ_{int}) or memories (kind κ_{mem}). The integer expressions include variables, constants, simple arithmetic operations, and values from a memory ($\textit{sel} \ E_m \ E_n$ is the integer located at address E_n in E_m). The memory expressions include variables, the empty memory (*emp*), and memory updates ($\textit{upd} \ E_m \ E_{n_1} \ E_{n_2}$ is a memory E_m updated so that address E_{n_1} stores value E_{n_2}).

The context Δ is a mapping from variables to kinds, and the judgment $\Delta \vdash E : \kappa$ classifies expression E as having kind κ. The judgment $\Delta \vdash S : \Delta'$ holds when the substitution S maps variables in $Dom(\Delta')$ to values well-formed in Δ with types in $Rng(\Delta')$. The judgment $\Delta \vdash E_1 = E_2$ is valid when E_1 and E_2 are equal objects in the standard model. The function $[\![E]\!]$ supplies the denotation of the closed static expression E as either an integer or a memory, depending on its kind. These definitions are shown in Figure 6.

3.2 Value Typing

Since faults strike values, corrupting their bit patterns in arbitrary ways, the subtleties of value typing are a key concern. Informally, the type system maintains three key pieces of information about every value:

1. *A color (green or blue).* The type system is organized to ensure that when a value is known to be green, its contents can only depend on the contents of other green values not blue ones, and likewise, blue

$\boxed{\Delta \vdash\ E : \kappa}$

$$\frac{x \in Dom(\Delta)}{\Delta \vdash\ x : \Delta(x)}\ (\textit{E-var-t})$$

$$\frac{}{\Delta \vdash\ n : \kappa_{int}}\ (\textit{E-int-t}) \qquad \frac{\Delta \vdash\ E_1 : \kappa_{int} \qquad \Delta \vdash\ E_2 : \kappa_{int}}{\Delta \vdash\ E_1\ op\ E_2 : \kappa_{int}}\ (\textit{E-op-t})$$

$$\frac{}{\Delta \vdash\ emp : \kappa_{mem}}\ (\textit{E-emp-t}) \qquad \frac{\Delta \vdash\ E_m : \kappa_{mem} \qquad \Delta \vdash\ E_n : \kappa_{int}}{\Delta \vdash\ sel\ E_m\ E_n : \kappa_{int}}\ (\textit{E-sel-t})$$

$$\frac{\Delta \vdash\ E_m : \kappa_{mem} \qquad \Delta \vdash\ E_{n_1} : \kappa_{int} \qquad \Delta \vdash\ E_{n_2} : \kappa_{int}}{\Delta \vdash\ upd\ E_m\ E_{n_1}\ E_{n_2} : \kappa_{mem}}\ (\textit{E-upd-t})$$

$\boxed{\Delta \vdash\ S : \Delta'}$

$$\frac{}{\Delta \vdash\ \cdot : \cdot}\ (\textit{sub-emp-t}) \qquad \frac{\Delta \vdash\ S : \Delta' \qquad \Delta \vdash\ E : \kappa \qquad x \notin Dom(\Delta) \cup Dom(\Delta')}{\Delta \vdash\ S, E/x : \Delta', x : \kappa}\ (\textit{sub-t})$$

$\boxed{[\![E]\!]}$

$$\begin{array}{lcl}
[\![n]\!] & = & n \\
[\![emp]\!] & = & \cdot \\
[\![E_1\ op\ E_2]\!] & = & [\![E_1]\!]\ op\ [\![E_2]\!] \\
[\![sel\ E_m\ E_n]\!] & = & [\![E_m]\!]([\![E_n]\!]) \\
[\![upd\ E_m\ E_1\ E_2]\!] & = & [\![E_m]\!][\ [\![E_1]\!] \mapsto [\![E_2]\!]\ ]
\end{array}$$

$\boxed{\Delta \vdash\ E_1 = E_2}$

$$\frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \\ \forall S.\ \cdot \vdash S : \Delta \implies [\![S(E_1)]\!] = [\![S(E_2)]\!]\end{array}}{\Delta \vdash\ E_1 = E_2}\ (\textit{E-eq}) \qquad \frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{int} \qquad \Delta \vdash E_2 : \kappa_{int} \\ \forall S.\ \cdot \vdash S : \Delta \implies [\![S(E_1)]\!] \neq [\![S(E_2)]\!]\end{array}}{\Delta \vdash\ E_1 \neq E_2}\ (\textit{E-neq})$$

$$\frac{\begin{array}{c}\Delta \vdash E_1 : \kappa_{mem} \qquad \Delta \vdash E_2 : \kappa_{mem} \\ \forall \ell \in\ Dom([\![S(E_1)]\!]) \cup Dom([\![S(E_2)]\!]).\ [\![S(E_1)]\!](\ell) = [\![S(E_2)]\!](\ell)\end{array}}{\Delta \vdash\ E_1 = E_2}\ (\textit{E-mem-eq})$$

Figure 6: Semantics of Static Expressions.

$\boxed{\Psi \vdash \ n : b}$

$$\frac{}{\Psi \vdash \ n : \mathit{int}}\ (\mathit{int\text{-}t}) \qquad \frac{}{\Psi \vdash \ n : \Psi(n)}\ (\mathit{base\text{-}t})$$

$\boxed{\Psi; \Delta \vdash^Z \ v : t}$

$$\frac{\Psi \vdash \ n : b \qquad \Delta \vdash \ E = n}{\Psi; \Delta \vdash^Z \ c\ n : \langle c, b, E \rangle}\ (\mathit{val\text{-}t})$$

$$\frac{n \neq 0 \qquad \Psi; \Delta \vdash^Z \ c\ n : \langle c, b, E \rangle \qquad \Delta \vdash \ E' = 0}{\Psi; \Delta \vdash^Z \ c\ n : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (\mathit{cond\text{-}t}) \qquad \frac{\Delta \vdash \ E' \neq 0}{\Psi; \Delta \vdash^Z \ c\ 0 : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (\mathit{cond\text{-}t\text{-}n0})$$

$$\frac{\Delta \vdash \ E : \kappa_{int}}{\Psi; \Delta \vdash^c \ c\ n : \langle c, b, E \rangle}\ (\mathit{val\text{-}zap\text{-}t}) \qquad \frac{\Delta \vdash \ E : \kappa_{int}}{\Psi; \Delta \vdash^c \ c\ n : E' = 0 \Rightarrow \langle c, b, E \rangle}\ (\mathit{val\text{-}zap\text{-}cond})$$

Figure 7: Value Typing.

can only depend upon blue. Hence, while a fault in a green value can eventually corrupt arbitrarily many other green values, it cannot corrupt any blue values, and vice versa.

2. *A "basic type".* When no fault has occurred in the value's color, the value's basic type describes its shape. Values with type *int* may have any bit pattern. Values with type $\Theta \rightarrow \mathit{void}$ are pointers to code (continuations). One must satisfy the precondition Θ before jumping to them. Values with type $b\ \mathit{ref}$ are pointers to values with type b.

3. *A static expression.* When there has been no fault in a value's color, the value exactly equals the static expression. Static expressions are used to guarantee that in the absence of faults, the green and blue computations produce equal values, and hence, dynamic fault detection checks always succeed.

To summarize, every value is typed using a triple $\langle c, b, E \rangle$, where c is a color, b is a basic type, and E is a static expression. The presence of the static expression makes this type a kind of singleton type.

Value Typing Judgment. The value typing judgment has the form $\Psi; \Delta \vdash^Z \ v : t$, where Ψ maps heap addresses to basic types, and Δ contains the free expression variables. In the rule *val-t*, a colored value $c\ n$ is given the type $\langle c, b, E \rangle$ when the static expression E is equal to n, and $\Psi \vdash n : b$. The judgment $\Psi \vdash n : b$ allows n to be given either the basic type *int* or the type of the address n in memory.

The two rules *cond-t* and *cond-t-n0* are used to type the conditional type $(E' = 0 \Rightarrow \langle G, \Theta \rightarrow \mathit{void}, E'_r \rangle)$. When the static expression E' is equal to zero, values of this type also have type $\langle G, \Theta \rightarrow \mathit{void}, E'_r \rangle$. When E' is not equal to zero, values with this type must be 0.

The final two rules for $\Psi; \Delta \vdash^Z \ v : t$ make use of the *zap tag* Z, which is either empty or a color c. If the zap tag is a color c, then there may have been a fault affecting data of that color. Data colored the same as the zap tag can be given any type, as it may have been arbitrarily corrupted. The static expression used in this type may not contain any free expression variables.

Value Subtyping. There is also a subtyping relation $\Delta \vdash t \leq t'$ that allows all types $\langle c, b, E_1 \rangle$ to be subtypes of $\langle c, \mathit{int}, E_2 \rangle$ when $\Delta \vdash E_1 = E_2$. This relation is extended to register file subtyping $\Delta \vdash \Gamma_1 \leq \Gamma_2$, by requiring that the type of each general-purpose register in Γ_2 be a supertype of the corresponding register in Γ_1. Note that here is no required relationship between the special registers d, pc_G, and pc_B. The rules for these judgments appear in Figure 8.

$\boxed{\Delta \vdash\ t \leq t'}$

$$\frac{\Delta \vdash\ E_1 = E_2}{\Delta \vdash\ \langle c, b, E_1 \rangle \leq \langle c, b, E_2 \rangle}\ (\textit{subtp-triple})$$

$$\frac{\Delta \vdash\ E_1 = E_2}{\Delta \vdash\ \langle c, b, E_1 \rangle \leq \langle c, \mathit{int}, E_2 \rangle}\ (\textit{subtp-int})$$

$$\frac{\Delta \vdash\ t \leq t' \qquad \Delta \vdash\ E = E'}{\Delta \vdash\ (E = 0 \Rightarrow t) \leq (E' = 0 \Rightarrow t')}\ (\textit{subtp-cond})$$

$\boxed{\Delta \vdash\ \Gamma_1 \leq \Gamma_2}$

$$\frac{\forall r \in \mathit{Dom}(\Gamma_2).\ \Gamma_1(r) \leq \Gamma_2(r)}{\Delta \vdash\ \Gamma_1 \leq \Gamma_2}\ (\textit{reg-file-comp})$$

Figure 8: Subtyping.

3.3 Instruction Typing

While many of the instruction typing rules are quite complex, the essential principles guiding their construction may be summarized as follows.

1. *In the absence of faults, standard type theoretic principles should be valid.* In order to guarantee basic safety properties, the type system checks standard properties in much the same manner as previous typed assembly languages [8]. For example, jump targets must have code types, while loads and stores must operate over values with reference types.

2. *Green values only depend on other green values, and blue values only depend on blue values.* When this invariant is maintained, a fault in a blue value can never corrupt a green value and vice versa.

3. *Both green and blue computations have equal say in any dangerous actions.* Dangerous actions include storing values to memory-mapped output devices and executing control-flow operations. When both blue and green computations are involved, a fault in just one color is insufficient to deceive the hardware fault detection mechanisms.

4. *In the absence of faults, green and blue computations must compute identical values.* To be more precise, green and blue computations must store identical values to identical storage locations and must issue orders to transfer control to identical addresses. If not, the hardware will claim to detect faults when there have been none, or alternatively, might exhibit incorrect behaviors when there is a fault.

The first three principles are relatively straightforward to enforce. The fourth principle leads to the most technical challenges as it requires we check equality constraints between values. Moreover, since construction of these values depends on storage, the type system must maintain a relatively accurate static representation of storage. We accomplish this latter challenge using techniques drawn from Hoare Logics. The former challenge (testing values for equality) is achieved through the use of the singleton types described earlier.

The Instruction Typing Judgment. The judgment for typing instructions has the form $\Psi; \Theta \vdash \mathit{ir} \Rightarrow \mathit{RT}$. Unlike the context Ψ, which only contains invariant heap typing assumptions, Θ contains fine-grained context-sensitive information about the current state of memory and the register file. More specifically, Θ consists of the following subcontexts: (1) Δ, which describes the free expression variables appearing in the

$$\boxed{\Psi;\Theta \vdash\ ir\ \Rightarrow RT}$$

$$\frac{}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ \cdot\ \Rightarrow (\Delta;\Gamma;\overline{(E_d,E_s)};E_m)}\ (\cdot\text{-}t)$$

$$\frac{\Gamma(r_s) = \langle c, int, E_s' \rangle \qquad \Gamma(r_t) = \langle c, int, E_t' \rangle}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ op\ \ r_d, r_s, r_t\ \Rightarrow (\Delta;\Gamma\text{++}[r_d \mapsto\ \langle c, int, E_s'\ op\ E_t' \rangle];\overline{(E_d,E_s)};E_m)}\ (op2r\text{-}t)$$

$$\frac{\Gamma(r_s) = \langle c, int, E_s' \rangle}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ op\ \ r_d, r_s, c\ n\ \Rightarrow (\Delta;\Gamma\text{++}[r_d \mapsto\ \langle c, int, E_s'\ op\ n \rangle];\overline{(E_d,E_s)};E_m)}\ (op1r\text{-}t)$$

$$\frac{\Psi;\Delta \vdash\ v : t}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ mov\ \ r_d, v\ \Rightarrow (\Delta;\Gamma\text{++}[r_d \mapsto\ t];\overline{(E_d,E_s)};E_m)}\ (mov\text{-}t)$$

$$\frac{\Gamma(r_s) = \langle G, b\ ref, E_s' \rangle \qquad E\ =\ sel\ (\overline{upd}\ E_m\ \overline{(E_d,E_s)})\ E_s'}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ ld_G\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma\text{++}[r_d \mapsto\ \langle G, b, E \rangle];\overline{(E_d,E_s)};E_m)}\ (ld_G\text{-}t)$$

$$\frac{\Gamma(r_s) = \langle B, b\ ref, E_s' \rangle \qquad E\ =\ sel\ E_m\ E_s'}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ ld_B\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma\text{++}[r_d \mapsto\ \langle B, b, E \rangle];\overline{(E_d,E_s)};E_m)}\ (ld_B\text{-}t)$$

$$\frac{\Gamma(r_d) = \langle G, b\ ref, E_d' \rangle \qquad \Gamma(r_s) = \langle G, b, E_s' \rangle}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)};E_m) \vdash\ st_G\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma\text{++};(E_d',E_s'),\overline{(E_d,E_s)};E_m)}\ (st_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_d) = \langle B, b\ ref, E_d'' \rangle \quad \Gamma(r_s) = \langle B, b, E_s'' \rangle \\ \Delta \vdash\ E_d' = E_d'' \quad \Delta \vdash\ E_s' = E_s''\end{array}}{\Psi;(\Delta;\Gamma;\overline{(E_d,E_s)},(E_d',E_s');E_m) \vdash\ st_B\ r_d\ r_s\ \Rightarrow (\Delta;\Gamma\text{++};\overline{(E_d,E_s)};upd\ E_m\ E_d'\ E_s')}\ (st_B\text{-}t)$$

Figure 9: Instruction Typing Rules for Basic Instructions and Memory Instructions

$\boxed{\Psi;\Theta \vdash\ ir\ \Rightarrow RT}$

$$\frac{\begin{array}{c}\Gamma(d) = \langle G, int, 0\rangle \\ \Gamma(r_z) = \langle G, int, E_z\rangle \\ \Gamma(r_d) = \langle G, \Theta \rightarrow void, E'_d\rangle \\ \Theta = (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \\ \Gamma'(d) = \langle G, int, 0\rangle\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \vdash\ bz_G\ r_z\ r_d\ \Rightarrow (\Delta; \Gamma\text{++}[d \mapsto\ E_z = 0 \Rightarrow \langle G, \Theta \rightarrow void, E'_d\rangle]; \overline{(E_d, E_s)}; E_m)}\ (bz_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(r_z) = \langle B, int, E_z\rangle \\ \Gamma(r_d) = \langle B, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \rightarrow void, E_r\rangle \\ \Gamma(d) = E'_z = 0 \Rightarrow \langle G, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \rightarrow void, E'_r\rangle \\ \Delta \vdash\ E_z = E'_z \\ \Delta \vdash\ E_r = E'_r \\ \exists S.\Delta \vdash\ S : \Delta' \\ S(\Gamma')(d) = \langle G, int, 0\rangle \\ S(\Gamma')(pc_G) = \langle G, int, E'_r\rangle \\ S(\Gamma')(pc_B) = \langle B, int, E_r\rangle \\ \Delta \vdash \Gamma \leq\ S(\Gamma') \\ \Delta \vdash\ \overline{(E_d, E_s)} = S(\overline{(E'_d, E'_s)}) \\ \Delta \vdash\ E_m = S(E'_m)\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \vdash\ bz_B\ r_z\ r_d\ \Rightarrow (\Delta; \Gamma\text{++}; \overline{(E_d, E_s)}; E_m)}\ (bz_B\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(d) = \langle G, int, 0\rangle \\ \Gamma(r_d) = \langle G, \Theta \rightarrow void, E_{rd'}\rangle \\ \Theta = (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \\ \Gamma'(d) = \langle G, int, 0\rangle\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \vdash\ jmp_G\ r_d\ \Rightarrow (\Delta; \Gamma\text{++}[d \mapsto\ \langle G, \Theta \rightarrow void, E_{rd'}\rangle]; \overline{(E_d, E_s)}; E_m)}\ (jmp_G\text{-}t)$$

$$\frac{\begin{array}{c}\Gamma(d) = \langle G, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \rightarrow void, E'_r\rangle \\ \Gamma(r_d) = \langle B, (\Delta'; \Gamma'; \overline{(E'_d, E'_s)}; E'_m) \rightarrow void, E_r\rangle \\ \Delta \vdash\ E_r = E'_r \\ \exists S.\Delta \vdash\ S : \Delta' \\ S(\Gamma')(d) = \langle G, int, 0\rangle \\ S(\Gamma')(pc_G) = \langle G, int, E'_r\rangle \\ S(\Gamma')(pc_B) = \langle B, int, E_r\rangle \\ \Delta \vdash \Gamma \leq\ S(\Gamma') \\ \Delta \vdash\ \overline{(E_d, E_s)} = S(\overline{(E'_d, E'_s)}) \\ \Delta \vdash\ E_m = S(E'_m)\end{array}}{\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \vdash\ jmp_B\ r_d\ \Rightarrow void}\ (jmp_B\text{-}t)$$

Figure 10: Instruction Typing Rules for Control Flow Instructions.

other context-sensitive objects, (2) Γ, which describes the mapping of register names to types for register values, (3) $\overline{(E_d, E_s)}$, which describes the values in the queue, and (4) E_m, which describes memory, as one does in Hoare Logic.

The "result" of checking an instruction is a result type RT. A result type may either be *void*, indicating control does not proceed past the instruction (it is a jump), or a postcondition Θ', which describes the state of memory and the register file after execution of the instruction.

The typing rules are defined using several notational abbreviations. The notation Γ++ adds one to the static expression associated with each program counter register in Γ. The expression $\overline{upd}\ E_m\ \overline{(E_d, E_s)}$ is $(upd\ (...(upd\ E_m\ E_{d_k}\ E_{s_k})...)\ E_{d_1}\ E_{s_1})$ when $\overline{(E_d, E_s)} = ((E_{d_1}, E_{s_1}), ..., (E_{d_k}, E_{s_k}))$. Figures 9 and 10 presents the typing rules for instructions, and the following paragraphs explain the main points of interest.

Typing Basic Instructions. Basic arithmetic operations are not "dangerous" to execute, so the definitions of their typing rules are driven by principles 1 and 2, mentioned earlier. Consider, for example, rule *op2r-t* for an arithmetic operation *op*. This rule requires that the operand registers contain integers with the same color c in accordance with principal 2 (green depends on green, blue depends on blue). The result register r_d has a type colored c as well. In accordance with principle 1, the result has integer type. The rule also states that the static expression describing the result register is $E'_s\ op\ E'_t$ and that the state of the queue and memory are unchanged by evaluation of the instruction.

Typing Memory Instructions. Store operations are "dangerous" — they make computed values observable by the outside world — so we must be particularly careful in the formulation of their typing rules. In accordance with principle 1, both green and blue store instructions (rules st_G-t and st_B-t) require that the address register has the basic type *b ref* and the value register has the corresponding basic type b. Intuitively, the store queue is a green object, and in accordance with principle 2, the green store instruction may push an address-value pair onto the front the queue as long as both values are green. In accordance with principle 4, the rule for the blue store checks that the address-value pair to be stored is exactly equal to the address-value pair at the end of the queue. Since the arguments to the blue store have a blue type and the queue always contains green objects, both blue and green computations contribute to the actual storage operation (in accordance with principle 3).

The load operations are somewhat simpler than the store instructions since they are not "dangerous" in our model. However, like the store instructions, the operands of blue loads must be blue and the operands of green loads must be green. Once again, in accordance with principle 2, the result of a blue load is value with a blue type and likewise for a green load.

Typing Control-Flow Instructions. While the typing rules for control-flow instructions have many premises, they continue to follow the same four principles as the other instructions. Much of the complexity is inherently due to principle 1, which mandates checking all the usual constraints associated with jumps in any typed assembly language.

The simplest rule involves the green unconditional jump. This instruction is just a move from register r_d to the special destination register d. The type of register d is updated to the type of r_d (obeying both principles 1 and 2). The rule contains constraints that d must be equal to 0 in both Γ and Γ' since the hardware resets the destination register to 0 after a jump.

The blue unconditional jump is a true jump. According to principle 1, it checks the standard typing invariants needed to ensure safety in any typed assembly language, including (1) that the jump target has code type (see the first two premises), and (2) that the current state, including register file, memory, and queue, matches the expected state at the jump target, modulo some substitution S of static expressions for universally quantified variables Δ from the code type (see the final seven premises).

The typing of the conditional branches is quite similar to that of unconditional jumps. One difference is that the bz_G instruction is now a conditional move as opposed to an unconditional move. Hence, to represent the result of the move (unknown at compile time) the conditional type $(E'_z = 0 \Rightarrow \langle G, \Theta \rightarrow void, E'_r \rangle)$ is used. In addition, since the conditional branch may fall through, the result of typing the bz_G instruction is a proper postcondition as opposed to *void*, like jmp_G.

3.4 Machine State Typing

In order to prove various properties of the type system, we need to specify the invariants of machine states that are preserved during execution. The judgments for typing a machine state Σ are shown in Figure 11 and explained below.

Register File Typing. The judgment $\Psi \vdash^Z R : \Gamma$ states that the register file R has the register file type Γ under heap typing Ψ and a zap tag Z. The contents of each register must have the type given to that register by Γ. Each program counter must have the appropriate color, and the program counters must compute equal values. (In the case where one program counter is corrupted, the zap tag Z in the first premise allows its actual value to differ from the expected computed value.)

Code Typing. The judgment $\Psi \vdash C$ states that code memory C is well-formed with respect to heap typing Ψ. The address 0 is not a valid code address. Each address must have a code type, and the code type must contain the precondition for the instruction at that address. If the instruction typing results in a postcondition Θ' (meaning that control may fall through to the next instruction) then the subsequent instruction must be well typed using Θ' as its precondition.

Memory Typing. The judgment $\Psi \vdash M : E_m$ states that given heap typing Ψ the value memory M is well-formed and can be described by the static expression E_m. The static expression E_m must have kind κ_{mem}, and M must be the denotation of E_m. Each location in the domain of M must have a type b *ref* and the contents of that location must have type b.

Queue Typing. The judgment $\Psi \vdash^Z Q : \overline{(E_d, E_s)}$ means that queue Q can be described by the sequence of static expressions $\overline{(E_d, E_s)}$ given heap typing Ψ and zap tag Z. When the queue is empty, it is described by the empty sequence. When the zap tag Z is not G, the first pair (n_1, n_2) must consist of an address n_1 with type b *ref* and a value n_2 with type b. This pair is described by the static expression pair (E_d, E_s) when E_d evaluates to n_1 and E_s evaluates to n_2. The remainder of the queue must be described by the remainder of the static expression sequence. All values in the queue are considered to be green, so when the zap tag is G, these values may have been arbitrarily corrupted. Accordingly in this case, the only requirements are that each static expression must have kind κ_{int} and the length of the queue must be the same as the length of the static expression sequence.

Machine State Typing. The judgment $\vdash^Z \Sigma$ states that a machine state Σ is well-typed under zap tag Z. This judgment holds when Σ is a five-tuple (R, C, M, Q, ir), and these elements are each well-typed and consistent with each other. Note that Σ is not well-typed when it is the fault state *fault*.

The domain of the heap typing Ψ must be the union of the domains of the code memory C and the value memory M. When Z is not equal to G, the queue has not been corrupted, and so each address in Q is also a valid address in M. The code memory C must be well-formed with respect to the heap typing Ψ.

The zap tag Z must be either empty or colored blue or green. At least one of the program counters, and possibly both, will not be colored by Z, and therefore must not be corrupted. If ir is an instruction, these correct program counters must point to an address that contains ir. The heap typing Ψ gives this address a code type $(\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \to void$.

The context Δ contains the free variables in Γ, $\overline{(E_d, E_s)}$, and E_m. There must be some substitution S that gives values to each of the variables in Δ. Value memory M must be well-formed and described by the static expression $S(E_m)$. The queue Q must be described by $S\overline{(E_d, E_s)}$. The register file R must have type $S(\Gamma)$.

$\boxed{\Psi \vdash^{Z} \; R : \Gamma}$

$$\frac{\begin{array}{c}\forall a.\ \Psi; \cdot \vdash^{Z} \; R(a) : \Gamma(a) \\ \cdot \vdash \Gamma(pc_G) \leq \langle G, int, E_G \rangle \qquad \cdot \vdash \Gamma(pc_B) \leq \langle B, int, E_B \rangle \qquad \cdot \vdash \; E_G = E_B\end{array}}{\Psi \vdash^{Z} \; R : \Gamma} \; (R\text{-}t)$$

$\boxed{\Psi \vdash \; C}$

$$\frac{\begin{array}{c}0 \notin Dom(C) \\ \forall n \in Dom(C).\ \Psi(n) = \Theta \to void \;\wedge\; \Psi; \Theta \vdash \; C(n) \Rightarrow RT \wedge \; (RT = \Theta' \; implies \; \Psi(n+1) = \Theta' \to void)\end{array}}{\Psi \vdash \; C} \; (C\text{-}t)$$

$\boxed{\Psi \vdash \; M : E_m}$

$$\frac{\cdot \vdash E_m : \kappa_{mem} \qquad [\![E_m]\!] = M \qquad \forall \ell \in Dom(M).\ \; \Psi \vdash \ell : b \; ref \;\wedge\; \Psi \vdash M(\ell) : b}{\Psi \vdash \; M : E_m} \; (M\text{-}t)$$

$\boxed{\Psi \vdash^{Z} \; Q : \overline{(E_d, E_s)}}$

$$\frac{}{\Psi \vdash^{Z} \; () : ()} \; (Q\text{-}emp\text{-}t)$$

$$\frac{\begin{array}{c}Z \neq G \\ \Psi \vdash n_1 : b \; ref \qquad \Psi \vdash n_2 : b \qquad \cdot \vdash E_d = n_1 \qquad \cdot \vdash E_s = n_2 \\ \Psi \vdash^{Z} \; \overline{(n_1', n_2')} : \overline{(E_d', E_s')}\end{array}}{\Psi \vdash^{Z} \; (n_1, n_2), \overline{(n_1', n_2')} : (E_d, E_s), \overline{(E_d', E_s')}} \; (Q\text{-}t)$$

$$\frac{\cdot \vdash \; E_d : \kappa_{int} \qquad \cdot \vdash \; E_s : \kappa_{int} \qquad \Psi \vdash^{G} \; \overline{(n_1', n_2')} : \overline{(E_d', E_s')}}{\Psi \vdash^{G} \; (n_1, n_2), \overline{(n_1', n_2')} : (E_d, E_s), \overline{(E_d', E_s')}} \; (Q\text{-}zap\text{-}t)$$

$\boxed{\vdash^{Z} (R, C, M, Q, ir)}$

$$\frac{\begin{array}{c}Dom(\Psi) = Dom(C) \cup Dom(M) \\ Z \neq G \implies Dom(Q) \subseteq Dom(M) \\ \Psi \vdash C \\ \forall c \neq Z.\ ir \neq \; \cdot \; \implies \; C(R_{val}(pc_c)) = ir \\ \forall c \neq Z.\ \Psi(R_{val}(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \to void \\ \exists S.\ \cdot \vdash S : \Delta \\ \Psi \vdash M : S(E_m) \\ \Psi \vdash^{Z} Q : S\overline{(E_d, E_s)} \\ \Psi \vdash^{Z} R : S(\Gamma)\end{array}}{\vdash^{Z} (R, C, M, Q, ir)} \; (\Sigma\text{-}t)$$

Figure 11: Machine State Typing.

4 Formal Results

In this section, we present our formal results, including sketches of the corresponding proofs. We have carried through all proof details on paper, but have elected not to type set them. The following sketches should suffice for the reader to reconstruct the full details.

Section 4.1 discusses useful lemmas used throughout the proofs. Section 4.2 proves the standard notions of type safety. Section 4.3 defines a multistep operation semantics and some associated lemmas, including the No False Positives Corollary. Section 4.4 contains the Fault Tolerance Theorem and its associated lemmas. Each lemma and theorem is preceeded by a brief English explanation of its role in the proof, and followed by details on how the proof was constructed.

4.1 Lemmas

The proofs of the theorems in the remainder of this section rely on the lemmas discussed below.

4.1.1 Properties of Static Expressions

When an expression is closed, applying substitutions to that expression results in a syntactically equivalent expression.

Lemma 1 (Substituting Closed Expressions)
1. *If* $\cdot \vdash E : \kappa$ *then* $\forall S.\ S(E) = E$

Proof. *By induction on the structure of* $\cdot \vdash E : \kappa$.

Even though $[\![E]\!]$ is a partial function, it is always defined over well-kinded closed expressions.

Lemma 2 (Expression Denotation)
1. *If* $\cdot \vdash E : \kappa_{int}$ *then* $\exists\ n.\ [\![E]\!] = n$.
2. *If* $\cdot \vdash E : \kappa_{mem}$ *then* $\exists\ M.\ [\![E]\!] = M$.

Proof. *By induction on the structure of* $\cdot \vdash E : \kappa$.

The equality of expressions is transitive.

Lemma 3 (Expression Equality Transitivity)
If $\Delta \vdash E_1 = E_2$ *and* $\Delta \vdash E_2 = E_3$ *then* $\Delta \vdash E_1 = E_3$.

Proof. *By inspection of the definition of* $\Delta \vdash E_1 = E_2$.

Substituting an expression of kind κ for a free variable of type κ preserves typing.

Lemma 4 (Substitution Lemma)
1. *If* $\Delta, x : \kappa \vdash E' : \kappa'$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E'[E/x] : \kappa'$.
2. *If* $\Delta, x : \kappa \vdash E_1 = E_2$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E_1[E/x] = E_2[E/x]$.
3. *If* $\Delta, x : \kappa \vdash E_1 \neq E_2$ *and* $\Delta \vdash E : \kappa$ *then* $\Delta \vdash E_1[E/x] \neq E_2[E/x]$.
4. *If* $\Psi; \Delta, x : \kappa \vdash^Z v : t$ *and* $\Delta \vdash E : \kappa$ *then* $\Psi; \Delta \vdash^Z v : t[E/x]$.

5. *If* $\Psi; (\Delta, x : \kappa; \Gamma; \overline{(E_d, E_s)}; E_m) \vdash^Z ir \Rightarrow RT$ *and* $\Delta \vdash E : \kappa$ *then* $\Psi; (\Delta; \Gamma[E/x]; \overline{(E_d, E_s)}[E/x]; E_m[E/x]) \vdash^Z ir \Rightarrow RT[E/x]$.

6. *If* $\cdot \vdash S : \Delta$ *and* $\Psi; \Delta \vdash^Z v : t$ *then* $\Psi; \cdot \vdash^Z v : S(t)$.

7. *If* $\cdot \vdash S : \Delta$ *and* $\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \vdash^Z ir \Rightarrow (\Delta; \Gamma'; \overline{(E'_d, E'_s)}; E'_m)$ *then* $\Psi; (\cdot; S(\Gamma); S(\overline{(E_d, E_s)}); S(E_m)) \vdash^Z ir \Rightarrow (\cdot; S(\Gamma'); S(\overline{(E'_d, E'_s)}); S(E'_m))$.

Proof. *Parts 1, 4, and 5 – by induction on the respective typing derivation. Parts 2 and 3 – by inspection of the equality judgment definition. Parts 6 and 7 – by induction on the size of* Δ*, using parts 4 and 5 respectively.*

4.1.2 Properties of Well-Typed Values

Well-typed values are described by static expressions denoting integers and not memories.

Lemma 5 (Value Kinding)

1. *If* $\Psi; \Delta \vdash^Z v : \langle c, b, E\rangle$ *then* $\Delta \vdash E : \kappa_{int}$.

2. *If* $\Psi; \Delta \vdash^Z v : (E' = 0) \Rightarrow \langle c, b, E\rangle$ *then* $\Delta \vdash E : \kappa_{int}$.

Proof. *By case analysis on the value typing judgment.*

The type of a value gives us information about the shape of the value.

Lemma 6 (Canonical Forms)
If $\Psi; \cdot \vdash^Z c\ n : t$ *and* $Dom(\Psi) = Dom(C) \cup Dom(M)$ *and* $\Psi \vdash M : E_m$ *and* $\Psi \vdash C$*, then*

1. *If* $t = \langle c, b, E\rangle$ *or* $t = (E' = 0) \Rightarrow \langle c, b, E\rangle$ *and* $c = Z$ *then no particular properties of* n *are known.*

2. *If* $t = \langle c, int, E\rangle$ *and* $c \neq Z$ *then* $\cdot \vdash E = n$.

3. *If* $t = \langle c, \Theta \to void, E\rangle$ *and* $c \neq Z$ *then* $\Psi(n) = \Theta \to void$ *and* $n \in Dom(C)$ *and* $\cdot \vdash E = n$ *and* $n \neq 0$.

4. *If* $t = \langle c, b\ ref, E\rangle$ *and* $c \neq Z$ *then* $\Psi(n) = b\ ref$ *and* $n \in Dom(M)$ *and* $\cdot \vdash E = n$.

5. *If* $t = (E' = 0) \Rightarrow t'$ *and* $c \neq Z$ *and* $\cdot \vdash E' = 0$ *then* $n \neq 0$.

6. *If* $t = (E' = 0) \Rightarrow t'$ *and* $c \neq Z$ *and* $\cdot \vdash E' \neq 0$ *then* $n = 0$.

Proof. *By induction on the structure of* $\Psi; \cdot \vdash^Z c\ n : t$.

If a value has a type, and this type has a supertype, then the value also has the supertype.

Lemma 7 (Subtyping)
If $\Psi; \Delta \vdash^Z v : t$ *and* $\Delta \vdash t \leq t'$ *then* $\Psi; \Delta \vdash^Z v : t'$.

Proof. *By induction on the derivation of* $\Psi; \Delta \vdash^Z v : t$.

If a value is well-typed under the empty zap tag, then that value is well-typed under all colors.

Lemma 8 (Color Weakening)
If $\Psi; \cdot \vdash v : t$ *then* $\forall c.\ \Psi; \cdot \vdash^c v : t$.

Proof. *By induction on the value typing judgment.*

Color weakening extends to register files.

Lemma 9 (Register File Color Weakening)
If $\Psi \vdash R : \Gamma$ *then* $\forall c.\ \Psi \vdash^c R : \Gamma$.

Proof. *By inversion of the register file typing rule* R-t *and the Color Weakening Lemma.*

4.1.3 Properties of Well-typed Memories

Well-typed programs with no faults only load values from valid locations.

Lemma 10 (Well-typed Domain)
If $\vdash (R, C, M, Q,\ ld_G\ r_d,\ r_s)$ *then* $R_{val}(r_d) \in Dom(M)$.

Proof. *By inversion of* $\vdash (R, C, M, Q, ir)$*, inversion of the* ld_G-t *typing rule, and the Canonical Forms lemma.*

When looking up a value in a memory with an update, if the updated location is not the requested location, then this is equivalent to looking up the location in the memory without the update.

Lemma 11 (Irrelevant Update)
If $E = sel\ (upd\ E_m\ E_d\ E_s)\ E_n$ and $\cdot \vdash E_d \neq E_n$ *then* $E = sel\ E_m\ E_n$.

Proof. *By inspection of the denotation of sel and upd.*

4.1.4 Properties of Well-typed Queues

The length of the queue is the same as the length of the sequence that describes it. When the zap tag is not green, then each item in the queue is described by the corresponding expression. In addition, the first element of each pair has type *b ref* and its value has type *b*.

Lemma 12 (Queue)

1. *If* $\Psi \vdash^Z Q : \overline{(E_d, E_s)}$ *then* $length(Q) = length(\overline{(E_d, E_s)})$.
2. *If* $\Psi \vdash^Z \overline{(n_1, n_2)} : \overline{(E_d, E_s)}$ and $Z \neq G$ *then for all* k *from 1 to* $length(\overline{(n_1, n_2)})$, $\cdot \vdash E_{dk} = n_{1k}$ and $\cdot \vdash E_{sk} = n_{2k}$ *and there is some base type* b *such that* $\Psi \vdash n_{1k} : b\ ref$ and $\Psi \vdash n_{2k} : b$.

Proof. *By induction on the structure of* $\Psi \vdash^Z Q : \overline{(E_d, E_s)}$.

If the $find$ function returns no match on an address n_1, and the queue is described by the sequence of address-value pairs $\overline{(E_d, E_s)}$, then none of the address expressions are equal to n_1.

Lemma 13 (Find)
If $find(Q, n_1) = ()$ and $\Psi \vdash Q : \overline{(E_d, E_s)}$ *then for* k *from 1 to* $length(Q)$, $\cdot \vdash E_{dk} \neq n_1$.

Proof. *By definition of the $find$ function and the queue typing judgment.*

If a queue is well-typed under the empty zap tag, then it is also well-typed under any colored zap tag.

Lemma 14 (Queue Color Weakening)
If $\Psi \vdash Q : \overline{(E_d, E_s)}$ *then* $\forall c.\ \Psi \vdash^c Q : \overline{(E_d, E_s)}$.

Proof. *By induction on the queue typing judgment.*

4.2 Type Safety

Progress states that well-typed states can take a step. In particular, a machine state that is well-typed under the empty zap tag can take a non-faulty step to another ordinary, non-faulty machine state. A machine state that is well-typed under a zap tag of color c can take a step, but the result of that step may either be another ordinary machine state or the *fault* state.

Theorem 1 (Progress)

1. *If* $\vdash \Sigma$ *then* $\Sigma \longrightarrow_0^s \Sigma'$ *and* $\Sigma' \neq$ *fault.*
2. *If* $\vdash^c \Sigma$ *then* $\Sigma \longrightarrow_0^s \Sigma$.

Proof. *By case analysis on the instruction ir in state Σ.*

Part 1 uses inversion on the typing rules to determine that the preconditions hold for the appropriate operational rule. For example, Figure 12 shows the case for st_B*. By inverting various typing rules, we gather enough information to conclude that the last pair in the queue is equal to the contents of the two registers. Notice that step 9 inverts the* val-t *rule. This can only be done because we know that $Z = \cdot$, and so the* val-zap-t *rule cannot apply instead. (And similarly, step 12 inverts rule* Q-t*.) Once we have these equalities, we can apply the operational rule* st_B-mem*. Other cases are similar, although the cases for* ld_G *and* bz_B *subdivide further based on the result of the* find *function and the value in the branch register.*

Part 2 is simpler than part 1. Instead of using typing rules to gather as much information, we just further subdivided based on properties needed to apply the rules. These subcases take either the normal operation rule, or the corresponding failure rule as necessary. Some inversion may be done on the typing rules to show that the case does not get stuck. Again, Figure 12 shows the case for st_B*. From the typing information, we know that the Q is not empty. Then either the last pair in the queue is equal to the registers (rule* st_B *applies) or it is not (rule* st_B-fail *applies).*

According to Preservation, if a machine state is well-typed under a zap tag Z, and it takes a non-faulty step to another machine state, then that resulting state will also be well-typed under Z. Additionally, if a state is well-typed under the empty zap tag, and it takes a faulty step, then there is some color c such that the resulting state is well-typed under c.

Theorem 2 (Preservation)

1. *If* $\vdash^Z \Sigma$ and $\Sigma \longrightarrow_0^s \Sigma'$ *and* $\Sigma' \neq$ *fault then* $\vdash^Z \Sigma'$.
2. *If* $\vdash \Sigma$ and $\Sigma \longrightarrow_1^s \Sigma'$ *then* $\exists\ c.\ \vdash^c \Sigma'$.

Proof. *By case analysis of the structure of the derivation* $\Sigma \longrightarrow_k^s \Sigma'$.

Part 1 only applies to cases where Σ' is not $fault$. We use inversion to take apart the judgment for $\vdash^Z \Sigma$, modify it as necessary, and build the judgment $\vdash^Z \Sigma'$. Each case is subdivided based on the actual value of the zap tag. In subcases where the zap tag is not the same as the color of a value, we can invert rule val-t

Part 1 Example Case: $\mathbf{st}_B$

a1.	$\vdash (R, C, M, Q,\ st_B\ \ r_d, r_s)$	[assumption]
1.	$\Psi \vdash C$	[Inversion of Σ-t, a1]
2.	$\forall c.\ C(R_{val}(pc_c)) = st_B\ \ r_d, r_s$	[Inversion of Σ-t, a1]
3.	$\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}, (E'_d, E'_s); E_m) \vdash\ st_B\ r_d\ r_s$ $\Rightarrow (\Delta; \Gamma{+}{+}; \overline{(E_d, E_s)}; upd\ E_m\ E'_d\ E'_s)$	[Inversion of C-t, 1, 2, inspection of st_B-t]
4.	$\exists S.\ \cdot \vdash S : \Delta$	[Inversion of Σ-t, a1]
5.	$\Psi; (\cdot; S(\Gamma); S(\overline{(E_d, E_s)}, (E'_d, E'_s)); S(E_m)) \vdash\ st_B\ r_d\ r_s$ $\Rightarrow (\cdot; S(S(\Gamma){+}{+}; S(\overline{(E_d, E_s)}); upd\ S(E_m\ E'_d\ E'_s))$	[Lemma 4 (Substitution), 4, 3]
6.	$\Psi \vdash R : S(\Gamma)$	[Inversion of Σ-t, a1]
7.	$S(\Gamma)(r_d) = \langle B, b\ ref, E''_d \rangle$ $S(\Gamma)(r_s) = \langle B, b, E''_s \rangle$	[Inversion of st_B-t, 5]
8.	$\Psi; \cdot \vdash R(r_d) : \langle B, b\ ref, E''_d \rangle$ $\Psi; \cdot \vdash R(r_s) : \langle B, b, E''_s \rangle$	[Inversion of R-t, 6, 7]
9.	$\cdot \vdash Rval(r_d) = E''_d$ $\cdot \vdash Rval(r_s) = E''_s$	[Inversion of val-t, 8]
10.	$\cdot \vdash S(E'_d) = E''_d$ $\cdot \vdash S(E'_s) = E''_s$	[Inversion of st_B-t, 5]
11.	$\Psi \vdash Q : S(\overline{(E_d, E_s)}, (E'_d, E'_s))$	[Inversion of Σ-t, a1]
12.	$Q = (\overline{(n, n')}, (n_l, n'_l))$ $\cdot \vdash S(E'_d) = n_l$ and $\cdot \vdash S(E'_s) = n'_l$	[Lemma 12 (Queue), 11]
13.	$R_{val}(r_d) = n_l$ and $R_{val}(r_s) = n'_l$	[Lemma 3 (Exp Eq Transitivity), 9, 10, 12]
14.	$(R, C, M, (\overline{(n, n')}, (n_l, n'_l)), st_B\ \ r_d, r_s) \longrightarrow_0^{(n_l, n'_l)}$ $(R{+}{+}, C, M[n_l \mapsto n'_l], \overline{(n, n')}, \cdot)$	[st_B-mem, 14]

Case complete.

Part 2 Example Case: $\mathbf{st}_B$

a1.	$\vdash^c (R, C, M, Q,\ st_B\ \ r_d, r_s)$	[assumption]
1.	$\Psi \vdash^c Q : S(\overline{(E_d, E_s)}, (E'_d, E'_s))$	[Inversion of Σ-t, a1]
2.	$Q = (\overline{(n, n')}, (n_l, n'_l))$	[Lemma 12 (Queue), 1]

Case on whether $R_{val}(r_d) \stackrel{?}{=} n_l$ and $R_{val}(r_s) \stackrel{?}{=} n'_l$

subcase a: r_d and r_s are the same as the last pair in the queue

a4a.	$R_{val}(r_d) = n_l$ and $R_{val}(r_s) = n'_l$	[subcase assumption]
3a.	$(R, C, M, Q, st_B\ \ r_d, r_s) \longrightarrow_0^{(n_l, n'_l)} (R{+}{+}, C, M[n_l \mapsto n'_l], \overline{(n, n')}, \cdot)$	[st_B-mem, a4a]

Subcase complete.

subcase b: either r_s or r_d or the last pair in the queue has been corrupted

a4b.	$R_{val}(r_d) \neq n_l$ or $R_{val}(r_s) \neq n'_l$	[subcase assumption]
3b.	$(R, C, M, Q, st_B\ \ r_d, r_s) \longrightarrow_0 fault$	[st_B-mem-$fail$, a4b]

Subcase complete.

Case complete.

Figure 12: Example Cases of Theorem 1 (Progress).

on the typing of a value and use that information to construct a typing derivation for Σ'. In cases where the zap tag has the same color as the value, modified values in Σ' can be trivially typed using rules val-zap-t *and* Q-zap-t. *Figures 13 and 14 show an example case for rule* st_B-mem.

Part 2 chooses c to be the color of the zapped value. It uses rules val-zap-t, val-zap-cond *or* Q-zap-t *to show that the zapped value can be typed under zap tag c. The remainder of the state can be typed as before using the Color Weakening Lemmas 8, 9, and 14. Figure 15 shows an example case for rule* reg-zap.

The Progress and Preservation Theorems define the usual notion of type safety.

4.3 Multistep Transitions

In order to prove properties of our type system, we extend our single-step transition $\Sigma_1 \longrightarrow^s_k \Sigma_2$ from Section 2 to a sequence of n transitions containing exactly k faults $\Sigma_1 \xrightarrow{n}{}^{s}_{k} \Sigma_2$, where n is greater than or equal to zero, and k is still either 0 or 1.

$$\frac{}{\Sigma \xrightarrow{0}{}^{()}_{0} \Sigma}\ \textit{(multi-base)} \qquad \frac{\Sigma \longrightarrow^{s_1}_{k_1} \Sigma'' \quad \Sigma'' \xrightarrow{(n-1)}{}^{s_2}_{k_2} \Sigma'}{\Sigma \xrightarrow{n}{}^{(s_1,s_2)}_{k_1+k_2} \Sigma'}\ \textit{(multi-compose)}$$

4.3.1 No False Positives

By combining part one of Progress with part one of Preservation, we get the following important corollary: The hardware never claims to have detected a fault when no fault has occurred during execution of a well-typed program.

Corollary 3 (No False Positives)
If $\vdash \Sigma$ *then* $\forall\ n.\ \Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$ *and* $\vdash \Sigma'$.

Proof. *By Progress Part 1, Preservation Part 1, and induction on the derivation of* $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$.

4.3.2 Multistep Split and Combine

The following two lemmas about the multistep relation let us take apart and put together different sequences of steps.

If a machine state evaluates in a sequence of steps with no faults to a final state, then this computation can be divided into a sequence of non-faulty steps reaching an intermediate state, and a sequence of non-faulty steps from this intermediate state to the final state.

Lemma 15 (Multistep Split)
If $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$ *then there exists* n_1, n_2, Σ'', s_1, *and* s_2 *such that* $n = n_1 + n_2$ *and* $s = (s_1, s_2)$ *and* $\Sigma \xrightarrow{n_1}{}^{s_1}_{0} \Sigma''$ *and* $\Sigma'' \xrightarrow{n_2}{}^{s_2}_{0} \Sigma'$.

Proof. *By induction on the structure of* $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$.

If a machine state evaluates in a sequence of n_1 non-faulty steps to another state, that state faults to a third state, and the third state evaluates in n_2 non-faulty steps to a final state, then the original state can reach the faulty state in a sequence of $n_1 + 1 + n_2$ steps including exactly one fault step.

Example Case: st$_B$-mem

$$\frac{R_{val}(r_d) = n_l \qquad R_{val}(r_s) = n'_l}{(R, C, M, (\overline{(n, n')}, (n_l, n'_l)), st_B \;\; r_d, r_s) \longrightarrow_0^{(n_l, n'_l)} (R\text{++}, C, M[n_l \mapsto n'_l], \overline{(n, n')}, \cdot)} \; (st_B\text{-}mem)$$

a1.	$\vdash^Z (R, C, M, Q, \; st_B \;\; r_d, r_s)$	[assumption]
p1.	$R_{val}(r_d) = n_l$	[premise]
p2.	$R_{val}(r_s) = n'_l$	[premise]
1.	$Dom(\Psi) = Dom(C) \cup Dom(M)$	[Inversion of Σ-t, a1]
2.	$Z \neq G \implies Dom(\overline{(n, n')}, (n_l, n'_l)) \subseteq Dom(M)$	[Inversion of Σ-t, a1]
3.	$\Psi \vdash C$	[Inversion of Σ-t, a1]
4.	$\forall c \neq Z.\; C(R_{val}(pc_c)) = \; st_B \;\; r_d, r_s$	[Inversion of Σ-t, a1]
5.	$\forall c \neq Z.\; \Psi(R_{val}(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \rightarrow void$	[Inversion of Σ-t, a1]
6.	$\exists S.\; \cdot \vdash S : \Delta$	[Inversion of Σ-t, a1]
7.	$\Psi \vdash M : S(E_m)$	[Inversion of Σ-t, a1]
8.	$\Psi \vdash^Z (\overline{(n, n')}, (n_l, n'_l)) : S(\overline{(E_d, E_s)}, (E'_d, E'_s))$	[Inversion of Σ-t, a1]
9.	$\Psi \vdash^Z R : S(\Gamma)$	[Inversion of Σ-t, a1]
10.	$\Psi; (\Delta; \Gamma; \overline{(E_d, E_s)}, (E'_d, E'_s); E_m) \vdash \; st_B \; r_d \; r_s \Rightarrow \Theta'$	[Inversion of C-t, 3, 4]
11.	$\Theta' = (\Delta; \Gamma\text{++}; \overline{(E_d, E_s)}; upd \; E_m \; E'_d \; E'_s)$	[10, inspection of st_B-t]
12.	$\forall c \neq Z.\; C(R_{val}(pc_c) + 1) = \Theta' \mapsto void$	[Inversion of C-t, 3, 4, 10, 11]
5'.	$\forall c \neq Z.\; C(R\text{++}_{val}(pc_c)) = \Theta' \mapsto void$	[12, def of R++]
9'.	$\Psi \vdash^Z R\text{++} : S(\Gamma)\text{++}$	[9, def of R++ and Γ++]
13.	$\Psi; (\cdot; S(\Gamma); S(\overline{(E_d, E_s)}, (E'_d, E'_s)); S(E_m)) \vdash \; st_B \; r_d \; r_s$ $\Rightarrow (\cdot; S(S(\Gamma)\text{++}; S(\overline{(E_d, E_s)}); upd \; S(E_m \; E'_d \; E'_s))$	[Lemma 4 (Substitution), 10, 11, 6]
14.	$S(\Gamma)(r_d) = \langle B, b \; ref, E''_d \rangle$	[Inversion of st_B-t, 5]
15.	$S(\Gamma)(r_s) = \langle B, b, E''_s \rangle$	[Inversion of st_B-t, 5]
16.	$\cdot \vdash S(E'_d) = E''_d$	[Inversion of st_B-t, 5]
17.	$\cdot \vdash S(E'_s) = E''_s$	[Inversion of st_B-t, 5]

Case on Z

Subcase a1: $Z = G$

a2a.	$Z = G$	[subcase assumption]
18a.	$\Psi \vdash^Z \overline{(n, n')} : S(\overline{(E_d, E_s)})$	[Repeated Inversion of Q-zap-t, a2a, 8, Q-zap-t]

Subcase b1: $Z = \cdot$ or $Z = B$

a2b.	$Z = \cdot$ or $Z = B$	[subcase assumption]
18b.	$\Psi \vdash^Z \overline{(n, n')} : S(\overline{(E_d, E_s)})$	[Repeated Inversion of Q-t, a2a, 8, Q-t]

Merge subcases a1 and b1:

8'.	$\Psi \vdash^Z \overline{(n, n')} : S(\overline{(E_d, E_s)})$	[18a/18b]
2'.	$Z \neq G \implies Dom(\overline{(n, n')}) \subseteq Dom(M)$	[2]

Continued in Figure 14...

Figure 13: Example Case from Theorem 2 (Preservation) Part 1.

Example Case: st$_B$-mem (...continued from Figure 13)

Case on Z

Subcase a2: $Z = B$ (the queue is correct)

a3a.	$Z = B$	[subcase assumption]
19a.	$\cdot \vdash S(E'_d) = n_l$	[Inversion of *Q-t*, a3a, 8]
20a.	$\Psi \vdash n_l : b\ ref$	[Inversion of *Q-t*, a3a, 8]
21a.	$\Psi; \cdot \vdash^B n'_l : \langle B, b\ ref, E''_d \rangle$	[*val-t*, 19a, 20a]
22a.	$n_l \in Dom(M)$	[Lemma 6 (Canonical Forms), 7, 3, 21a
23a.	$\cdot \vdash S(E'_s) = n'_l$	[Inversion of *Q-t*, a3a, 8]
24a.	$\Psi \vdash n'_l : b$	[Inversion of *Q-t*, a3a, 8]
25a.	$[\![S(E'_d)]\!] = n_l$ and $[\![S(E'_s)]\!] = n'_l$	[Inversion of *E-eq*, 19a, 23a]

Subcase b2: $Z = \cdot$ or $Z = G$ (r_d and r_s are correct)

a3b.	$Z = \cdot$ or $Z = G$	[subcase assumption]
19b.	$\Psi; \cdot \vdash^Z R_{val}(r_d) : \langle B, b\ ref, E''_d \rangle$	[Inversion of *R-t*, 9, 14]
20b.	$\Psi; \cdot \vdash^Z n_l : \langle B, b\ ref, S(E'_d) \rangle$	[19b, p1, Lemma 3 (Exp Eq Transitivity), 16]
21b.	$n_l \in Dom(M)$	[Lemma 6 (Canonical Forms), 7, 3, 20b
22b.	$\Psi \vdash n_l : b\ ref$	[Inversion of *val-t*, a3b, 20b]
23b.	$\Psi; \cdot \vdash^Z R_{val}(r_s) : \langle B, b, E''_s \rangle$	[Inversion of *R-t*, 9, 15]
24b.	$\Psi; \cdot \vdash^Z n'_l : \langle B, b, S(E'_s) \rangle$	[23b, p2, Lemma 3 (Exp Eq Transitivity), 17]
25b.	$\Psi \vdash n'_l : b$	[Inversion of *val-t*, a3b, 24b]
26b.	$\cdot \vdash S(E'_d) = n_l$ and $\cdot \vdash S(E'_s) = n'_l$	[Inversion of *val-t*, a3b, 20b, 24b]
27b.	$[\![S(E'_d)]\!] = n_l$ and $[\![S(E'_s)]\!] = n'_l$	[Inversion of *E-eq*, 26b]

Merge subcases a2 and b2:

30.	$\forall \ell \in Dom(M).\ \Psi \vdash \ell : b\ ref \wedge \Psi \vdash M(\ell) : b$	[Inversion of *M-t*, 7]
31.	$\forall \ell \in Dom(M[n_l \mapsto n'_l]).\ \Psi \vdash \ell : b\ ref \wedge \Psi \vdash M(\ell) : b$	[30, 20a/22b, 24a/25b]
32.	$[\![S(E_m)]\!] = M$	[Inversion of *M-t*, 7]
33.	$[\![upd\ S(E_m)\ S(E'_d)\ S(E'_s)]\!] = [\![S(E_m)]\!][\ [\![S(E'_d)]\!] \mapsto [\![S(E'_s)]\!]\ ]$	[def of $[\![E]\!]$]
34.	$[\![upd\ S(E_m)\ S(E'_d)\ S(E'_s)]\!] = M[n_l \mapsto n'_l]$	[33, 32, 25a/27b]
35.	$\cdot \vdash S(E_m) : \kappa_{mem}$	[Inversion of *M-t*, 7]
36.	$\cdot \vdash S(E'_d) : \kappa_{int}$ and $\cdot \vdash S(E'_s) : \kappa_{int}$	[Lemma 5 (Value Kinding), Inversion of *R-t*, 9, 14, 15]
37.	$\cdot \vdash upd\ S(E_m)\ S(E'_d)\ S(E'_s) : \kappa_{mem}$	[*E-upd-t*, 35, 36]
7'.	$\Psi \vdash M[n_l \mapsto n'_l] : S(E_m\ E'_d\ E'_s)$	[*M-t*, 31, 34, 37]
1'.	$Dom(\Psi) = Dom(C) \cup Dom(M[n_l \mapsto n'_l])$	[1, 22a/21b]
38.	$\vdash^Z (R{+}{+}, C, M[n_l \mapsto n'_l], \overline{(n, n')}, \cdot)$	[Σ-*t*, 1', 2', 3, *ir* = ., 5', 6, 7', 8', 9'

Figure 14: Example Case from Theorem 2 (Preservation) Part 1.

Example Case: reg-zap

$$\frac{R(a) = c\ n}{(R, C, M, Q, ir) \longrightarrow_1 (R[a \mapsto c\ n'], C, M, Q, ir)}\ (\textit{reg-zap})$$

a1.	$\vdash (R, C, M, Q, ir)$	[assumption]
1.	$Dom(\Psi) = Dom(C) \cup Dom(M)$	[Inversion of Σ-t, a1]
2.	$Dom(Q) \subseteq Dom(M)$	[Inversion of Σ-t, a1]
3.	$\Psi \vdash C$	[Inversion of Σ-t, a1]
4.	$\forall c.\ ir \neq \cdot \implies C(R_{val}(pc_c)) = ir$	[Inversion of Σ-t, a1]
5.	$\forall c.\ \Psi(R_{val}(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \to void$	[Inversion of Σ-t, a1]
6.	$\exists S.\ \cdot \vdash S : \Delta$	[Inversion of Σ-t, a1]
7.	$\Psi \vdash M : S(E_m)$	[Inversion of Σ-t, a1]
8.	$\Psi \vdash Q : S\overline{(E_d, E_s)}$	[Inversion of Σ-t, a1]
9.	$\Psi \vdash R : S(\Gamma)$	[Inversion of Σ-t, a1]

Case on the shape of $S(\Gamma)(a)$

Subcase a: $S(\Gamma)(a)$ is a triple $\langle c, b, E \rangle$

a2a.	Let $\langle c, b, E \rangle = S(\Gamma)(a)$	[subcase assumption]
10a.	$\cdot \vdash E : \kappa_{int}$	[Lemma 5 (Value Kinding), Inversion of R-t, 9, a2a]
11a.	$\Psi; \cdot \vdash^c c\ n' : \langle c, b, E \rangle$	[*val-zap-t*, 10a]
12a.	$\Psi; \cdot \vdash^c R[a \mapsto c\ n'](a) : S(\Gamma)(a)$	[11a, a2a, def of R[ ]]

Subcase b: $S(\Gamma)(a)$ is a conditional type $E' = 0 \Rightarrow \langle c, b, E \rangle$

a2b.	Let $(E' = 0 \Rightarrow \langle c, b, E \rangle) = S(\Gamma)(a)$	[subcase assumption]
10b.	$\cdot \vdash E : \kappa_{int}$	[Lemma 5, Inversion of R-t, 9, a2b]
11b.	$\Psi; \cdot \vdash^c c\ n' : E' = 0 \Rightarrow \langle c, b, E \rangle$	[*val-zap-cond*, 10b]
12b.	$\Psi; \cdot \vdash^c R[a \mapsto c\ n'](a) : S(\Gamma)(a)$	[11b, a2b, def of R[ ]]

Merge subcases a and b:

9'.	$\Psi \vdash^c R[a \mapsto c\ n'] : S(\Gamma)$	[R-t, 9, Lemma 9 (Reg File Color Weakening), 12a/12b]
8'.	$\Psi \vdash^c Q : S\overline{(E_d, E_s)}$	[Lemma 14 (Queue Color Weakening), 8]
2'.	$Z \neq G \implies Dom(Q) \subseteq Dom(M)$	[2]
4'.	$\forall c \neq Z.\ ir \neq \cdot \implies C(R_{val}(pc_c)) = ir$	[4]
5'.	$\forall c \neq Z.\ \Psi(R_{val}(pc_c)) = (\Delta; \Gamma; \overline{(E_d, E_s)}; E_m) \to void$	[5]
13.	$\vdash^c (R[a \mapsto c\ n'], C, M, Q, ir)$	[Σ-t, 1, 2', 3, 4', 5', 6, 7, 8', 9']

Case complete.

Figure 15: Example Case of Theorem 2 (Preservation) Part 2.

$\boxed{v_1 \ sim^Z \ v_2}$

$$\frac{}{C\ n\ sim^Z\ C\ n}\ (\textit{sim-val}) \qquad \frac{}{C\ n\ sim^C\ C\ n'}\ (\textit{sim-val-zap})$$

$\boxed{R \ sim^Z \ R'}$

$$\frac{\forall a.\ R(a)\ sim^Z\ R'(a)}{R\ sim^Z\ R'}\ (\textit{sim-R})$$

$\boxed{Q \ sim^Z \ Q'}$

$$\frac{}{\cdot\ sim^Z\ \cdot}\ (\textit{sim-Q-empty})$$

$$\frac{G\ n_1\ sim^Z\ G\ n_1' \qquad G\ n_2\ sim^Z\ G\ n_2' \qquad Q\ sim^Z\ Q'}{((n_1, n_2), Q)\ \ sim^Z\ ((n_1', n_2'), Q')}\ (\textit{sim-Q})$$

$\boxed{\Sigma_1 \ sim^Z \ \Sigma_2}$

$$\frac{R\ sim^Z\ R' \qquad Q\ sim^Z\ Q'}{(R, C, M, Q, ir)\ sim^Z\ (R', C, M, Q', ir)}\ (\textit{sim-}\Sigma)$$

Figure 16: Similarity of Machine States.

Lemma 16 (Multistep Combine)
If $\Sigma \xrightarrow{n_1}{}_0^{s_1} \Sigma'$ *and* $\Sigma' \longrightarrow_1 \Sigma_f'$ *and* $\Sigma_f' \xrightarrow{n_2}{}_0^{s_2} \Sigma''$ *then* $\Sigma \xrightarrow{n'}{}_1^{(s_1,s_2)} \Sigma''$ *where* $n' = n_1 + 1 + n_2$.

Proof. *By induction on the structure of* $\Sigma \xrightarrow{n_1}{}_0^{s_1} \Sigma'$.

4.4 Fault Tolerance

A program is fault tolerant when all the faulty executions of that program *simulate* fault-free executions of the program. More precisely, the sequence of outputs from the faulty executions are required either to be identical to the fault-free execution or, in the case the hardware detects the fault, a prefix of the fault-free execution.

4.4.1 Simulation Relation

In order to reason about pairs of faulty and fault-free executions, we define similarity relations between values, register files, queues and machine states. Each of these relations is defined relative to the zap tag Z. Intuitively, if Z is empty, the related objects must be identical. If Z is a color c, the objects must be identical modulo values colored c. In the latter case, values colored c may be corrupted, and there is no hope they satisfy any particular relation. The formal definitions of these relations are shown in Figure 16.

4.4.2 Singlestep Fault Detection

We begin by defining fault detection for a single step in the execution of a program. It essentially says that if we have two similar computations, one with a fault and one without, then either the faulty computation can take a step indistinguishable from that of the the non-faulty version, or the faulty computation reaches the fault state.

Lemma 17 (Singlestep Fault Detection)
If $\vdash \Sigma$ *and* Σ sim^c Σ_f *and* $\Sigma \longrightarrow_0^s \Sigma'$ *then* $\Sigma_f \longrightarrow_0^{s_f} \Sigma'_f$ *and either*

1. Σ' sim^c Σ'_f *and* $s = s_f$*, or*

2. $\Sigma'_f = fault$ *and* $s_f = ()$*.*

Proof. *By case analysis of* $\Sigma \longrightarrow_0^s \Sigma'$*. Each case is handled in one of three ways.*

Failure Cases. *Rules where* Σ *steps to fault are not applicable because* Σ *is well typed under the empty zap tag, and so Progress 1 tells us that* Σ' *is not fault.*

Random Cases. *Rules* ld_B-rand *and* ld_G-rand *only apply when loading from an invalid address that is not in the domain of memory. According to Lemma 10 (Well-typed Domain), states that are well-typed under the empty zap tag only load from valid addresses, so these cases can be ruled out.*

Standard Cases. *The remaining rules are all handled in approximately the same way.*

Each rule has a handful of premises. These relationships may or may not hold in the faulty computation. It will depend on what exactly has been corrupted. The proof divides into subcases based on these relationships. Some subcases may further divide based on whether c is G *or* B*.*

Figure 17 shows the case for st_B-mem*. There are two premises relating the two registers to the last pair in the queue. If one of these premises does not hold in the faulty computation, then it will step to fault using the* st_B-fail *rule with no output.*

However, if both equalities hold then the proof further subdivides based on the color c. If c is B*, then we know that the faulty and non-faulty queues are equal because they contain green values and simulate each other under color* B*. From there, we can determine that the faulty computation also steps using* st_B*, the resulting state is similar to* Σ'*, and the outputs are equal.*

If c is G*, then we use the fact that* Σ *is well-typed to show that the registers* r_d *and* r_s *are colored blue, and so both the faulty and non-faulty registers are correct and equal to each other. And since the registers are assumed to be equal to the ends of the queues, we also know that the ends of the queues are equal. From there, we continue as in the previous subcase.*

4.4.3 Multistep Fault Detection

The Multistep Fault Detection Lemma extends the Singlestep Fault Detection Lemma for n steps. If a fault has occurred and the non-faulty computation takes n steps to a state Σ', then the faulty computation with either take n steps to a state that simulates Σ', or it will terminate in the fault state *fault* during this time.

Lemma 18 (Multistep Fault Detection)
If $\vdash \Sigma$ *and* Σ sim^c Σ_f *and* $\Sigma \xrightarrow{n}{}_0^s \Sigma'$ *then either*

1. $\Sigma_f \xrightarrow{n}{}_0^{s_f} \Sigma'_f$ and Σ' sim^c Σ'_f and $s_f = s$, or

2. *Exists* $m \leq n$*.* $\Sigma_f \xrightarrow{m}{}_0^{s_f} fault$ *and* s_f *is a prefix of* s*.*

Proof. *By induction on the structure of* $\Sigma \xrightarrow{n}{}_0^s \Sigma'$*. The base case for* multi-base *is immediate.*
In the case for multi-compose*, we know from the premises that* Σ *takes a single step to some* Σ''*. Using this and the Singlestep Fault Detection Lemma, we know that* Σ_f *either takes a single step with no output to fault or takes a single step to some state* Σ''_f *that simulates* Σ'' *while generating equal output.*
In the former case, we can immediately prove the second possibility with $m = 1$ *and* $()$ *as a prefix of* s*.*

Example Case: st$_B$-mem

$$\frac{R_{val}(r_d) = n_l \qquad R_{val}(r_s) = n_l'}{(R, C, M, (\overline{(n, n')}, (n_l, n_l')), st_B \;\; r_d, r_s) \longrightarrow_0^{(n_l, n_l')} (R\text{++}, C, M[n_l \mapsto n_l'], \overline{(n, n')}, \cdot)} \; (st_B\text{-}mem)$$

a1.	$\vdash (R, C, M, (\overline{(n, n')}, (n_l, n_l')),\; st_B \;\; r_d, r_s)$	[assumption]
a2.	$(R, C, M, (\overline{(n, n')}, (n_l, n_l')),\; st_B \;\; r_d, r_s) \;\; sim^c \; \Sigma_f$	[assumption]
a3.	$(R, C, M, (\overline{(n, n')}, (n_l, n_l')),\; st_B \;\; r_d, r_s) \longrightarrow_0^{(n_l, n_l')} (R\text{++}, C, M[n_l \mapsto n_l'], \overline{(n, n')}, \cdot)$	[assumption]
p1.	$R_{val}(r_d) = n_l$	[premise]
p2.	$R_{val}(r_s) = n_l'$	[premise]

1.	$\Sigma_f = (R_f, C, M, Q_f, ir)$	[a2, definition of *sim*-Σ]
2.	$R \; sim^c \; R_f$	[a2, 1, inversion of *sim*-Σ]
3.	$(\overline{(n, n')}, (n_l, n_l')) \; sim^c \; Q_f$	[a2, 1, inversion of *sim*-Σ]
4.	$Q_f = \overline{(n_f, n_f')}, (n_{lf}, n_{lf}')$	[3, definition of *sim*-Q]

Case on whether $R_{f\,val}(r_d) \stackrel{?}{=} n_{lf}$ and $R_{f\,val}(r_s) \stackrel{?}{=} n_{lf}'$ and the color c

subcase a: either r_s or r_d or the last pair in the queue has been corrupted

a4a.	$R_{f\,val}(r_d) \neq n_{lf}$ or $R_{f\,val}(r_s) \neq n_{lf}'$	[subcase assumption]
6a.	$(R_f, C, M, Q_f, st_B \;\; r_d, r_s) \longrightarrow_0 fault$	[st_B-*fail*, 4, a4a]
7a.	$\Sigma_f \longrightarrow_0^{s_f} fault$ and $s_f = ()$	[6a]

subcase complete.

subcase b: blue values are corrupted, but not r_s or r_d

a4b.	$R_{f\,val}(r_d) = n_{lf}$ and $R_{f\,val}(r_s) = n_{lf}'$	[subcase assumption]
a5b.	$c = B$	[subcase assumption]
6b.	$G \; n_l \;\; sim^B \;\; G \; n_{lf}$	[3, inversion of *sim*-Q]
7b.	$n_l = n_{lf}$	[6b, inversion of *sim-val*, a5b]
8b.	$G \; n_l' \;\; sim^B \;\; G \; n_{lf}'$	[3, inversion of *sim*-Q]
9b.	$n_l' = n_{lf}'$	[8b, inversion of *sim-val*, a5b]
10b.	$M[n_l \mapsto n_l'] = M[n_{lf} \mapsto n_{lf}']$	[7b, 9b, a5b]
11b.	$\overline{R\text{++} \; sim^B \; R_f\text{++}}$	[2, definition of R++, *sim-val*, *sim-R*, a5b]
12b.	$\overline{(n, n')} \; sim^B \; \overline{(n_f, n_f')}$	[3, 4, inversion of *sim-Q*, a5b]
13b.	$(R\text{++}, C, M[n_l \mapsto n_l'], \overline{(n, n')}, \cdot) \; sim^B \; (R_f\text{++}, C, M[n_{lf} \mapsto n_{lf}'], \overline{(n_f, n_f')}, \cdot)$	[10b, 11b, 12b]
14b.	$\Sigma_f \longrightarrow_0^{(n_{lf}, n_{lf}')} (R_f\text{++}, C, M[n_{lf} \mapsto n_{lf}'], \overline{(n_f, n_f')}, \cdot)$	[st_B-*mem*, a4b, 1, 4]
15b.	$(n_{lf}, n_{lf}') = (n_l, n_l')$	[7b, 9b]
16b.	$\Sigma_f \longrightarrow_0^{s_f} \Sigma_f'$ and $\Sigma' \; sim^c \; \Sigma_f'$ and $s_f = s$	[14b, 13b, 15b]

subcase complete.

subcase c: green values are corrupted, but not the last pair in the queue

a4c.	$R_{f\,val}(r_d) = n_{lf}$ and $R_{f\,val}(r_s) = n_{lf}'$	[subcase assumption]
a5c.	$c = G$	[subcase assumption]
6c.	$R_{col}(r_d) = R_{col}(r_s) = B$	[a1, inversion of Σ-*t*, inversion of st_B-*t*]
7c.	$R(r_d) \; sim^G \; R_f(r_d)$	[2, inversion of *sim-R*, a5c]
8c.	$R(r_s) \; sim^G \; R_f(r_s)$	[2, inversion of *sim-R*, a5c]
9c.	$R_{val}(r_d) = R_{f\,val}(r_d)$	[7c, 6c, inversion of *sim-val*]
10c.	$R_{val}(r_s) = R_{f\,val}(r_s)$	[8c, 6c, inversion of *sim-val*]
11c.	$n_l = n_{lf}$	[9c, a4c, p1]
12c.	$n_l' = n_{lf}'$	[10c, a4c, p2]
13c.	$M[n_l \mapsto n_l'] = M[n_{lf} \mapsto n_{lf}']$	[11c, 12c]
14c.	$\overline{R\text{++} \; sim^G \; R_f\text{++}}$	[2, definition of R++, *sim-val*, *sim-R*, a5c]
15c.	$\overline{(n, n')} \; sim^G \; \overline{(n_f, n_f')}$	[3, 4, inversion of *sim-Q*, a5c]
16c.	$(R\text{++}, C, M[n_l \mapsto n_l'], \overline{(n, n')}, \cdot) \; sim^G \; (R_f\text{++}, C, M[n_{lf} \mapsto n_{lf}'], \overline{(n_f, n_f')}, \cdot)$	[13c, 14c, 15c]
17c.	$\Sigma_f \longrightarrow_0^{(n_{lf}, n_{lf}')} (R_f\text{++}, C, M[n_{lf} \mapsto n_{lf}'], \overline{(n_f, n_f')}, \cdot)$	[st_B-*mem*, a4c]
18c.	$(n_{lf}, n_{lf}') = (n_l, n_l')$	[11c, 12c]
19c.	$\Sigma_f \longrightarrow_0^{s_f} \Sigma_f'$ and $\Sigma' \; sim^c \; \Sigma_f'$ and $s_f = s$	[17c, 16c, 18c]

subcase complete.

case complete.

Figure 17: Example Case of Lemma 17.

In the latter case, we call the Induction Hypothesis, which tells us that either Σ''_f takes $n-1$ steps to a state that simulates Σ' generating equal output, or it reaches fault in no more than $n-1$ steps with a prefix of the output. In the first case, where fault is never reached, we use multi-compose *to conclude that Σ_f takes n steps to Σ'_f and the total output is equal. In the second case, where a fault is reached later in execution, we use* multi-compose *to show that Σ_f reaches fault in no more than n steps, and its output is a prefix of the non-faulty output.*

4.4.4 Fault Similarity

The Fault Similarity Lemma states that if a non-faulty machine state takes a single faulty step, then the resulting machine state is similar to the original state under some color c.

Lemma 19 (Fault Similarity)
If $\Sigma \longrightarrow_1 \Sigma_f$, then $\exists\ c.\ \Sigma\ sim^c\ \Sigma'$.

Proof. *By case analysis on the definition of $\Sigma \longrightarrow_1 \Sigma'$. In each case, c is assigned the color of the value that is zapped. The zapped value is similar to the original value by* sim-val-zap*. The remainder of the state is equal, and is similar using* sim-val.

4.4.5 Fault Tolerance Theorem

By using the three previous lemmas, we can state and prove the fault tolerance theorem for well-typed programs. Assume that machine state Σ is well-typed under the empty zap tag, and non-faulty execution of Σ for n steps results in a state Σ' and outputs a sequence of value-address pairs s. If somewhere during that execution a single fault is encountered, the faulty execution will either run for $n+1$ steps or terminate in the fault state during that time. If the faulty execution takes $n+1$ steps and reaches the non-faulty state Σ'_f, then Σ' simulates Σ'_f and the sequence of output pairs is identical the original execution. Alternatively, if the faulty execution reaches the fault state then the output pairs will be a prefix of the non-faulty output pairs.

Theorem 4 (Fault Tolerance)
If $\vdash \Sigma$ and $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$ then either $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ or $\exists m \leq (n{+}1)\ .\ \Sigma \xrightarrow{m}{}^{s'}_{1} fault$, and

1. *For all derivations $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ where $\Sigma'_f \neq fault$. $s' = s$ and $\exists\ c.\ \Sigma'\ sim^c\ \Sigma'_f$.*
2. *For all derivations $\Sigma \xrightarrow{m}{}^{s'}_{1} fault$ where $m \leq (n{+}1)$. s' is a prefix of s.*

Proof. *By case analysis on the definition of $\Sigma \xrightarrow{n}{}^{s}_{0} \Sigma'$.*

In the base case for multi-base*, we can easily prove result 1 by having Σ take a faulty step to Σ_f and using the Fault Similarity Lemma to show that $\exists\ c.\ \Sigma\ sim^c\ \Sigma_f$.*

In the recursive case multi-compose*, we use the Multistep Split Lemma to divide the computation into two pieces with an intermediate state Σ''. Σ'' can take a faulty step to Σ_f, and we can again use the Fault Similarity Lemma to show that they are similar. We then call the Multistep Fault Detection Lemma with the execution from Σ'' to Σ' and the similarity between Σ'' and Σ_f. This tells us that Σ_f can either step to a state Σ'_f that is similar to Σ' or reaches a fault before that point. Finally, we use the Multistep Combine Lemma to combine the first part of the non faulty execution, the fault step, and the resulting execution from the Multistep Fault Detection Lemma, to show that either $\Sigma \xrightarrow{(n+1)}{}^{s'}_{1} \Sigma'_f$ or $\Sigma \xrightarrow{m}{}^{s'}_{1} fault$. The length of the faulty computation is at most $n+1$ because of the addition of the single fault step. Since the Multistep Split*

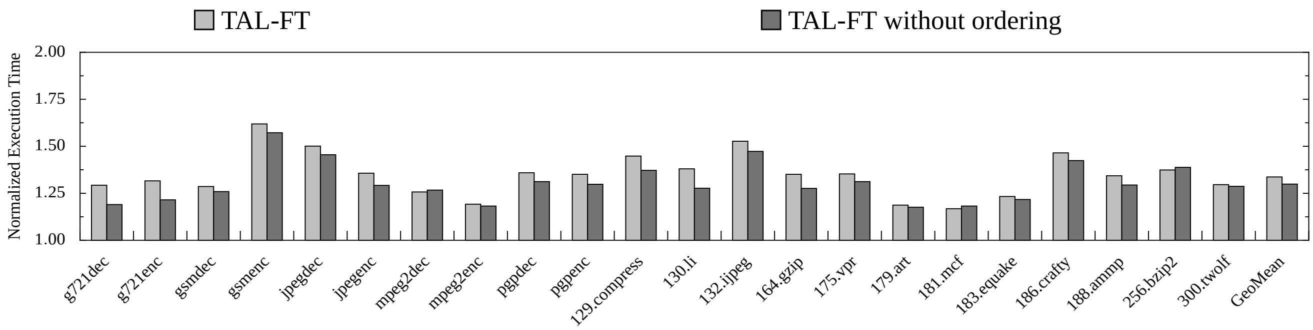

Figure 18: Performance Normalized to Unprotected Version.

Lemma chooses some unspecified division of the original computation, the result hold regardless of exactly where the fault is injected.

5 Performance

To better understand how $\mathrm{TAL_{FT}}$ can be applied to real world situations, we simulated the $\mathrm{TAL_{FT}}$ hardware in the framework of a current computer architecture, the Intel Itanium 2 ISA. The instruction set of the Itanium 2 contains many more types of instructions than those specified in $\mathrm{TAL_{FT}}$. While not an exact representation of the performance of $\mathrm{TAL_{FT}}$, simulating the performance of $\mathrm{TAL_{FT}}$ applied to this architecture will give guidance as to the feasibility of this system in a real architecture.

To evaluate the performance impact of our techniques, a version of the VELOCITY compiler [22] was modified to add the reliability techniques of $\mathrm{TAL_{FT}}$ and was used to compile the SPEC CINT2000 and MediaBench benchmark suites. These executions were compared against binaries generated by the original VELOCITY compiler, which have no fault detection. The reliability transformation was compiled into the low level code immediately before register allocation and scheduling. To simulate the new hardware structures of $\mathrm{TAL_{FT}}$, extra instructions were inserted to emulate the timing and dependences of the hardware structure accesses.

Performance metrics were obtained by running the resulting binaries with reference inputs on an HP workstation zx6000 with 2 900Mhz Intel Itanium 2 processors running Redhat Advanced Workstation 2.1 with 4Gb of memory. The `perfmon` utility was used to measure the CPU time.

Figure 18 presents the execution time of the fault-tolerant code relative to baseline binaries with no fault detection. Naïvely, one might expect the fault-tolerant code to run twice as slowly as the fault intolerant code since the number of instructions is essentially doubled. However, we find that smart instruction scheduling and efficient allocation of resources reduces the execution time to only 34% more than the fault-intolerant baseline average. These simulations are in line with previously published software-only reliability performance experiments [17] that show the degradation due to redundant code to be less than double.

As alluded to in Section 2.2, Figure 18 compares the performance degradation both with and without the scheduling constraint that green memory and control flow instructions must be executed before the corresponding blue versions. In order to perform the second set of experiments, our compiler was modified to produce code that had more flexibility in the scheduling of the green and blue versions. We then simulated a more aggressive hardware implementation that could correlate the original and redundant memory operations regardless of the executed order. As expected, this version has better performance (in most cases) than the unconstrained code. Comparing both to the unprotected code, the version without the ordering constraint increases execution time by 30% while the version with the ordering increases execution time by 34%.

Although the colored ordering restriction of TAL_{FT} may seem costly, removing this restriction provides only a small improvement.

6 Related Work

Fault tolerance based on software replication is a well-populated field with decades of history. TAL_{FT} differs from previous approaches in that it provides a type-theoretic framework for obtaining strong guarantees about the reliability of machine code.

Most closely related to TAL_{FT} is our previous work on λ_{zap}, a highly abstract type-theoretic model for studying the basic principles of fault tolerance in the lambda calculus [25]. There are two important distinctions between TAL_{FT} and λ_{zap}. First, λ_{zap}, working at the level of the lambda calculus, is very far removed from real machine code. For instance, it lacks a program counter, a register file, memory, and load or store instructions. Memory references in particular constitute a key challenge in the current technical work. Second, the properties of the λ_{zap} type system are relatively weak compared with the properties of the current type system. The "end-to-end" fault tolerance property proven for λ_{zap} depends not only on the type system but also the nature of the translation from the ordinary simply-typed lambda calculus. In contrast, the type system of TAL_{FT} is much stronger, capable of ensuring a strong fault tolerance property independently of the process that compiles the code.

Also closely related to TAL_{FT} is the original TAL system, which first applied strong type checking to machine code to guarantee its safety [8]. TAL operates under the assumption of nonfaulty hardware and therefore ignores the major issues of reliability on which this paper has focused.

There have been various implementations of software-only, hardware-only, and hybrid techniques for transient fault mitigation. Hardware techniques have a long history of using very localized bit-level techniques like error correcting codes or parity bits additions. These techniques are efficient for storage structures like memory, but are costly or impossible to apply to other processor elements like pipeline latches or arithmetic units. Higher level techniques are used when protection is necessary for larger segments of the processor. These techniques include the duplication of coarse-grained structures such as functional units, processor cores [5, 21, 26], or hardware contexts [9, 15, 24].

To provide protection when the hardware costs of these approaches are prohibitive, software-only approaches have been proposed as alternatives [12, 13, 16, 17, 19, 23]. While software-only systems are cheaper to deploy and can be configured after deployment, they cannot achieve the same performance or reliability as hardware-based techniques, since they have to execute additional instructions and are unable to examine microarchitectural state. Despite these limitations, software-only techniques have shown promise, in the sense that they can significantly improve reliability with reasonable performance overhead [12, 13, 17].

TAL_{FT} attempts to exploit the benefits of both sorts of systems by using a hybrid approach to fault tolerance. There have been previous hybrid approaches to transient fault tolerance, some focusing solely on control-flow protection [14] and recently others looking at full processor protection [18]. This work differs from those previous approaches because regardless of the type of implementation, software, hardware, or hybrid, none of those previous approaches have given rigorous formal proofs of the correctness of their systems.

7 Conclusions

In conclusion, transient faults are already a significant cause for concern at major semiconductor manufacturers and threaten to be more so in the coming years and decades. This paper takes one step forward for the science of fault tolerance by presenting a principled and practical hybrid software-hardware scheme for detecting transient faults. More specifically, we identify four general principles for verifying correctness of fault tolerant systems and capture these in an assembly language type system. From a theoretical perspective, the type system acts as a sound proof technique for verifying reliability properties of programs. From a practical perspective, it can be used as a debugging aid within a compiler, strictly dominating any conventional testing technique. Our two main formal results show that a single fault affecting observable behavior in a well-type program will always be detected, and that the system will not claim to have detected a fault when none has occurred. Despite the fact that well-typed programs essentially duplicate all computation, we provide simulation results showing a performance overhead of 1.34x.

Acknowledgements This research is funded in part by NSF awards CNS-0627650, CNS-0615250, and CCF 0633268. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.

[2] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1 – 121_01.14, April 2002.

[3] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. In *IEEE Micro*, volume 25, pages 10–16, December 2005.

[4] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109. ACM Press, 2003.

[5] R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 216–226, May 1990.

[6] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[7] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos National Labratory's ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.

[8] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[9] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110. IEEE Computer Society, 2002.

[10] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[11] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41–49, January 1996.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. In *IEEE Transactions on Reliability*, volume 51, pages 111–122, March 2002.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

[14] J. Ohlsson and M. Rimen. Implicit signature checking. In *International Conference on Fault-Tolerant Computing*, June 1995.

[15] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36. ACM Press, 2000.

[16] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery methods. In *IEEE Micro Top Picks*, volume 27, January 2007.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[19] P. P. Shirvani, N. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. In *IEEE Transactions on Reliability*, volume 49, pages 273–284, 2000.

[20] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–399, June 2002.

[21] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[22] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.

[23] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE International On-Line Testing Symposium*, pages 137–143, July 2003.

[24] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98. IEEE Computer Society, 2002.

[25] D. Walker, L. Mackey, J. Ligatti, G. Reis, and D. I. August. Static typing for a faulty lambda calculus. In *ACM International Conference on Functional Programming*, Portland, Oregon, Sept. 2006.

[26] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.

[27] J. F. Ziegler and H. Puchner. *SER - History, Trends, and Challenges: A Guide for Designing with Memory ICs*. 2004.