# Static Typing for a Faulty Lambda Calculus

David Walker Lester Mackey Jay Ligatti George A. Reis David I. August

Department of Computer Science
Princeton University
{dpw,lmackey,jligatti,gareis,august}@princeton.edu

## Abstract

A *transient hardware fault* occurs when an energetic particle strikes a transistor, causing it to change state. These faults do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values. While the likelihood that such transient faults will cause any significant damage may seem remote, over the last several years transient faults have caused costly failures in high-end machines at America Online, eBay, and the Los Alamos Neutron Science Center, among others [6, 44, 15]. Because susceptibility to transient faults is proportional to the size and density of transistors, the problem of transient faults will become increasingly important in the coming decades.

This paper defines the first formal, type-theoretic framework for studying reliable computation in the presence of transient faults. More specifically, it defines $\lambda_{\mathrm{zap}}$, a lambda calculus that exhibits intermittent data faults. In order to detect and recover from these faults, $\lambda_{\mathrm{zap}}$ programs replicate intermediate computations and use majority voting, thereby modeling software-based fault tolerance techniques studied extensively, but informally [10, 20, 30, 31, 32, 33, 41].

To ensure that programs maintain the proper invariants and use $\lambda_{\mathrm{zap}}$ primitives correctly, the paper defines a type system for the language. This type system guarantees that well-typed programs can tolerate any single data fault. To demonstrate that $\lambda_{\mathrm{zap}}$ can serve as an idealized typed intermediate language, we define a type-preserving translation from a standard simply-typed lambda calculus into $\lambda_{\mathrm{zap}}$.

***Categories and Subject Descriptors*** D.3.1 [*Programming languages*]: Formal Definitions and Theory—Semantics; B.8.1 [*Hardware*]: Reliability, Testing, and Fault-Tolerance

***General Terms*** Languages, Reliability, Theory, Verification

***Keywords*** Transient hardware faults, soft faults, type systems, typed intermediate languages, lambda calculus, fault tolerance, reliable computing

## 1. Transient Faults and Trustworthy Computing

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors enabled by improved fabrication technology. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [5, 23, 37]. Processors that use these transistors are more susceptible to *transient faults* (also known as *soft faults*), which result from external events, such as energetic particles striking the chip. These faults do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values. As each processor generation increases the density of transistors, the effects of transient faults will become more pronounced. To mitigate the deleterious effects of processor strikes, processor designers are devoting more of their attention to the growing reliability problem.

While discussions of alpha particles, neutrons, and cosmic rays interfering with earthly transistors may sound like science fiction to those unfamiliar with state-of-the-art processor design, it absolutely is not; transient faults are already causing substantial failures with significant costs in high-end machines. Consider, for instance, the following well-documented failures:

- In 2000, Sun Microsystems acknowledged that cosmic rays interfered with cache memories and caused crashes in server systems at major customer sites, including America Online, eBay, and dozens of others [6].
- Cypress Semiconductor acknowledged, "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." [44]
- Cypress Semiconductor also states: "Another incident occurred at an automotive supplier, where their *billion-dollar factory* ground to a halt *every month* due to what was traced to a single-bit flip in their network" [44]. (Emphasis added was our own.)
- At the Los Alamos Neutron Science Center, Hewlett Packard acknowledged their AlphaServer ES45 supercomputer was frequently crashing due to transient faults [15].

Hence, reliability in the presence of transient faults is already a significant cause for concern. Moreover, in the next 10 to 20 years, a desire to keep Moore's law on track will continue to provide huge incentives to reduce transistor sizes even further, substantially increasing the threat of transient faults.

***The case for software-implemented fault tolerance.*** Processor designers must constantly make trade-offs to obtain the best performance while still meeting their constraints. With the increasing importance of transient fault tolerance, reliability will emerge as another critical axis that can be traded off against performance, power, and cost. However, reliability, like security, can be a more difficult sell to the general public. The number of GHz your newest processor has, the lifetime of your laptop battery, and the cost of your computing solution all attract more attention. This is particularly true since hardware manufacturers are generally loath to pub-



*ICFP'06* September 16–21, 2006, Portland, Oregon, USA.

lish the soft error rates of their chips – such numbers can only generate negative publicity and can potentially even lead to lawsuits. So with all the focus on power, cost, and performance, reliability may be the axis to suffer.

One might speculate that the hardware industry will only deploy hardware fault tolerance techniques in its chips when it has actually suffered severe monetary losses. For many consumers, this may be too late. Indeed, the anecdotal evidence above suggests it is already too late for some consumers. Moreover, while many hardware techniques are available for dealing with transient faults, including using double or triple hardware redundancy [4, 13, 14, 38, 42, 43], these heavyweight techniques are extremely costly. Hence, although hardware fault tolerance techniques can be crucial for mission-critical applications, they are currently infeasible for commodity systems. As a result, even processor manufacturers are considering techniques with at least some software component.

Software-only fault tolerance techniques possess a massive potential advantage over hardware-only techniques in that they may be deployed *selectively* and *immediately* on existing hardware to whomever needs it, whenever they need it. At the first sign of trouble from transient faults, one could deploy new fault-resilient software to correct the problem. One certainly would not have to wait for a new generation of microprocessors while the current generation is failing in the field, costing millions or more to affected industries. Most importantly, companies and services with high reliability requirements could make the decisions themselves to deploy software that covers for potential hardware errors. The fast reaction time that is possible only in software could avert potential disasters for these companies. Surely, America Online, EBay, the affected telephone and automotive suppliers, and the Los Alamos super computer users mentioned above would have welcomed immediate software technology to avert further losses.

Consequently, over the last several years there has been substantial interest in developing new software fault detection methods for protecting memory [36], control flow [25, 28, 40], and general computation [7, 12, 24, 26, 27, 29]. At a high level, these systems effectively perform every subcomputation twice or three times and then compare the results of redundant executions to detect faults. From a performance standpoint, there is certainly a cost to this repeated computation, but it is not as high as one might think. For instance, the code redundancy in the SWIFT-R technique, a software-only system for fault tolerance that most closely matches the model used in this paper, comes at a reasonable cost. Intelligent instruction scheduling and other optimizations bring the performance cost down from the naïve 3x normal execution to a geometric mean overhead of 36%, over a set of recognized benchmarks [8].

***A trustworthy, end-to-end solution.*** Replacing hardware fault tolerance with software techniques that purport to tolerate and recover, but fail or introduce new errors, is unacceptable for customers who require highly reliable and trustworthy systems. Unfortunately, industrial-strength compilers are typically hundreds of thousands of lines of code and, naturally, contain bugs. Traditional compiler writers attempt to track down their bugs by building massive test suites. However, no matter how large the test suite, it cannot cover all combinations of programming language features. Bugs inevitably slip through the testing net and manifest themselves in the field. In the "good" case, relatively speaking, program developers catch compiler errors as they develop and test their programs. Developers then rewrite their programs, often awkwardly, to avoid actually deploying buggy applications. At the same time, they can communicate the bugs they found back to compiler writers who will then fix these problems in the next version of the compiler software. Hence, in a real sense, application developers work as a second line of testers for compiler writers. Of course, in the bad case, compiler errors slip through the last line of testing defense by the application developers and result in buggy applications.

For compiler writers attempting to generate fault-tolerant code, the compiler reliability problems are many orders of magnitude more difficult to overcome, yet the most concerned target customers demand greater end-to-end reliability than anyone else. The central difficulty is that transient hardware faults are an infrequently occurring and completely nondeterministic phenomenon. Faults may occur at any point in a computation: during execution of any instruction or in the midst of a control-flow transfer. Faults may also affect many different elements of observable hardware state: architectural registers, the program counter, condition flags, caches, and memory. If developing tests to cover compilation of all possible combinations of features ranges from difficult to near-impossible, then developing tests to cover all features in addition to all different kinds of faults at all different times in execution should be beyond consideration. If the situation could be any worse, one should also recognize that compiler writers cannot use application developers as a second line of testers. Transient faults that occur in the field show up infrequently and are generally unreproducible. Their only effect is to cause great damage, and when they do, it is too late to wish one had added that extra test case.

The clear conclusion is that a trustworthy platform for computing in the presence of transient faults cannot be built on the basis of traditional testing techniques alone. However, while conventional testing falls short, typed intermediate languages and type-preserving compilation [3, 16, 19, 21, 22, 35, 39] have the potential to provide substantially better reliability and to guarantee that compiled code is indeed fault tolerant. Unlike conventional compilers, a type-preserving compiler propagates typing information into its compiler intermediate languages. After each code transformation or optimization, the compiler can run an intermediate language type checker on the resulting code. If the type checker detects an error, then the compiler has produced incorrect code. Type-preserving compilation is extremely helpful for compiler writers attempting to debug their compiler. In addition, application developers can avoid shipping incorrect and unreliable code by type checking their compiled products. While they may be annoyed at type errors indicating that the compiler they are using has a bug, finding out earlier is substantially better than suffering expensive consequences later.

Type checking intermediate language programs is an important complement to conventional testing. While conventional testing only catches errors that show up on a particular run of a compiled program, a type checker can verify that certain properties hold for all runs of the program, no matter the inputs. So far, type preserving compilers have only been used to verify standard sorts of "type safety" and "memory safety" properties and, crucially, do so under the assumption of perfect hardware. However, we believe that the role of type checkers for intermediate languages can be dramatically expanded to take on the new task of verifying reliability properties. When verifying reliability, the type checker will not assume perfect hardware. Rather, the type checker will guarantee that under a certain hardware fault model, *for all runs* of the program and *for all possible faults* in the model, the program will not fail.

The guarantee over all runs and all faults is a guarantee no test suite can ever provide, but a type checker can. In fact, we believe that compiling for reliability in the presence of transient faults is a true "killer app" for type-directed compilation.

***Contributions: sound static typing for a faulty lambda calculus.*** This paper describes the first step in our program to develop trustworthy compilers for reliable computing in the presence of transient faults. More specifically, it defines the syntax and semantics of a faulty lambda calculus, $\lambda_{\mathrm{zap}}$. Operationally, the most interesting aspect of $\lambda_{\mathrm{zap}}$ is that at any point in execution, the (abstract)

machine can suddenly exhibit a fault. These faults are modeled by picking an arbitrary program value and spontaneously changing it to any other value, even an ill-typed one. To detect and recover from such faults, $\lambda_{\text{zap}}$ programs replicate computations and use $\lambda_{\text{zap}}$ primitives to compare the values computed from multiple replicas. If a single fault has occurred, a majority vote will detect that fault and recover before any problem becomes observable to the outside world. Overall, $\lambda_{\text{zap}}$'s operational semantics provides a high-level, formal model for the programs produced by the SWIFT-R system [8] and the data faults from which that system can recover.

In order for $\lambda_{\text{zap}}$ programs to be fault-tolerant, replicated computations must not depend upon one another. If they do then a single fault in one replica may propagate to other replicas. In such a case, comparing replicas will not necessarily detect faults, and the faulty computation might continue, possibly crashing or producing erroneous outputs. More interesting still, while implementing SWIFT-R, we noticed that some completely standard optimizations, including common subexpression elimination, are actually *unsound* in this context — they do not preserve fault-tolerance properties. Hence, in order to guarantee that programs are indeed fault-tolerant, we have defined a novel type system for $\lambda_{\text{zap}}$. The type system enforces the basic invariant necessary for fault tolerance — replicated computations cannot depend upon one another. The type system's strength is its simplicity and elegance. We believe it should easily scale to realistic typed intermediate languages. In examples, we show that the incorrect application of common subexpression elimination does indeed lead to type-incorrect code, and we conclude that the type system will be effective in helping compiler writers debug their code.

Our formal analysis of the language begins with proofs of the standard progress and preservation lemmas and continues with formulation and proof of a fault tolerance property for well-typed programs. Finally, we prove that it is possible to produce well-typed $\lambda_{\text{zap}}$ programs automatically by starting with terms in the ordinary simply typed lambda calculus and defining a correct, type-preserving translation to $\lambda_{\text{zap}}$.

## 2. Introduction to a Faulty Lambda Calculus

The faulty lambda calculus $\lambda_{\text{zap}}$ is an idealized intermediate language for compilers that generate fault-tolerant code. As mentioned in the introduction, the central mechanism these compilers use to detect and recover from faults is replication of computations. In this paper, we adopt the *Single Event Upset* model, which is a standard in the literature on hardware fault tolerance and which assumes that only a single fault will happen at a time.[1] In order to *detect* a single fault it is necessary to perform the same computation twice. Before the results of the two computations become observable to the outside world, the program compares them for equivalence. If the results are different then a fault has occurred. In general, in order to both detect and *recover* from a single fault, one performs three equivalent computations. A majority vote amongst the three results suffices for detecting and recovering from a fault.[2]

A second important element of our fault model is that we do not consider memory faults. When designing the SWIFT system [8], we observed that hardware (with error-correcting codes) protects against faults in cache and main memory much more efficiently than software. Hence, our software replication is intended to protect data in the processor pipeline, rather than data in cache and main memory. We rely on auxiliary hardware to tolerate faults in the memory hierarchy. In our abstract, high-level lambda calculus, we will model this by considering faults in integers and pointers to closures, but not in the closure data or code itself.

***A first example.*** Let us assume a source-level programmer writes a program to perform the following simple arithmetic computation and then print out the result:

```
let x = 2 in
let y = x + x in
out y
```

A compiler for single fault detection and recovery might produce the following code:

```
let x1 = 2 in
let x2 = 2 in
let x3 = 2 in
let y1 = x1 + x1 in
let y2 = x2 + x2 in
let y3 = x3 + x3 in
out [y1,y2,y3]
```

Each of the arithmetic computations is replicated three times. However, to simulate the observable behavior of the source program, there is only a single output command. This output command atomically compares `y1`, `y2`, and `y3` and prints out the majority value.[3] If we assume that addition is a total function mapping any two values to a third, this code is guaranteed to tolerate a single data fault. In other words, at any point during execution any one of the values may be corrupted and changed to any other value; nonetheless, the program is guaranteed to print out `4`. Moreover, any errors and incorrect intermediate results are unobservable to the outside world.

***Triples and color tags.*** We call the argument to `out`, `[y1,y2,y3]`, a triple, and it deserves some comment as its structure is subtly different from the structure of conventional *tuple* data structures one finds in standard functional programming languages. The key difference is that the components of a triple live in separate registers whereas a conventional tuple is implemented as a pointer into memory. While a single data fault can corrupt a tuple pointer, making all its components inaccessible, a single data fault can only corrupt one of the elements of a triple. The most general introduction form for a triple is $[e1, e2, e3]$. Its elimination form is `let` $[x1, x2, x3] = e1$ `in` $e2$, where $e1$ should be a triple and $x1$, $x2$, and $x3$ are bound to the components of that triple in $e2$. Unlike tuples, triples may not be nested. Triple evaluation proceeds from left to right. Hence, the example above could be rewritten as follows:

```
let [x1,x2,x3] = [2, 2, 2] in
let [y1,y2,y3] = [x1 + x1, x2 + x2, x3 + x3] in
out [y1,y2,y3]
```

This example code is still not quite syntactically correct $\lambda_{\text{zap}}$ code. For the purposes of typing, which will be discussed in the coming sections, each value is tagged with a color ($C$), which may be either red ($R$), green ($G$) or blue ($B$). After color tagging, our running example looks like this:

```
let [x1,x2,x3] = [R 2, G 2, B 2] in
let [y1,y2,y3] = [x1 + x1, x2 + x2, x3 + x3] in
out [y1,y2,y3]
```

[1] It would be trivial to change this assumption. Slight variants of $\lambda_{\text{zap}}$ involving $2k + 1$ replicas could be used tolerate any fixed $k$ number of faults.

[2] Though we have designed our calculus for both detection and recovery, changing this assumption and focusing on detection only would also likely be straightforward, though the forms of various theorems would change slightly.

[3] While conventional machines do not generally support a fault tolerant atomic output instruction, methods to implement such functionality are described in the literature (see Section 7 for references).

Colors are all equivalent and have no impact on execution. Readers may ignore them for the rest of this section.

***Control flow.*** Faults in data used to determine control flow can clearly have an impact on observable program behavior. In the following example, a single bit flip in the value `x` can cause different results to be printed.

```
if x then out [y1,y2,y3]
else out [z1,z2,z3]
```

Consequently, all control-flow transfers must be coupled with fault-detection checks. Hence, in $\lambda_{\text{zap}}$, `if` statements require a triple of booleans in the primary position. Majority voting detects and recovers from any faults in the boolean before making the control-flow transfer. Any correct $\lambda_{\text{zap}}$ control-flow transfer has this form: `if [eb1,eb2,eb3] then e1 else e2`, where the `ebi` are boolean computations producing equivalent values.

***Functions.*** Function calls entail control-flow transfers and hence, just like `if` statements, must be coupled with a fault-detection check. Moreover, because function arguments are susceptible to faults, it is necessary to pass multiple replicas to a function at any call site. Therefore, any correct $\lambda_{\text{zap}}$ function call has the following form:

```
[ef1,ef2,ef3] [ea1,ea2,ea3]
```

Here, `efi` are the replicated functions and `eai` are the replicated arguments.

While checking for equivalence of integers and booleans is clearly trivial, the discerning reader might wonder how we propose to check for equivalence of functions. Fortunately, it will suffice to check for equivalence of functions simply by comparing function pointers for equality, or, in a system with closures, by comparing closure pointers for equality. Of course, this equivalence test is not semantically complete, but the incompleteness has no impact on the practicality of the system. In fact, it is unnecessary to consider a stronger semantic equivalence checker because a compiler generates the replicated code, and the simplest, most efficient thing for the compiler to do is to create a single function or closure pointer and to thread the same (replicated) pointers throughout the intermediate language code.

To model this design at a high level of abstraction, $\lambda_{\text{zap}}$ allocates function closures and returns three equal pointers to them in a single command. At run time, these pointers will be represented as abstract locations $l$. All functions have multiple arguments and return multiple results. Hence, an example function declaration and use looks like this:[4]

```
let [f1,f2,f3] =
  λ[x1,x2,x3].
    [x1 + R 1,x2 + G 1,x3 + B 1]
in
[f1,f2,f3] [R 7,G 7,B 7]
```

In the code above, `f1`, `f2`, and `f3` are all bound to the location holding the function's closure.

***Summary of $\lambda_{\text{zap}}$ syntax.*** Figure 1 summarizes the syntax of $\lambda_{\text{zap}}$ with integers, booleans, and functions. For completeness, we include the syntax of types, which will be explained in subsequent sections. We only include a couple of simple arithmetic operators ($+$, $\leq$) in our formal syntax, but we will freely use others with obvious meanings in our examples. The only requirement on these operators is that they be total, effect-free functions over all values in the language. When supplied with an unexpected value (e.g., passing `true` to the addition operator), the operator is free to return any arbitrary value of the expected type.

[4] We have omitted typing annotations on function arguments for clarity here.

| | | | |
|---|---|---|---|
| *colors* | $C$ | ::= | $R \mid G \mid B$ |
| *uncolored types* | $I$ | ::= | $\texttt{int} \mid \texttt{bool} \mid T_1 \to T_2$ |
| *colored types* | $T$ | ::= | $C\ I \mid [R\ I, G\ I, B\ I]$ |
| *code locations* | $l$ | | |
| *uncolored vals* | $w$ | ::= | $l \mid n \mid \texttt{true} \mid \texttt{false}$ |
| *colored vals* | $v$ | ::= | $C\ w$ |
| *expressions* | $e$ | ::= | $x \mid v \mid e + e \mid e \leq e$ |
| | | | $\mid\ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$ |
| | | | $\mid\ \lambda[x_1{:}R\ I, x_2{:}G\ I, x_3{:}B\ I].e \mid e\ e$ |
| | | | $\mid\ [e_1, e_2, e_3]$ |
| | | | $\mid\ \texttt{let}\ [x_1, x_2, x_3]\ =\ e\ \texttt{in}\ e$ |
| | | | $\mid\ \texttt{let}\ x\ =\ e\ \texttt{in}\ e \mid \texttt{out}\ e;\ e$ |

**Figure 1.** Syntax of $\lambda_{\text{zap}}$.

In the rest of the paper we will use several abbreviations. For example, triples of equal values $[R\ w, G\ w, B\ w]$ or equal uncolored types $[R\ I, G\ I, B\ I]$ will be abbreviated $RGB\ w$ and $RGB\ I$ respectively. Function declarations will use similar abbreviations: $\lambda[x_1{:}R\ I, x_2{:}G\ I, x_3{:}B\ I].e$ will usually be written $\lambda\vec{x}{:}RGB\ I.e$. Likewise, $\texttt{let}\ [x_1, x_2, x_3]\ =\ e_1\ \texttt{in}\ e_2$ will be written $\texttt{let}\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2$. In general, three related variables $x_1$, $x_2$, $x_3$ will be written $\vec{x}$, and we will use $x_i$ to refer to any one of them. We will also write $\vec{v}$ ($\vec{e}$) for triples of three values (expressions). As usual, we treat terms that differ only in the names of bound variables as equivalent and indistinguishable.

## 3. A Faulty Operational Semantics

We define the meaning of $\lambda_{\text{zap}}$ programs using a small-step operational semantics. This semantics explains how to rewrite one machine state $S$ into another. Machine states are pairs of a code heap $M$ and a program expression $e$. A code heap is a finite partial map from locations $l$ to closed functions $\lambda\vec{x}{:}RGB\ I.e$. We write $M(l)$ for the closed function associated with $l$ in $M$. We write $M, l \mapsto \lambda\vec{x}{:}RGB\ I.e$ to create $M'$, an extension of $M$. We treat machine states $(M; e)$ that differ only because of consistent renaming of locations $l$ as identical and indistinguishable.

The operational semantics depends upon an auxiliary majority voting mechanism $\texttt{vote}_j(v_1, v_2, v_3)$. This operation compares the values contained in $v_1$, $v_2$, and $v_3$, ignoring the associated color tags. In the normal case, the $j$ is 2, and if at least 2 of the 3 uncolored values are equal, that uncolored value is returned; otherwise, all the uncolored values are unequal, and the $\texttt{vote}_j(v_1, v_2, v_3)$ is undefined. For the purposes of proving certain metatheoretic properties of our type system, we will sometimes consider situations in which $j$ is 3. In this case, if all three uncolored values are equal, that uncolored value is returned; otherwise, $\texttt{vote}_j(v_1, v_2, v_3)$ is undefined. Since $j$ is 2 everywhere except in proofs of certain theorems, we will usually omit the parameter $j$ and write $\texttt{vote}(v_1, v_2, v_3)$ instead.

For convenience, we use evaluation contexts $E$ to help describe the operational semantics. The definition of evaluation contexts specifies that evaluation is call-by-value and proceeds left-to-right:

$$
\begin{array}{rcl}
E & ::= & [\,] \mid E + e \mid v + E \mid E \leq e \mid v \leq E \\
& \mid & \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid E\ e \mid [v_1, v_2, v_3]\ E \\
& \mid & [E_1, e_2, e_3] \mid [v_1, E_2, e_3] \mid [v_1, v_2, E_3] \\
& \mid & \texttt{let}\ [x_1, x_2, x_3]\ =\ E\ \texttt{in}\ e \\
& \mid & \texttt{let}\ x\ =\ E\ \texttt{in}\ e \mid \texttt{out}\ E;\ e
\end{array}
$$

To substitute closed expression $e$ for the single hole [ ] in $E$, we write $E[e]$.

More general than an evaluation context $E$ is a fault context $F$. A fault context is used to specify the places a fault may occur. We place the fewest possible restrictions on where faults may occur by defining fault contexts to be arbitrary expressions with a single hole [ ] in the place of any subexpression. To substitute closed value $v$ for the hole in $F$, we write $F[v]$.

The operational semantics is split into a pair of relations. The first relation describes top-level evaluation and has the form $(M; e)_j \longrightarrow_k^s (M'; e')$. Here $s$ is the (possibly empty) sequence of (uncolored) integer values printed by the `out` command. We write nothing for the empty sequence and use commas to separate elements of non-empty sequences. The metavariable $k$ indicates the number of faults. Since we will be using the standard Single Event Upset model, $k$ will always either be 1, indicating a single fault, or 0, indicating no faults. The meta-variable $j$ is the voting parameter and will be set to either 2 or 3, as discussed above. Since $j$ is a global parameter that never changes throughout execution, we will omit mentioning it except in certain theorems and their proofs, writing $(M; e) \longrightarrow_k^s (M'; e')$ instead. Unless notified otherwise, the reader may assume $j$ is 2. The second relation describes fault-free execution of the core commands and has the form $(M; e)_j \Longrightarrow^s (M'; e')$. As with the top-level semantics, we will normally omit the parameter $j$. In general, the reader may assume it is 2. We denote capture-avoiding substitution of a single value $v$ for $x$ in $e$ using the notation $e[v/x]$ and three values for three variables using the notation $e[\vec{v}/\vec{x}]$.

Figure 2 specifies the details of the operational semantics. The key top-level rule, which gives our calculus its name, is *zap*.[5] It specifies that a fault may occur in any fault context $F$, changing any uncolored value $w$ to any other uncolored value $w'$, leaving the color tag (which has no operational effect) unchanged. There are no constraints on the resulting value $w'$ — it may have a different type from $w$ or no valid type at all. The context rule *Octxt* allows for fault-free execution of a core command in some evaluation context. We use $(M; e) \longrightarrow^{*\,s}_{k} (M'; e')$ to denote the reflexive, transitive closure of the top-level operational relation.

We have already discussed the operational semantics of the core commands informally in the previous section, so we will only comment on a couple of key elements of the formal specification. First, notice that the rule *Oadd* depends upon the function `addtot`. This total function over values sums its two arguments if they are integers, and if not, returns an arbitrary integer. The function `lesstot` used in rule *Oless* is similar, but produces a boolean $b$. Intuitively, our rules model the fact that addition and comparisons never get stuck, no matter what bit patterns they operate on. Moreover, external observers cannot tell when these operations are processing corrupted values unless those values are subsequently printed. Second, notice that the rules for `if`, function application, and output all use majority voting to detect and recover from faults before continuing. On the other hand, the `let` forms potentially copy and propagate faulty values without checking for problems.

## 4. Simple Typing for Faulty Computations

The first goal of any $\lambda_{\text{zap}}$ type system is to ensure that well-typed $\lambda_{\text{zap}}$ programs are safe, even in the presence of transient faults. The second goal is to guarantee that faults cannot change observed program behavior, a property we informally call *fault tolerance*.

***What can go wrong?*** $\lambda_{\text{zap}}$ programs can go wrong when supposedly redundant computations actually depend upon one another.

[5] All the other operational rule names are prefixed with *O*; we decided to make this name special.

$\boxed{(M; e) \longrightarrow_k^s (M'; e')}$

$$\frac{}{(M; F[C\ w]) \longrightarrow_1 (M; F[C\ w'])}\ (\textit{zap})$$

$$\frac{(M; e) \Longrightarrow^s (M'; e')}{(M; E[e]) \longrightarrow_0^s (M'; E[e'])}\ (\textit{Octxt})$$

$\boxed{(M; e) \longrightarrow^{*\,s}_{k} (M'; e')}$

$$\frac{}{(M; e) \longrightarrow^{*}_{0} (M; e)}\ (\textit{Oreft})$$

$$\frac{(M_1; e_1) \longrightarrow_{k_1}^{s_1} (M_2; e_2) \quad (M_2; e_2) \longrightarrow^{*\,s_2}_{k_2} (M_3; e_3)}{(M_1; e_1) \longrightarrow^{*\,s_1, s_2}_{k_1+k_2} (M; e_3)}\ (\textit{Otrans})$$

$\boxed{(M; e) \Longrightarrow^s (M'; e')}$

$$\frac{\texttt{addtot}(w_1, w_2) = n}{(M; C\ w_1 + C\ w_2) \Longrightarrow (M; C\ n)}\ (\textit{Oadd})$$

$$\frac{\texttt{lesstot}(w_1, w_2) = b}{(M; C\ w_1 \leq C\ w_2) \Longrightarrow (M; C\ b)}\ (\textit{Oless})$$

$$\frac{\texttt{vote}(\vec{v}) = \texttt{true}}{(M; \texttt{if}\ \vec{v}\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \Longrightarrow (M; e_1)}\ (\textit{Oif1})$$

$$\frac{\texttt{vote}(\vec{v}) = \texttt{false}}{(M; \texttt{if}\ \vec{v}\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \Longrightarrow (M; e_2)}\ (\textit{Oif2})$$

$$\frac{}{(M; \lambda\vec{x}{:}RGB\ I.e) \Longrightarrow (M, l \mapsto \lambda\vec{x}{:}RGB\ I.e; RGB\ l)}\ (\textit{Olam})$$

$$\frac{\texttt{vote}(\vec{v_1}) = l \quad M(l) = \lambda\vec{x}{:}RGB\ I.e}{(M; \vec{v_1}\ \vec{v_2}) \Longrightarrow (M; e[\vec{v_2}/\vec{x}])}\ (\textit{Oapp})$$

$$\frac{}{(M; \texttt{let}\ \vec{x} = \vec{v}\ \texttt{in}\ e) \Longrightarrow (M; e[\vec{v}/\vec{x}])}\ (\textit{Olett})$$

$$\frac{}{(M; \texttt{let}\ x = v\ \texttt{in}\ e) \Longrightarrow (M; e[v/x])}\ (\textit{Olet})$$

$$\frac{\texttt{vote}(\vec{v}) = n}{(M; \texttt{out}\ \vec{v}; e) \Longrightarrow^n (M; e)}\ (\textit{Oout})$$

**Figure 2.** Operational semantics of $\lambda_{\text{zap}}$.

Original code:

```
let [x1,x2,x3] = [2, 2, 2] in
let [y1,y2,y3] = [x1 + x1, x2 + x2, x3 + x3] in
out [y1,y2,y3]
```

Correct transformation (`let` conversion):

```
let x1 = 2 in
let x2 = 2 in
let x3 = 2 in
let y1 = x1 + x1 in
let y2 = x2 + x2 in
let y3 = x3 + x3 in
out [y1,y2,y3]
```

Incorrect transformation (common subexpression elimination) leading to fault-intolerant code:

```
let x1 = 2 in
let y1 = x1 + x1 in
out [y1,y1,y1]
```

**Figure 3.** Erroneous optimization for fault-tolerant code.

For instance, consider the following subtle tweak to one of the examples in the previous section:

```
let [f1,f2,f3] =
  λ[x1,x2,x3].
    [x1 + R 1,x1 + G 1,x1 + B 1]
in
[f1,f2,f3] [R 7,G 7,B 7]
```

Perhaps it took the reader a moment to identify where the error was? Our compiler has accidentally emitted code that uses the same argument `x1` three times in supposedly redundant computations. Consequently, a single fault in the first argument will corrupt the entire computation. To make matters worse, in the absence of faults, the program executes just fine. Hence, no conventional test suite will detect this sort of error. One might try to find the error by injecting faults at random throughout the program and testing it to see what happens, but doing so might still miss the problem. In contrast, type checking is guaranteed to succeed and to pinpoint the location of such mistakes.

***Faulty optimizations.*** It is easy to create arbitrarily many random examples with the sorts of flaws exhibited above. More interesting is the fact that this sort of problem has shown up in practice in the SWIFT optimizing compiler for fault tolerance [31]. SWIFT was architected so that the fault tolerance transformation was a standalone compiler pass that could be inserted anywhere into the backend of the compiler. Initially, the transformation was inserted prior to the major optimization routines in the compiler. Naturally, the implementers wished to exploit their past work on optimizations to improve the performance of their fault-tolerant code. However, they were surprised to find that many conventional optimizations are completely unsound for fault-tolerant code. In retrospect, it might have been obvious that optimizations such as common subexpression elimination and copy propagation eliminated the redundancy specifically introduced to guarantee fault tolerance. A simple example of how this can happen in untyped $\lambda_{\text{zap}}$ code appears in Figure 3. Here, code using triples is transformed into code using ordinary `let` statements, which can more easily be rearranged and scheduled than triples. Next, common subexpression elimination removes all redundancy that existed in the program. This is exactly what happened in an early version of SWIFT – common subexpression elimination and copy propagation combined with other optimizations to eliminate all redundancy in the code. The current SWIFT solution is to do without these optimizations after introduction of redundancy. Of course, this is not ideal either, but at least it is sound until we can devise a sound set of reliability-preserving optimizations — a topic of our ongoing research.

***Colored typing: the setup.*** In order to guarantee safety and fault tolerance, our type system assigns a "color" to every data structure and maintains the invariant that data of color $C$, be it red ($R$), green ($G$), or blue ($B$), are only ever constructed from data with the same color. Consequently, if a fault occurs in one piece of blue data, other blue data may become corrupted, but red and green data will never be compromised. When it comes to a majority vote, and one vote for each color is cast, the votes of the correct red and green data will outweigh the vote of the erroneous blue data.

The types of expressions, which have already shown up in the syntax of our examples, include color tags to keep track of the color of the data produced by the expression. For instance, an expression with type $B$ `int` produces a blue integer. Triples always have the type $RGB\ I$ for some uncolored type $I$.[6] Hence, an expression with type $RGB\ (RGB$ `bool` $\rightarrow RBG$ `int`$)$ produces a triple of functions where each function takes three boolean arguments, one of each color, and returns three boolean results, again one of each color. Figure 1 presented the complete syntax of types.

The type system itself is formalized primarily through an expression typing judgment $\Gamma \vdash^{Z} e : T$, which states that given context $\Gamma$ and *zap tag* $Z$, expression $e$ has colored type $T$. The zap tag $Z$ specifies the location of possible corrupted values: If $Z$ is a color $C$ then there may be errors in data of that color, but not in any of the other colors. If $Z$ is " " (no color) then there are no errors in the data. At compile time, programs are checked under the assumption they contain no errors. The proof of our Type Preservation lemma uses the colored zap tags to demonstrate typing is preserved despite the occurrence of any single fault.

The context $\Gamma$ has the following form:

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, x{:}C\ I \mid \Gamma, l{:}I$$

The hypothesis $x{:}C\ I$ indicates $x$ will be bound to data with color $C$ and uncolored type $I$. There are no hypotheses with the more general form $x{:}T$ because $T$ includes triples; variables are only bound to elements of triples, not triples themselves (which are not real data structures). The second form of hypothesis associates a code location $l$ with an uncolored type I. The type is uncolored because different copies of the same location $l$ may be associated with different colors – if a fault corrupts one copy, it does not necessarily corrupt the others. When a context only contains hypotheses of the form $l{:}I$, we use the metavariable $L$ as opposed to $\Gamma$. Contexts are considered ill formed if the same variable or location appears more than once. We implicitly alpha-vary variables and locations before entering them into the context to avoid repetitions. We consider contexts that differ only in the order of hypotheses equivalent and indistinguishable.

***Colored typing: the rules.*** Figure 4 presents the formal rules for type checking $\lambda_{\text{zap}}$ expressions. This figure presents the most interesting and unusual rule, *Tany*, first. This rule states that when the zap tag is a color $C$, a value $w$ tagged with that color need not be verified and may be granted any type with the form $C\ I$. Hence, the integer 3 could be given a boolean or function type by this rule. Alternatively, a location $l$ that does not appear anywhere in the current code store could be given a valid function type. This will happen when a fault occurs and data colored $C$ no longer satisfies the conventional canonical forms lemma. Nevertheless, the rest of

[6] Recall, $RGB\ I$ is an abbreviation for `[R I, G I, B I]`.

the language is organized to prevent anything from going wrong in such situations.

The rest of the rules for values and the rule variables are quite a bit more standard. Regardless of the zap tag $Z$, integers $n$ may have integer type with a color given by the associated color tag. Likewise booleans have boolean type. Variables and locations have the types ascribed to them by the context. Simple operators such as addition require their arguments to have the same color and return a result with that color, thereby preserving the invariant that faults in one color do not percolate to another. When the zap tag $Z$ is the same color as the arguments of these operators, the arguments might contain faults and therefore might be non-integer values. However, since addition and similar operators are required to be total functions over values, programs will not get stuck here.

The rule for `if` statements requires that the primary argument $e_1$ have type $RGB$ `bool`. In words, this constraint means $e_1$ must evaluate to a triple of booleans, and the booleans will be tagged with different colors. Operationally, a majority vote detects and recovers from any fault before the control-flow decision is made. If a majority is found, the winning value will be a boolean since at least two of the three values voting are tagged with fault-free colors.

Notice, however, that it is possible for there to be no majority. For instance, suppose the compiler generates the following statement: `if` $[R$ `true`$, G$ `true`$, B$ `false`$]$ `then` $e1$ `else` $e2$. This code is obviously incorrect — the generated code should have three equivalent values in the triple. Nevertheless, it type checks provided `e1` and `e2` have equal types. Now, a single fault can corrupt the first value, perhaps changing it to a random code location $l$ giving us the statement `if` $[R\, l, G$ `true`$, B$ `false`$]$ `then` $e1$ `else` $e2$. In this case, voting detects the fault but cannot recover (we assume failed votes lead to a graceful program exit, perhaps alerting the user of the presence of a fault).

A particularly unlucky user can also be hit by a single fault that is *not* detected and changes the behavior of the program. For instance, if the value `R true` changes to `R false`, opposite control-flow branches will be taken in the faulty and non-faulty cases. In either case, programs remain safe.

Despite this difficulty, we can formulate and prove a fault tolerance property for all well-typed programs. Intuitively, if, in a fault-free run, whenever we reach a majority vote, all three values are equivalent, then in a faulty run, at least two of the three will remain the same and continue to be equivalent. Hence, every fault-free run that uses 3-voting will be simulated by every faulty run that uses 2-voting. This result shows that responsibility for correctness is factored between type system and compiler: the compiler is responsible for producing equivalent computations for voting; the type system guarantees their independence.

Returning to our typing rules, one may observe the remaining rules for expressions follow similar patterns to those already discussed. Recall that operationally, a function declaration $\lambda\vec{x}{:}RGB\ I.e$ allocates a closure and returns a triple of locations. Hence the type of such an expression is $RGB(RGB\ I \rightarrow T_2)$. Function application behaves analogously to `if` statements in that the object in the function position must be a triple of locations, all with different colors. The output command also requires its argument be a triple of (integer) values, all with different colors.

Figure 5 presents the typing rules for the abstract machine. The judgment $\vdash^Z M : L$ ascribes type $L$ to the code heap $M$. The judgment $\vdash^Z (M; e) : T$ checks that the complete code heap $M$ is well formed and that $e$ produces a result with type $T$.

***Faulty optimizations revisited.*** Looking back at the faulty optimization illustrated at the beginning of this section, we can see that the $\lambda_{\text{zap}}$ type system detects the problem and rejects the optimized code. More specifically, no matter what color tag we attempt to as-

$\boxed{\Gamma \vdash^Z e : T}$

$$\frac{}{\Gamma \vdash^C C\ w : C\ I}\ (\textit{Tany})$$

$$\frac{}{\Gamma \vdash^Z C\ n : C\ \texttt{int}}\ (\textit{Tint})$$

$$\frac{}{\Gamma \vdash^Z C\ \texttt{true} : C\ \texttt{bool}}\ (\textit{Ttrue})$$

$$\frac{}{\Gamma \vdash^Z C\ \texttt{false} : C\ \texttt{bool}}\ (\textit{Tfalse})$$

$$\frac{\Gamma(x) = C\ I}{\Gamma \vdash^Z x : C\ I}\ (\textit{Tvar})$$

$$\frac{\Gamma(l) = I}{\Gamma \vdash^Z C\ l : C\ I}\ (\textit{Tloc})$$

$$\frac{\Gamma \vdash^Z e_1 : C\ \texttt{int} \quad \Gamma \vdash^Z e_2 : C\ \texttt{int}}{\Gamma \vdash^Z e_1 + e_2 : C\ \texttt{int}}\ (\textit{Tadd})$$

$$\frac{\Gamma \vdash^Z e_1 : C\ \texttt{int} \quad \Gamma \vdash^Z e_2 : C\ \texttt{int}}{\Gamma \vdash^Z e_1 \leq e_2 : C\ \texttt{bool}}\ (\textit{Tless})$$

$$\frac{\Gamma \vdash^Z e_1 : RGB\ \texttt{bool} \quad \Gamma \vdash^Z e_2 : T \quad \Gamma \vdash^Z e_3 : T}{\Gamma \vdash^Z \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}\ (\textit{Tif})$$

$$\frac{\Gamma, \vec{x}{:}RGB\ I \vdash^Z e : T_2}{\Gamma \vdash^Z \lambda\vec{x}{:}RGB\ I.e : RGB(RGB\ I \rightarrow T_2)}\ (\textit{Tfun})$$

$$\frac{\Gamma \vdash^Z e_1 : RGB(T_1 \rightarrow T_2) \quad \Gamma \vdash^Z e_2 : T_1}{\Gamma \vdash^Z e_1\ e_2 : T_2}\ (\textit{Tapp})$$

$$\frac{\Gamma \vdash^Z e_1 : R\ I \quad \Gamma \vdash^Z e_2 : G\ I \quad \Gamma \vdash^Z e_3 : B\ I}{\Gamma \vdash^Z [e_1, e_2, e_3] : RGB\ I}\ (\textit{Ttrip})$$

$$\frac{\Gamma \vdash^Z e_1 : RGB\ I \quad \Gamma, \vec{x}{:}RGB\ I \vdash^Z e_2 : T_2}{\Gamma \vdash^Z \texttt{let}\ \vec{x} = e_1\ \texttt{in}\ e_2 : T_2}\ (\textit{Tlett})$$

$$\frac{\Gamma \vdash^Z e_1 : C\ I \quad \Gamma, x{:}C\ I \vdash^Z e_2 : T_2}{\Gamma \vdash^Z \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2}\ (\textit{Tlet})$$

$$\frac{\Gamma \vdash^Z e_1 : RGB\ \texttt{int} \quad \Gamma \vdash^Z e_2 : T}{\Gamma \vdash^Z \texttt{out}\ e_1; e_2 : T}\ (\textit{Tout})$$

**Figure 4.** Simple type system for $\lambda_{\text{zap}}$.

$\boxed{\vdash^Z M : L}$

$$\frac{\begin{array}{l}\mathrm{Dom}(M) = \mathrm{Dom}(L) \\ \text{for all } l \in \mathrm{Dom}(M), \\ \quad M(l) = \lambda\vec{x}{:}RGB\ I.e \\ \quad L(l) = RGB\ I \to T \\ \quad L, \vec{x} : RGB\ I \vdash^Z e : T\end{array}}{\vdash^Z M : L}\ (\textit{Tmem})$$

$\boxed{\vdash^Z (M; e) : T}$

$$\frac{\vdash^Z M : L \quad L \vdash^Z e : T}{\vdash^Z (M; e) : T}\ (\textit{TS})$$

**Figure 5.** Typing $\lambda_{\text{zap}}$ machine states.

sign to the constant 2 in the final code fragment from Figure 3, the output command will not type check, as `y1` cannot possibly be three different colors at the same time. Interestingly, while some common subexpressions cannot be legally eliminated, others can. When common subexpressions produce values that inhabit the same color, they can be eliminated.

In the SWIFT compiler, we also observed similar sorts of problems occurring with copy propagation optimizations. Depending upon the specifics of how register allocation is implemented, there may be problems there as well. In general, developing a correct, possibly type-directed optimization suite for $\lambda_{\text{zap}}$ programs and its practical counterpart in SWIFT appears to be a rich problem we leave to future work.

## 5. Properties of the Colored Type System

We have proven two important theorems concerning the type system. First, we show that well-typed programs execute safely in the presence of a single fault. By *safe*, we mean that while programs may terminate early when the voting operation cannot find a majority, programs nevertheless remain type safe: they never attempt to use integers or booleans as if they were functions (and vice versa), never call functions that do not exist, and never use non-booleans to make a decision on which branch of an if statement to execute. Second, we show that an execution with one fault simulates the behavior of an execution with no faults, provided that all three values are equivalent in every majority vote in the fault-free execution.

***Type safety.*** In order to state what type safety means precisely in our system, we define the notion of a $j$-safe state. Intuitively, a state is $j$-safe if either it can make *progress* in the usual sense (*i.e.,* it is either a value or it can take a step) or it fails to make progress only because a dynamic voting operation ($\texttt{vote}_j$) is undefined. In the latter case, an implementation would be able to exit gracefully (or throw an exception which might allow recovery). The progress lemma states that well-typed machine states are $j$-safe. The proof of progress follows the usual strategy. Notice that if $\Gamma \vdash^C v : C\ I$ then we can conclude that $v$ has the shape $C\ w$, but $w$ itself has no particular canonical form. This fact does not hinder progress because at function application sites and `if` statements, one takes a majority vote between values of three colors. At least two of the three values do not have color $C$ and therefore do have well-determined canonical forms based on their type.

**Definition 1 ($j$-safe states)**
$(M; e)$ *is a $j$-safe state if one of the following is true:*

- $e$ *is a value $v$ or a triple of values $[v_1, v_2, v_3]$,*
- $e = E[e']$ *and* $(M; e')_j \Longrightarrow^s (M'; e'')$, *or*
- $e = E[e']$ *and $e$ is (*$\texttt{if}\ \vec{v}\ \texttt{then}\ e_1\ \texttt{else}\ e_2$*) or (*$\vec{v}\ \vec{v_2}$*) or (*$\texttt{out}\ \vec{v}$*) and* $\texttt{vote}_j(\vec{v})$ *is undefined.*

**Lemma 2 (Progress)**
*If* $\vdash^Z (M; e) : T$ *then* $(M; e)$ *is $j$-safe.*

The second component of a standard type safety result is a preservation lemma. Proving preservation for the core operational rules is uninteresting – no faults occur here. Proving preservation for the top-level rules, particularly the *zap* rule, depends on the simple *Reliability Weakening* lemma stated below, but it is otherwise not difficult.

**Lemma 3 (Reliability Weakening)**
1. *If* $\Gamma \vdash e : T$ *then* $\Gamma \vdash^C e : T$ *for any color* $C$.
2. *If* $\vdash M : L$ *then* $\vdash^C M : L$ *for any color* $C$.

**Lemma 4 (Preservation: Core Rules)**
*If* $\vdash^Z (M; e) : T$ *and* $(M; e)_j \Longrightarrow^s (M'; e')$ *then* $\vdash^Z (M'; e') : T$.

**Lemma 5 (Preservation: Top-level Rules)**
1. *If* $\vdash^Z (M; e) : T$ *and* $(M; e) \longrightarrow_0^s (M'; e')$ *then* $\vdash^Z (M'; e') : T$.
2. *If* $\vdash (M; e) : T$ *and* $(M; e) \longrightarrow_1^s (M'; e')$ *then* $\vdash^C (M'; e') : T$ *for some color* $C$.

Notice that preservation for the top-level rules involves two cases, one for fault-free execution and one for faulty execution. In the first case, the initial machine state may be well typed under any zap tag $Z$, and the result is well typed under that same tag $Z$. In contrast, in the second case, the initial machine state must be fault free and therefore is typed under the empty (colorless) tag. After a faulty machines step, the machine state is typed under a zap tag for some color $C$.

The final type safety theorem states that if there has been no more than one fault, execution will only lead to safe states. The proof of this theorem uses both cases of the preservation lemma for the top-level rules to show that execution preserves typing. Progress is used to establish that the final state in the execution is safe.

**Theorem 6 ($j$-Safety)**
*If* $\vdash (M; e) : T$ *and* $(M; e)_j \longrightarrow^{*\ s}_{k} (M'; e')$ *and* $k \leq 1$ *then* $(M'; e')$ *is $j$-safe.*

***Simulation.*** We say that one machine state is an error-free simulation of the second machine state when the two states are syntactically identical (modulo consistent renaming of bound variables and locations as usual):

$$(M; e)\ \texttt{sim}\ (M; e)$$

Intuitively, two machine states $C$-simulate when they are syntactically identical except possibly for values with color $C$. More formally, when $M$ and $e$ contain free variables $x_1, ..., x_n$ and there exists substitutions $\theta_1$ and $\theta_2$ such that $\theta_1 = [C\ w_1/x_1] \cdots [C\ w_n/x_n]$ and $\theta_2 = [C\ w'_1/x_1] \cdots [C\ w'_n/x_n]$, the following states $C$-simulate:

$$(\theta_1(M); \theta_1(e))\ \texttt{sim}_C\ (\theta_2(M); \theta_2(e))$$

To refer to either sort of simulation we write $(M_1; e_1)\ \texttt{sim}_Z\ (M_2; e_2)$ where $Z$ is a zap tag (either $C$ or " ").

Our goal is to prove a fault-tolerance theorem: if we begin with a well-typed program, and an $n$-step execution of the program exhibits no faults, then no faulty execution will get stuck, and every $n+1$-step execution with one fault outputs the same values and reaches a $C$-similar state for some color $C$. To achieve this goal, we prove the following two lemmas, which express the fact that simulation is preserved by single evaluation steps. The first lemma describes the situation in which the evaluation step is not faulty (but if $Z$ is the color $C$, there might have been a fault in some past step). The second lemma describes the situation in which the evaluation step is faulty (and there can have been no fault in the past). Notice that Lemma 7 specifies that any vote taken during a step from the first state have all three voters agree (subscript "3" on the evaluation relation) whereas a step from the second state need only have two voters agree. This non-uniformity is necessary to avoid the Byzantine situation discussed earlier involving code that holds votes between inequivalent values even in the absence of faults. Intuitively, the compiler is responsible for producing equivalent computations; the type system guarantees the equivalent computations do not depend upon one another.

**Lemma 7 (Fault-free simulation)**
*If* $\vdash (M_1; e_1) : T$ *and* $\vdash^Z (M_2; e_2) : T$ *and* $(M_1; e_1)\ \mathtt{sim}_Z\ (M_2; e_2)$ *and* $(M_1; e_1)_3 \longrightarrow_0^s (M_1'; e_1')$ *then* $(M_2; e_2)_2 \longrightarrow_0^{s'} (M_2'; e_2')$ *and* $s = s'$ *and* $(M_1'; e_1')\ \mathtt{sim}_Z\ (M_2'; e_2')$.

**Lemma 8 (Faulty simulation)**
*If* $\vdash (M_1; e_1) : T$ *and* $(M_1; e_1)_2 \longrightarrow_1^s (M_1'; e_1')$
*then* $(M_1; e_1)\ \mathtt{sim}_C\ (M_1'; e_1')$ *for some color* $C$.

With these lemmas in hand, we may easily prove our fault tolerance theorem by induction on the length of the evaluation sequence.

**Theorem 9 (Fault Tolerance)**

1. *(Faulty computations do not get stuck)*
   *If* $\vdash (M_1; e_1) : T$ *and in* $n$ *steps,* $(M_1; e_1)_3 \longrightarrow^* {}_0^s (M_1'; e_1')$ *and in* $m \le n$ *steps,* $(M_1; e_1)_2 \longrightarrow^* {}_1^{s'} (M_2'; e_2')$ *then* $(M_2'; e_2')$ *can take at least one more step.*
2. *(Faulty computations simulate fault-free computations)*
   *If* $\vdash (M_1; e_1) : T$ *and in* $n$ *steps,* $(M_1; e_1)_3 \longrightarrow^* {}_0^s (M_1'; e_1')$ *and in* $n+1$ *steps,* $(M_1; e_1)_2 \longrightarrow^* {}_1^{s'} (M_2'; e_2')$ *then* $s = s'$ *and* $(M_1'; e_1')\ \mathtt{sim}_C\ (M_2'; e_2')$ *for some color* $C$.

## 6. From $\lambda_{\to}$ to $\lambda_{\mathrm{zap}}$

To illustrate that $\lambda_{\mathrm{zap}}$ is expressive enough to serve as a typed intermediate language, we have defined a type-preserving translation from the simply-typed lambda calculus to $\lambda_{\mathrm{zap}}$. Our simply-typed lambda calculus contains integers, booleans, functions, and an effectful output command `out`. Unlike $\lambda_{\mathrm{zap}}$, it does not have triples; all functions take a single argument. To simplify the translation and prove correctness, we give the lambda calculus an allocation-style semantics, which, like $\lambda_{\mathrm{zap}}$, and Morrisett et al.'s $\lambda_{\mathrm{gc}}$ [18], allocates function closures on an explicit heap $M$. The syntax, which is largely standard, appears below.

$$
\begin{array}{lcl}
T & ::= & \mathtt{int} \mid \mathtt{bool} \mid T_1 \to T_2 \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}T \mid \Gamma, l{:}T \\
w & ::= & l \mid n \mid \mathtt{true} \mid \mathtt{false} \\
e & ::= & x \mid w \mid e + e \mid e \le e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \\
 & \mid & \lambda x{:}T.e \mid e\ e \mid \mathtt{let}\ x\ =\ e\ \mathtt{in}\ e \mid \mathtt{out}\ e;\ e \\
M & ::= & \mid M, l \mapsto \lambda x{:}T.e
\end{array}
$$

The typing judgment for the lambda calculus has the standard form ($\Gamma \vdash e : T$), and the small-step operational semantics maps

$\boxed{1[\![T]\!] \stackrel{def}{=} I}$

$$
\begin{array}{rcl}
1[\![\mathtt{int}]\!] & \stackrel{def}{=} & \mathtt{int} \\
1[\![\mathtt{bool}]\!] & \stackrel{def}{=} & \mathtt{bool} \\
1[\![T_1 \to T_2]\!] & \stackrel{def}{=} & 3[\![T_1]\!] \to 3[\![T_2]\!]
\end{array}
$$

$\boxed{3[\![T]\!] \stackrel{def}{=} T'}$

$$3[\![T]\!] \stackrel{def}{=} \mathtt{RGB}\ 1[\![T]\!]$$

$\boxed{[\![\Gamma]\!] \stackrel{def}{=} \Gamma'}$

$$
\begin{array}{rcl}
[\![\cdot]\!] & \stackrel{def}{=} & \cdot \\
[\![\Gamma, x{:}T]\!] & \stackrel{def}{=} & [\![\Gamma]\!], x_1{:}R\ 1[\![T]\!], x_2{:}G\ 1[\![T]\!], x_3{:}B\ 1[\![T]\!] \\
[\![\Gamma, l{:}T]\!] & \stackrel{def}{=} & [\![\Gamma]\!], l{:}1[\![T]\!]
\end{array}
$$

**Figure 6.** From $\lambda_{\to}$ to $\lambda_{\mathrm{zap}}$: Types and Contexts.

$\boxed{[\![e]\!] \stackrel{def}{=} e'}$

$$
\begin{array}{rcl}
[\![x]\!] & \stackrel{def}{=} & [x_1, x_2, x_3] \\
[\![w]\!] & \stackrel{def}{=} & RGB\ w \\
[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!] & \stackrel{def}{=} & let\ \vec{x} = [\![e_1]\!]\ \mathtt{in}\ [\![e_2]\!] \\
[\![\lambda x{:}T.e]\!] & \stackrel{def}{=} & \lambda\vec{x}{:}3[\![T]\!].[\![e]\!] \\
[\![e_1\ e_2]\!] & \stackrel{def}{=} & [\![e_1]\!][\![e_2]\!] \\
[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!] & \stackrel{def}{=} & \mathtt{if}\ [\![\ e_1]\!]\ \mathtt{then}\ [\![e_2]\!]\ \mathtt{else}\ [\![e_3]\!] \\
[\![\mathtt{out}\ e_1; e_2]\!] & \stackrel{def}{=} & \mathtt{out}\ [\![e_1]\!]; [\![e_2]\!] \\
[\![e_1 + e_2]\!] & \stackrel{def}{=} & \mathtt{let}[x_1, x_2, x_3] = [\![e_1]\!]\ \mathtt{in} \\
 & & \mathtt{let}[y_1, y_2, y_3] = [\![e_2]\!]\ \mathtt{in} \\
 & & [x_1 + y_1, x_2 + y_2, x_3 + y_3] \\
 & & (\vec{x}\ \text{not in}\ FV([\![e_2]\!])) \\
[\![e_1 \le e_2]\!] & \stackrel{def}{=} & \mathtt{let}\ [x_1, x_2, x_3] = [\![e_1]\!]\ \mathtt{in} \\
 & & \mathtt{let}\ [y_1, y_2, y_3] = [\![e_2]\!]\ \mathtt{in} \\
 & & [x_1 \le\ y_1, x_2 \le\ y_2, x_3 \le\ y_3] \\
 & & (\vec{x}\ \text{not in}\ FV([\![e_2]\!]))
\end{array}
$$

$\boxed{[\![M]\!] \stackrel{def}{=} M}$

$$
\begin{array}{rcl}
[\![\cdot]\!] & \stackrel{def}{=} & \cdot \\
[\![M, l \mapsto \lambda x{:}T.e]\!] & \stackrel{def}{=} & [\![M]\!], l \mapsto \lambda\vec{x}{:}3[\![T]\!].[\![e]\!]
\end{array}
$$

**Figure 7.** From $\lambda_{\to}$ to $\lambda_{\mathrm{zap}}$: Expressions and Code.

machine states to machine states: $(M_1; e_1) \longrightarrow^s (M_2; e_2)$. We omit the inference rules for both of these judgments.

The translation behaves like a highly abstract and idealized version of the SWIFT-R compiler, replicating all computations three times to detect and recover from faults. Figure 6 presents the type translation, while Figure 7 presents the translation of expressions and code heaps.

The main element of interest in the type translation is the translation of function types. A single-argument function in the lambda calculus with type $T_1 \rightarrow T_2$ is translated into a function with a triple of arguments and a triple of results, each with different colors: $3[\![T_1]\!] \rightarrow 3[\![T_2]\!]$. This way, if one of the three individual arguments is zapped, it can be compared against the other two to detect and recover from the fault. To translate the typing context, each individual source variable $x$ is transformed into three target variables $x_1$, $x_2$, and $x_3$, which will again each be bound to values with different colors. We implicitly assume there is some well-defined mapping that generates three unique target variables given any source variable.

The expression translation preserves the basic control-flow structure of the lambda calculus program being translated but replicates the values. Hence, the translation of a source variable $x$ is a triple of related target variables $x_1$, $x_2$ and $x_3$.[7] The translation of any lambda calculus value $w$ is a red, green and blue color-annotated triple of values. Function declarations in $\lambda_{\text{zap}}$ generate a triple of labels, so the translation of a lambda calculus function is a $\lambda_{\text{zap}}$ function with an appropriately translated typing annotation and body. Control-flow operations such as `if` statements and function application are translated by leaving their structure intact and translating their subexpressions recursively. An ordinary lambda calculus `let` statement is translated into a $\lambda_{\text{zap}}$ triple-`let` to preserve the invariant that every source variable $x$ is translated into three variables $x_1$, $x_2$ and $x_3$ in the target. Basic total operations such as addition and inequalities that do not require voting before execution are replicated three times. Notice that in the translation of these operations, the number and order of effects are preserved.

Lambda calculus code heaps are translated by recursively translating each of the elements in the heap.

***Properties of the translation.*** We have proven two important properties of the translation. First, we have shown that the translation is type preserving: well-typed lambda calculus programs are always translated into well-typed $\lambda_{\text{zap}}$ programs. This property establishes the fact that $\lambda_{\text{zap}}$'s type system is sufficiently expressive to check the results of compilation. Second, we have shown that the translation is correct in the absence of faults: if a lambda calculus program takes one step, producing output stream $s$ and resulting machine state $(M', e')$, then its translation will take one or more fault-free steps to produce to same output $s$ and the translation of $(M'; e')$. A key lemma required for this second result is that substitution commutes in an interesting way with the translation.

**Lemma 10 (Compilation preserves types)**
1. *If* $\Gamma \vdash e : T$ *then* $[\![\Gamma]\!] \vdash [\![e]\!] : 3[\![T]\!]$.
2. *If* $\vdash M : \Gamma$ *then* $\vdash [\![M]\!] : [\![\Gamma]\!]$.

**Lemma 11 (Substitution commutes with translation)**
$[\![e[w/x]]\!] = [\![e]\!][R\ w, G\ w, B\ w/x_1, x_2, x_3]$

**Lemma 12 (Compilation is correct)**
*If* $\vdash M : \Gamma$ *and* $\Gamma \vdash e : T$ *and* $(M; e) \longrightarrow^s (M'; e')$ *then* $([\![M]\!]; [\![e]\!])_3 \longrightarrow^{*}{}_0^{\,s} ([\![M']\!]; [\![e']\!])$.

[7] Again, this translation depends upon an implicit map from source variables to unique target variables.

Since the translation is type preserving and correct with respect to the fault-free operational semantics, we may exploit results from previous sections to prove that the faulty lambda calculus $\lambda_{\text{zap}}$ can serve as a correct and reliable platform for implementing the simply-typed lambda calculus.

**Theorem 13 (End-to-end Reliability)**
*If* $e$ *is a lambda calculus program such that* $\vdash e : T$ *and* $(\cdot; e) \longrightarrow^{*\ s} (M'; w)$ *then* $(\cdot; [\![e]\!])_2 \longrightarrow^{*}{}_k^{\,s} (\cdot; \vec{v})$ *with* $k \leq 1$ *and* $\texttt{vote}_2(\vec{v}) = w$.

## 7. Related Work

The process of propagating typing information from source language down through compiler intermediate and target languages is known as *type-preserving compilation*. Java, TIL [39], FLINT [34], Church [9], proof-carrying code (PCC) [21, 3], and typed assembly language (TAL) [19, 17] compilers typically apply some form of this technology in order to generate well-typed compiler intermediate code. Importantly, although these type-preserving compilers advocate using types to improve compiler reliability, they assume perfect hardware and provide no guarantees in the presence of transient faults.

Similar to our formal assumption that data faults can arbitrarily "zap" values at any operational step, related work in the security domain shows how, also by transforming software, to provably enforce interesting control-flow properties when attackers can arbitrarily perturb data memory at any operational step [1, 2]. This enforcement of control-flow integrity (CFI) guarantees that software dynamically obeys predefined control-flow policies, even as attackers have read access to all of memory and write access to data memory. Although enforcing CFI is useful for preventing many kinds of security attacks (such as buffer overflow exploits) and is more efficient than software-based fault tolerance, enforcing control-flow policies in the presence of an attack on a single data value provides a weaker guarantee than that of fault tolerance (where all of the fault-tolerant program's outputs are guaranteed to be equivalent to those of the fault-intolerant version). On the other hand, the formal CFI attack model is stronger than the fault model assumed here in that the CFI model places no restrictions on the number of times an attacker may perturb data memory. In addition, while the work on enforcing CFI provides a formal proof for translating from and to an idealized assembly language, in the present paper we have focused on the translation from a high-level, typed, functional language to a typed intermediate language.

While we have focused on reliability in this paper, these techniques may also have some relevance to security, as one cannot have security without reliability first. For example, Govindavajhala and Appel [11] recently demonstrated that it was easily possible to attack commercial virtual machines running completely type-safe code by inducing and exploiting soft faults. The software protections and type system described in this paper would defeat such attacks.

Software-only approaches to redundancy are attractive because they essentially come *free of hardware cost*. There have been numerous implementations of software-only redundancy, each varying in their level of coverage and performance degradation. At the high-level, Rebaudengo et al. [29] proposed a source-to-source prepass compiler that generates fault-detection code in the source language. Oh et al. [27] proposed EDDI, a low-level detection technique, which duplicates all instructions except branches and inserts validation code before all stores and control transfers, thus ensuring the correctness of values to be written to memory. This work was extended with ED$^4$I [26], which creates a different, but functionally equivalent program by mapping values in the original program to different values in the duplicated program. Reis et

al. [31] proposed SWIFT, which improves on EDDI performance and reliability through better control-flow validation and other optimizations. While the previous approaches have focused on fault detection, Chang et al. [8] proposed two software-only recovery techniques: SWIFT-R, a SWIFT-like technique that uses three versions of instructions with majority voting, and TRUMP, a recovery technique using AN-codes.

Finally, the idea that one can defend against faults by replicating software has been used extensively and studied in great depth, formally and experimentally, for decades in distributed systems. However, the context, assumptions, constraints, and kinds of faults that can occur are clearly different here, making it necessary to study the formal properties of transient faults in standalone hardware as well as in distributed systems.

## 8. Current and Future Work

The faulty lambda calculus $\lambda_{\text{zap}}$ and its simple type system provide a sound theoretical basis for implementation of reliable fault-tolerance systems. However, there is much more research to be done in this area.

One of our immediate concerns involves finding ways to strengthen $\lambda_{\text{zap}}$'s colored type system in order to provide correspondingly stronger fault tolerance guarantees. One possible improvement to our type system involves preventing a compiler from incorrectly generating inequivalent expressions in places where majority voting should occur. For instance, in Section 4 we considered the expression $\texttt{if}\ [R\,\texttt{true}, G\,\texttt{true}, B\,\texttt{false}]\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. To rule out "obviously bad" expressions such as this one, one can add static equivalence checking to the type-checking rules for triples. More precisely, suppose $\Gamma \vdash^Z e_1 \sim e_2$ is valid when $e_1$ and $e_2$ evaluate to equivalent values (in all well-typed contexts specified by $\Gamma$, modulo the equivalence tag $Z$). In this case the following extended typing rule for triple introduction not only checks that the elements of the triple do not depend upon one another, but also that they produce equivalent values.

$$\frac{\Gamma \vdash^Z e_1 : R\,I \quad \Gamma \vdash^Z e_2 : G\,I \quad \Gamma \vdash^Z e_3 : B\,I \quad \Gamma \vdash^Z e_1 \sim e_2 \quad \Gamma \vdash^Z e_2 \sim e_3}{\Gamma \vdash^Z [e_1, e_2, e_3] : RGB\,I} \ (\textit{ETrip})$$

We are in the process of devising an extended type system based on this idea and proving that it gives rise to a stronger safety property for $\lambda_{\text{zap}}$ (voting will never get stuck) and also a stronger fault-tolerance property. [8] However, we note that while these stronger properties are theoretically pleasing, it is not obvious that the more complex type system is more practical. An important strength of the type system presented in this paper is its simplicity, which should make it relatively easy to understand and to implement in a real compiler. Adding equivalence checking, while well-understood in the simply typed lambda calculus (though not, of course, with faults), may well be substantially more difficult to implement efficiently in the context of a realistic compiler intermediate language.

A second direction for future work is the development of a corresponding typed assembly language. The faulty lambda calculus $\lambda_{\text{zap}}$ operates at quite a high level of abstraction. To implement its abstractions in assembly language, the majority voting operations must be compiled into a sequence of comparisons. Developing a low-level type system that allows the comparisons to be scheduled efficiently yet guarantees fault tolerance will be a challenge.

A third aspect of our future work involves developing provably correct, type-preserving optimization infrastructure. As we have observed in this paper, common optimizations undo fault-tolerance transformations. We are in the process of trying to get a better understanding of the range of optimizations affected by this phenomenon and to develop correct new optimizations.

Finally, we are investigating applying the theoretical principles developed here to the intermediate languages used in the implementation of the SWIFT-R compiler system.

[8] In fact, we have already fully defined and proven correct one variant, but are exploring further variations.

## 9. Conclusions

This paper defines $\lambda_{\text{zap}}$, a lambda calculus that exhibits intermittent data faults. These faults may strike any value in the computation at any time, changing that value arbitrarily. Hence, $\lambda_{\text{zap}}$ provides an abstract model of computations performed on modern, occasionally faulty hardware.

In order to detect and recover from faults, $\lambda_{\text{zap}}$ programs compute intermediate results multiple times and use the majority voting mechanisms built in to many of $\lambda_{\text{zap}}$'s primitives. To detect errors in $\lambda_{\text{zap}}$ code that might make it unreliable, we devise a simple type system that uses colors to ensure replicated computations do not depend upon one another. We prove the type system sound, demonstrate that it can catch errors that might occur due to incorrect program optimization, and show that well-typed programs have important fault-tolerance properties. Finally, to prove that $\lambda_{\text{zap}}$ and its type system are sufficiently expressive to serve as an idealized typed intermediate language for compilers, we defined a type-preserving and provably correct translation from the simply-typed lambda calculus into $\lambda_{\text{zap}}$. Our compilation strategy simulates the compilation strategy used in SWIFT-R, a compiler for fault-tolerance. Together, the properties of the $\lambda_{\text{zap}}$ type system and the correctness of compilation provide a strong end-to-end reliability guarantee: Despite being faulty, $\lambda_{\text{zap}}$ is nevertheless powerful enough to implement the lambda calculus correctly. Overall, we believe that this is the first systematic, type-theoretic study of the problem of transient hardware faults.

## References


[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, Nov. 2005.

[2] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. A theory of secure control flow. In *International Conference on Formal Engineering Methods*, Nov. 2005.

[3] A. W. Appel. Foundational proof-carrying code. In *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE, 2001.

[4] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 196–207. IEEE Computer Society, 1999.

[5] 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.

[6] 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.

[7] C. Bolchini and F. Salice. A software methodology for detecting hardware faults in vliw data paths. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2001.

[8] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery methods. In *Proceedings of the 2006 International Conference on Dependendable Systems and Networks*, June 2006.

[9] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *ACM International*

*Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[10] 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.

[11] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, May 2003.

[12] J. G. Holm and P. Banerjee. Low cost concurrent error detection in a VLIW architecture using replicated instructions. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume 1, pages 192–195, August 1992.

[13] 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.

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

[15] 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.

[16] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, Mar. 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[17] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based Typed Assembly Language. *Journal of Functional Programming*, 12(1):43–88, Jan. 2002.

[18] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[19] 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.

[20] 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.

[21] G. Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[22] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

[23] 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.

[24] N. Oh and E. J. McCluskey. Low energy error detection technique using procedure call duplication. In *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*, 2001.

[25] 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.

[26] N. Oh, P. P. Shirvani, and E. J. McCluskey. ED$^4$I: Error detection by diverse data and duplicated instructions. In *IEEE Transactions on Computers*, volume 51, pages 180 – 199, February 2002.

[27] 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.

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

[29] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.

[30] 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.

[31] 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.

[32] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 84. IEEE Computer Society, 1999.

[33] N. Saxena and E. J. McCluskey. Dependable adaptive computing systems – the ROAR project. In *International Conference on Systems, Man, and Cybernetics*, pages 2172–2177, October 1998.

[34] Z. Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

[35] Z. Shao and A. Appel. A type-based compiler for Standard ML. In *ACM Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, June 1995.

[36] 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.

[37] 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.

[38] 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.

[39] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[40] 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.

[41] 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.

[42] 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.

[43] Y. Yeh. Design considerations in Boeing 777 fly-by-wire computers. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 64 – 72, November 1998.

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