

## Well-typed Trustworthy Computing in the Presence of Transient Faults

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, rendering processors that use them more susceptible to *transient faults* caused by energetic particles striking the chip. These faults do not cause permanent hardware damage, but they alter values and corrupt computations. In 2000, Sun Microsystems acknowledged that cosmic rays caused crashes in server systems at major customer sites, including America Online, eBay, and dozens of others. There are hardware-only solutions to the problem of transient faults, but such solutions exact high hardware costs, can't be deployed after-the-fact when transient faults appear in the field, and only provide one-size-fits-all reliability policies.

**Intellectual Merits:** The goal of our work is to develop a trustworthy, flexible and efficient platform for reliable computing in the presence of transient faults. Our multidisciplinary team will do so by developing an end-to-end solution for producing reliable software involving the following components: (1) *Programming language-level reliability specifications* so consumers can dictate the level of reliability they need on a case-by-case and application-by-application basis, (2) *reliability-directed compilation* so compilers can interpret programmer reliability specifications and introduce the redundancy necessary to tolerate the specified faults, (3) *typed intermediate languages* with novel type systems capable of automatically verifying reliability properties of intermediate program representations, (4) a new framework for *type- and reliability-preserving optimizations* as many current, common compiler optimizations are, surprisingly, *not* reliability-preserving, (5) *automatic, machine-level reliability verification* so compiler output can be proven reliable, (6) new *software-modulated fault tolerance techniques* at the hardware/software boundary to implement the reliability specifications, including both software-only and hybrid hardware/software techniques, to maximize speed and minimize power, offering reliability at the right cost, and finally (7) *microarchitectural optimization* that explores opportunities to trade reliability of components off against demands for performance, power, and cost.

Together our multi-disciplinary team is uniquely qualified to carry out this research. Walker and his collaborators developed the first design, implementation, type-preserving compiler for, and theory of Typed Assembly Language (TAL) [39, 40, 38, 36]. Appel was the main developer of SML of NJ [5] and introduced and implemented the concept of Foundational Proof Carrying Code [4], a framework for trustworthy implementation of Typed Assembly Language. August has already done ground-breaking work in software fault tolerance, introducing the concept of software-modulated fault tolerance [57, 58], developing novel HW/SW hybrid fault tolerance techniques, and recently winning the best paper award at CGO (the International Symposium on Code Generation and Optimization) [56]. Martonosi and Clark are experts in microarchitectural- and circuit-level exploitation of unreliable components to reduce power and cost [28].

**Broader Impact:** The PIs have a *proven track record* when it comes to integrating research and education. First, Walker has organized a series of summer schools on software security and reliability over the last three years [7, 8, 6], taught by experts brought from around the world, and involving over 150 participants in three years. He will continue his mission to develop these schools in the years to come. Second, combined, the 5 PIs have supervised approximately 68 undergraduate student-semesters of research over the past 5 years, and will continue to do so. Third, the PIs will remain active in increasing the participation of underrepresented groups in science and engineering, by, for instance, continuing their commitments to groups such as GWISE (Princeton's Graduate Women in Science and Engineering) and the NSF/CRA distributed mentoring program, as Martonosi has done. In addition to educational impact, the PIs also have extensive links with industry to ensure technology transfer of the proposed research. The PIs can accomplish this by disseminating software to the research community without restriction, as August has done with his Liberty Simulation Environment and fault-tolerant compiler work.

## 2 Well-typed, Trustworthy Computing in the Presence of Transient Faults

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 [13, 45, 65], rendering processors that use them more susceptible to *transient faults*, which are caused by 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, each processor generation will need to devote more and more attention simply to maintain the current level of reliability.

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. As a matter of fact, 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 [14].
- 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.” [82]
- 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” [82]. (Emphasis added was our own.)
- At the Los Alamos Neutron Science Center, Hewlett Packard acknowledged their AlphaServer ES45 supercomputer was frequently crashing due to soft errors [34].

Hence, reliability in the presence of transient faults is already a significant cause for concern at major semiconductor manufacturers and threatens to be more so in the future. For further compelling evidence, we strongly encourage all reviewers of this proposal to read the attached letter of support by Shubu Mukherjee, Director of Intel Research’s Fault Aware Computing Technology (FACT) Group and Co-Director of the Architecture Modeling Infrastructure (AMI) Group. He is a world expert on transient faults and their impact on hardware design, now and in the future. In his words,

Soft errors pose a key obstacle to the growth of the semiconductor industry. Moore’s Law-the continuous exponential growth in transistors per chip-has brought tremendous progress in the functionality and performance of semiconductor devices, particularly microprocessors. However, each succeeding technology generation has also introduced new obstacles to maintaining this growth rate. Transient faults due to single event upsets have emerged as a key challenge whose importance has increased dramatically and will continue to increase through the next decade.

**The Case for Software-Modulated 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 faults, 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 publish 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. Mukherjee speculates 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.

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 which 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.

If software-only fault tolerance techniques are the rapid response solution for today and tomorrow, *hybrid hardware-software* techniques are the flexible, efficient and reliable solution for our next-generation processors of the future. Hybrid solutions can provide a level of protection unavailable in software-only solutions — all currently known software-only techniques possess a *window of vulnerability* during which a bitflip can cause potential harm. The window of vulnerability may be narrowed in software-only solutions, but not entirely eliminated as current hardware lacks the necessary atomic, protected instructions. We call a hybrid solution *software-modulated* (i.e., *software-controlled*) when the hardware provides specific instructions and functional units to protect against faults, but these units have low hardware implementation costs and may be controlled and selectively invoked by the software. Such hybrid, software-modulated solutions will be the next to be deployed because of their low-cost and high performance. The true winners will be those future solutions that best exploit the capabilities of software and hardware to provide maximum performance in the widest range of different environments at the lowest cost with the ability to trade off reliability guarantees against other considerations as the consumer requires it.

**A Trustworthy, End-to-end Solution.** Replacing hardware fault tolerance with software (or software-modulated) 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, programs of this size 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 they face are many orders of magnitude more difficult to overcome, yet their 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: user registers, the program counter, conditional 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 starts at beyond-impossible and ranges to who-knows-what. 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 when they do, they 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* [68, 62, 40, 37, 4] 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 that signal the compiler they are using has a bug, finding out earlier is substantially better than finding out late and suffering expensive consequences.

Type checking intermediate language programs is an important complement to conventional testing, because 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 input. So far, type preserving compilers have only been used to verify standard sorts of “type safety” and “memory safety” properties and, crucially, they 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. On this basis, we will be able to develop the first truly trustworthy, flexible, and reliable software-modulated systems capable of detecting and recovering from transient faults.

**Summary: The Proposal’s Intellectual Merits.** The goal of our work is to develop a new, trustworthy, flexible and efficient platform for reliable computing in the presence of transient faults. Our team will do so by developing an end-to-end solution for producing reliable systems involving the following components:

1. **Reliability specifications.** To allow programmers a choice of reliability policies, we will introduce new semantically meaningful language features that allow programmers to specify the reliability they require for their application. Policy specifications will be very lightweight and as course-grained (at the level of whole programs) or fine-grained (at the level of modules or particular data structures) as the programmer chooses.
2. **Type- and reliability-directed compilation.** Reliability specifications will be interpreted by the compiler and direct it to introduce sufficient redundancy into the computation. Once the necessary redundancy has been introduced, new type systems we will design, implement, and prove sound will be used to automatically verify reliability properties.
3. **Type- and reliability-preserving optimization.** In August’s previous work on fault tolerance [56, 57, 58], he has observed that conventional compiler optimizations such as common subexpression elimination and copy propagation can and do eliminate artificial redundancy introduced for the purpose of fault tolerance and consequently has had to turn such optimizations off in his previous work. We propose to attack this problem and develop a completely new theory and implement a new framework for *sound* reliability-preserving optimization. Our new optimization framework will operate over the typed intermediate languages that we design and will be guaranteed to produce type-safe (ie: verifiably fault-tolerant) code.
4. **Machine-level reliability verification.** The output of our compiler will be a novel Typed Assembly Language (TAL) that, like our compiler intermediate languages, can be automatically checked for sophisticated type safety properties that imply fault tolerance. While conventional testing techniques will never guarantee that compiled programs are truly fault tolerant, we will prove that for a particular, but realistic fault model, our assembly language type system guarantees fault tolerance for all possible program inputs and all possible faults. More precisely, since we will explore a number of fault-tolerance schemes, which blend in both hardware and software fault tolerance techniques, we will develop a family of type systems capable of checking the different constraints required by the different schemes.
5. **New software-modulated techniques for fault tolerance.** We will develop new software-modulated techniques to implement the reliability specifications. This work will build off and improve our previous software-only [56] and hybrid hardware/software techniques [57, 58] to maximize performability, offering reliability at the right performance cost.
6. **Microarchitectural optimization.** With software-modulated techniques providing fault protection, we are free to explore new techniques which exploit intentional reduction in component reliability to reduce power, reduce cost, or improve performance.

Together our multi-disciplinary team is uniquely qualified to carry out this research. PIs Appel and Walker are experts in programming language design and type-preserving compilation. Walker and his collaborators developed the theory and first implementation of Typed Assembly Language (TAL) [39, 40, 38]. Appel was the main developer

of SML of NJ, the first efficient compiler and garbage collector for the SML programming language [5] and introduced and implemented the concept of Foundational Proof Carrying Code [4], a framework for trustworthy implementation of Typed Assembly Language. PI August is an expert in the integration of compiler and hardware technology. His group has already done ground-breaking work in software fault tolerance, introducing the concept of software-modulated fault tolerance [57, 58], developing novel HW/SW hybrid fault tolerance techniques, and recently winning the best paper award at CGO (the International Symposium on Code Generation and Optimization) [56]. Margaret Martonosi and Doug Clark are experts in microarchitectural- and circuit-level exploitation of unreliable components to reduce power and cost [28].

**Summary: The Proposal’s Broader Impacts** The PIs have extensive links with industry to ensure technology transfer of the proposed research. The PIs also have a track record of disseminating tools and software to the research and education community without restriction. Compiler, run-time, and profiling tools developed as a result of this work will be treated no differently. Appel’s previous work on implementing the Standard ML of New Jersey compiler, the most widely used SML compiler, and his resulting textbook *Compiling with Continuations* [3] are illustrative examples of the broad impact this group of PIs can make. Already, in fact, August’s prior (unsupported) work in this area has led to the distribution of a reliability-aware version of the IMPACT compiler, currently in use in academia and industry by at least four other research groups.

The proposed support will also help the PIs continue to serve several educational objectives. First, over the last three years, Walker, has organized a series of summer schools on secure and reliable computing. These summer schools [7, 8, 6] have brought in top researchers from all over the world to teach a well-structured curriculum on various topics in security and reliability to groups of senior undergraduate, graduate, postdoctoral, and, even in one exceptional case, high-school students. There is simply no other program in North America that compares to the program Walker has put together. He pledges to continue this important effort if funded. Second, the PIs will continue to supervise numerous student-semesters of undergraduate research, involving more students in the work proposed here. Over the last 5 years, the 5 PIs supervised approximately 68 student-semesters worth of research in total. Third, the proposal will support the development of publicly available course material on reliability. This material will be developed in consultation with leading reliability researchers and will be designed to prepare upper-class undergraduate and graduate students for the increasing reliability challenges of the next 10 years.

In addition to their other endeavors, the PIs will remain active in increasing the participation of underrepresented groups in science and engineering. Martonosi, for instance, is a clear leader in this area, as she has worked with a number of groups, ranging from GWISE (Princeton’s Graduate Women in Science and Engineering) to the NSF/CRA distributed mentoring group, which pairs top female scientists with female undergraduate students for a summer of research.

### 3 Research Plan

In this section, we will further explain how we plan to develop the first truly trustworthy and flexible platform for reliable computing in the presence of transient faults. More specifically, the following subsections explain these components of our proposal in order:

- Research on the theory, design, and implementation of high-level, semantically meaningful specifications of reliability requirements and techniques for analyzing and compiling source-level code into intermediate languages in order to guarantee such requirements are met.
- Research on the theory, design, and implementation of compiler intermediate languages equipped with novel, provably sound type systems capable of guaranteeing specified reliability requirements. Further research on the development of a Typed Assembly Language with an efficient type checker capable of verifying reliability properties of the machine code generated by our compiler infrastructure.
- Research on the theory, design, implementation and evaluation of novel reliability-aware and type- and reliability-preserving optimizations.
- Research on the development and correctness of new machine-level fault tolerance techniques that use novel combinations of software and hardware protections.

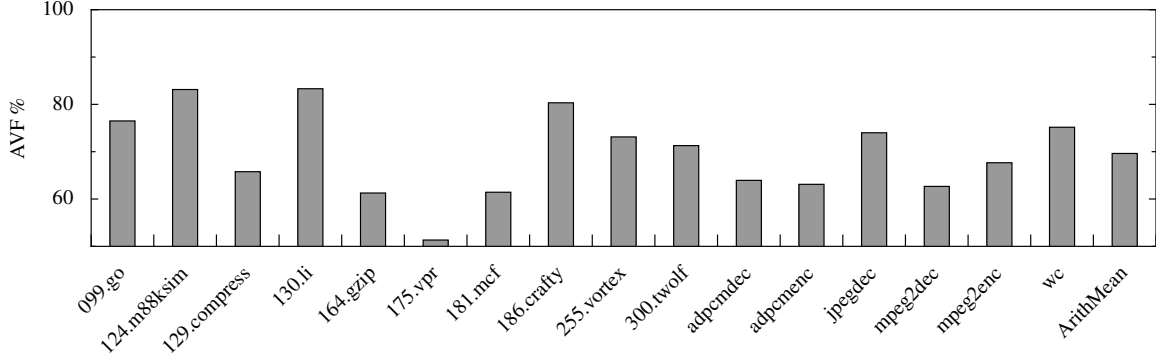


Figure 1: Natural reliability of applications

- Research on hardware design and optimization cognizant of the trade-offs between cost, performance, power, and reliability.

### 3.1 Specifying Reliability Requirements

Different applications, or the same applications used in different places, can exhibit remarkably different degrees of reliability. First, to get a sense of the impact of environment, consider that the fault rate flying in an airplane over the North Pole can be 100 to 500 times the fault rate at sea level in New York due to cosmic radiation. In future, a Wall Street banker who can accept lower fault-tolerance protection in her office, may need a substantially better protected system on her laptop during the trip over the pole to Hong Kong [82].

Second, to give a quantitative idea of the variation between applications, examine Figure 1, which shows the natural reliability of a variety of programs without any protection added to them. This figure measured the reliability using the architectural vulnerability factor (AVF) metric [55] (for the purposes here, the percentage of single-bit faults which lead to incorrect output) under a Single Event Upset (SEU) model. More specifically, we toggled a single bit in the register file, chosen at a random point across the entire run of the application, and then measured whether there was any difference in the observable outputs of the application. In this experiment, the per application reliability varied from 51% for `175.vpr` to 83% for `124.m88ksim` and `130.li`.

Third, the disparity in natural fault tolerance will be even greater when the semantics of the application is taken into consideration. For example, a fault in a single pixel in a single frame in the video stream playing at 30 fps is unlikely to be noticed by the user, so applying fault tolerance in this case would be superfluous and even detrimental if the increased reliability comes at the cost of choppy playback. While a bank might tolerate intermittent errors that perturb web site display, any interruptions in overall web site availability or errors in transactions could be enormously costly.

In order to allow programmers to control the trade-off between reliability and performance, we propose to develop a system of *lightweight* source-level reliability specifications. Our goal will be to give programmers control when they require it, and yet avoid placing any additional programming burden on users. Programmers unconcerned with performance, or conversely, unconcerned with reliability, will be able to write completely ordinary, unannotated code, and specify a uniform policy for their entire program with a simple compiler switch. Crucially, if faults are causing substantial unforeseen problems for your business or institution, as they did at the Los Alamos Neutron Science Center (see Section 2), the compiler switch can be changed (`-R2` to `-R4`), the code recompiled, and the improved fault-tolerant application deployed. We will also allow programmers concerned with fine-tuning the reliability-performance trade-off to override the default policy with fine-grained reliability specifications at the level of interfaces and modules, and more fine-grained still at the level of individual data structures.

While the specific details of the structure and syntax of programmer annotations will be an important topic of research, we initially envision using *type qualifiers* with the form `rel(k)` to specify reliability. Intuitively, a data structure (or module) qualified by `rel(k)` should be resilient against  $k$  simultaneous faults. Given a collection of such qualifier specifications, it will be the job of the compiler to automatically introduce sufficient redundancy into the computation to supply the required reliability. We propose to develop the static analyses necessary to make this possible.

One crucial, new insight that will allow us to solve this problem is that guaranteeing reliability is a classic *data and control-flow integrity* problem. For instance, a data structure annotated `rel(k)` can resist  $k$  faults only if all

data flowing to, and stored in, the structure is uncorrupted — ie, it too must resist  $k$  faults. Moreover, control flow leading to any data structure assignments must also be unperturbed in the presence of  $k$  faults. The classic solution to an integrity problem like this one is to perform an information flow analysis, and there has been much research on this topic over the last several years which we hope to exploit. Our team will study and identify the best information flow inference techniques and adapt them to our problem. With the information provided by information flow analysis, we believe we will be able to label all data and operations with the necessary reliability level. After labelling, the compiler will in turn be able to introduce the necessary redundancy into compiled code. However, the specifics of the analysis and program transformation will constitute a key research focus for us.

In addition to designing and implementing our language and associated compiler framework, we will formalize the semantics of our language and the corresponding static analysis. Our proposed work will also include usability studies of these annotations: their utility and their ability to give programmers more fine-grained control over reliability and performance. Our language and compiler framework will be freely available on the Web and we will encourage adoption and respond to feedback.

**Security vs. reliability.** There has been much previous work on the use of type systems to provide security against malicious attackers, which is a quite different problem from what we propose, which is to provide reliability in the face of random bit-flips. Simple type systems such as Java’s can guarantee that clients of an interface don’t read or modify internal data structures of other modules; such type systems intrude minimally on the programmer, but mean that inadvertent programming mistakes can reveal information. Type systems for information-flow security, including Jif [42] and FlowCaml [52], guarantee global security properties even in the face of programmer error, but the type systems are extremely cumbersome for the programmer.

We emphasize two points: First, our technique will not impose any cumbersome type system on the programmer. We do not need to prevent any programs from being written or compiled. We will allow all programs to compile; the reliability annotations serve to direct how much redundancy and replication of user code is required. All of our high-tech type systems will be used in internal compiler intermediate languages, to verify that the compiler correctly replicates as necessary.

Second, one cannot have security without reliability. Govindavajhala and Appel [24] have shown that a malicious attacker can easily turn single-bit upsets into successful attacks; the attacker can write a program that type-checks in Java (or in an information-flow type system), such that any single-bit error permits the cast of any pointer into any type. They demonstrated this in practice on real hardware, using a light bulb to warm the memory chips of a PC until bits started to flip. They also show that the use of ECC memory, along appropriate logging and response to ECC-detected bit-errors, is a good defense; but they showed no defense against bit-errors in processor datapaths. Our proposed research will provide such a defense. In summary, bit errors can be used to defeat type systems or static analyses used for purposes of security against malicious attackers, and our new research will not only provide reliability but will also support the security guarantees of static security mechanisms.

### 3.2 Typed Intermediate and Typed Assembly Languages

A central element of our compiler research will focus on the development of novel, provably sound *typed compiler intermediate and target languages* (TILs). These TILs will be capable of representing the results of compilation from our proposed source language. These results include code and data as well as reliability requirements, which will be represented as types in the intermediate language. After any compiler phase, it will be possible to type check the resulting intermediate program representations. Type checker success will indicate that reliability requirements are guaranteed; typechecker failure will indicate that there is a potentially dangerous compiler error and reliability requirements are *not* guaranteed.

Before defining the type systems themselves, it is necessary to develop *formal fault models* that specify the nature of the faults that can occur. With fault models in hand, the system vulnerabilities will be well-defined and design of type system support for protection against these faults may begin. Initially, we will focus on two different sorts of faults:

- **Data faults.** In general, data faults occur when integers, pointers or other data values, either in registers or in memory are corrupted by transient faults. However, the specific data values that can be corrupted will vary depending upon assumptions about hardware protections. For instance, some storage structures such as caches and memory include error correcting codes (ECC) and parity bits so the redundant bits can be used to detect or even

correct the fault. (Note that while ECC can protect storage and busses, it cannot protect general computation.) Hence, depending upon reliability requirements, ECC may be sufficient to protect memory and only register faults need be modeled. August’s (Co-PI) SWIFT system [56] assumes that ECC protections are available in the memory hierarchy. However, we will develop and analyze precise fault models for a variety of different hardware configurations.

- **Control-flow faults.** Control-flow faults occur when transient faults interfere with the program counter or instruction execution in the midst of a control-flow transfer. Control-flow faults may cause control to jump to almost arbitrary code locations. While protection against such faults may seem near impossible, researchers have developed software techniques, which work in concert with hardware protections, to successfully detect and recover from such faults. Once again, August (Co-PI) has developed and implemented such techniques for his SWIFT and CRAFT systems.

In preparation for this proposal, we have begun to develop formal fault models, a compiler intermediate language and a type system to guarantee resistance against data faults. If funded, we will carry through our preliminary investigations of data faults and then in later years of the proposal, focus our attention on control-flow faults.

At present, we are exploring a very strong model for data faults in a formal operational semantics. This model allows any program value to be “zapped” and rewritten as some arbitrarily different value. Given a machine state  $(M, E[v])$  where  $M$  is a memory,  $E[\ ]$  is an evaluation context, and  $v$  is a value, we model a value fault formally using the operational rule:

$$(M, E[v]) \longrightarrow (M, E[v']) \quad (\text{rule zap-val})$$

A memory fault may be modeled using a similar sort of rule.

Type theorists will notice immediately that such a fault model is problematic for ordinary typed programming languages. Since the value  $v$  becomes some other arbitrarily different value  $v'$  (with arbitrarily different type, or no valid type at all), the standard, yet essential Type Preservation theorem will not hold. Moreover, if the corrupted value  $v'$  is copied by the program code, a single fault may result in arbitrarily many faulty or corrupted values — checking for faults before every single instruction to prevent propagation of corrupted values is simply too expensive, and it turns out, not necessary.

Existing fault tolerance systems (the unverified ones), if they are correct, avoid “going wrong” in such circumstances by generating code that performs several independent, but identical, redundant computations. A key invariant is that while the corrupted values have unexpected types, and are copied arbitrarily many times, values from one independent computation do not flow into another. Before critical operations such as an indirect jump through a potentially corrupt pointer or a memory dereference occurs, results from these independent computations are compared against one another to detect faults.

In order verify and enforce this key invariant used by existing fault tolerance systems, we are developing a novel type system to track the multiple independent computations. Our type system assigns a different “color” to the data used in each different independent computation and guarantees that, for instance, red data only depends upon other red data and blue data only depends upon other blue data, etc. Hence, if a fault occurs in red data, arbitrarily many red values may be corrupted, but no blue or green values will be corrupted. Our system also requires that prior to executing any safety-critical operation, equivalent (but independently computed) red, blue and green values be compared to test for faults.

The colors assigned to datastructures will be carried around in types as type qualifiers, resulting in a type algebra with the following (rough) form:

$$\begin{aligned} (\text{colors}) \quad c &::= \text{red} \mid \text{blue} \mid \dots \\ (\text{types}) \quad \tau &::= c \text{ int} \mid c (\tau_1, \dots, \tau_m) \rightarrow (\tau'_1, \dots, \tau'_n) \mid \dots \end{aligned}$$

Typing rules will prevent data with one color from being used to construct data with a different color – such a dependence would incorrectly propagate faults from one color to another. Typing rules will also guarantee that when checking for faults, data with different colors are compared, *not* data with the same color. If data with the same color were compared, a single fault could fool the fault detector.

Our preliminary work has involved developing an idealized typed intermediate language, based on a lambda calculus with faults. We are currently in the process of attempting to prove that our type system guarantees the underlying code is fault-tolerant. Our progress is very promising, but the proof remains to be completed. Once complete, we

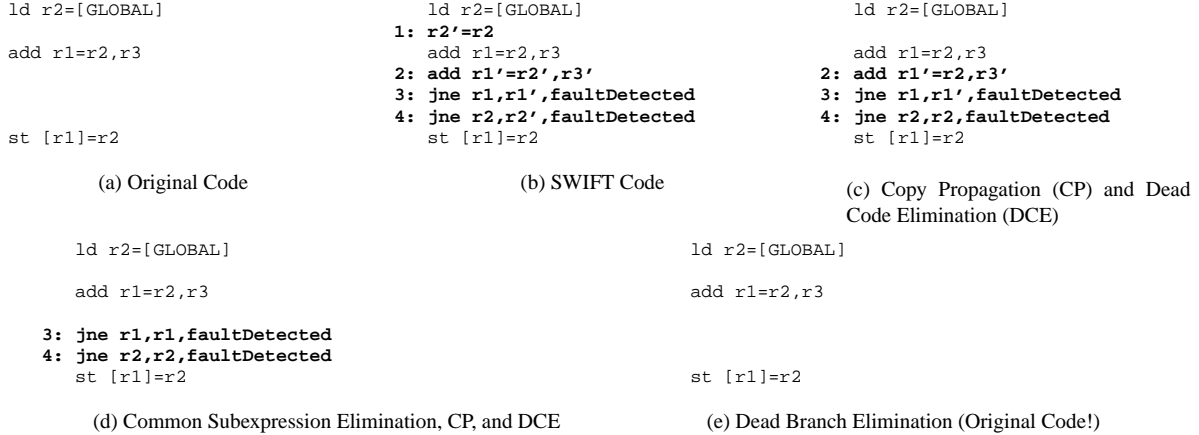


Figure 2: The SWIFT scheme and the effect of subsequent optimizations upon it.

will have a sound basis for implementation of a practical typed compiler intermediate language, though moving from an idealized lambda calculus to a complete compiler intermediate language is never trivial. Moreover, once we have typed compiler intermediate languages, we must also develop a typed assembly language with fault-tolerance checking capabilities. As high-level languages are compiled into lower-level languages, typing becomes more difficult since the high-level abstractions are broken down into many low-level instructions that are transformed and scheduled for optimal performance. Once the question of data faults is understood, the problem of verifying code at risk from control-flow faults persists. Finally, as we plan to develop new software-modulated hybrid techniques, each of these new techniques will carry with them new constraints on software that must be checked to guarantee correctness. Hence, if funded, we anticipate developing an entire family of related type systems, each capable of checking different constraints dependent on the details of the hardware architecture and assumptions concerning the fault model. As PIs Appel and Walker are world experts in the theory, design and implementation of typed intermediate languages and typed assembly language, and August, Clark, and Martonosi have equal expertise in the details of fault tolerance and hardware design, we are convinced, if funded, we will be able to discover, implement and prove correct solutions to this substantial series of challenges.

### 3.3 Reliability Preserving Optimization

To defend against transient faults, redundancy of computation must exist either in hardware or software. In software fault tolerance, this redundancy comes in the form of additional code operating upon additional state. The simplest method to address transient faults may be to run the same program multiple times, taking the most common result as correct. Unfortunately, this approach does not work for systems with external interaction, such as with users, I/O devices, or the network. For these “live” systems, redundant computation with redundant state must be performed concurrently, and the results must be checked before every I/O operation. Further, for systems with memory-mapped I/O, checks must be made before every load or store operation.<sup>1</sup> The SWIFT (software-implemented fault tolerance) technique developed by PI August, like techniques developed by others [54, 49], takes this approach [56].

Figure 2 shows a trivial segment of code made redundant by a simple SWIFT scheme. The SWIFT scheme shown here is for a machine with ECC on the memory subsystem and without any ISA support for software-modulated fault tolerance. In all SWIFT schemes, computation instructions are duplicated (as instruction 2 in Figure 2b) and set to operate on a second, redundant set of registers (denoted in the figure with tick marks). For the given machine, SWIFT cannot duplicate loads and stores for reasons given earlier, among others [56]. After each load, the register corresponding to the destination register in the redundant register set is updated using a `mov` instruction (instruction 1 in Figure 2b). Before each store, the original and redundant registers used by the store are checked for equivalence (instructions 3 and 4 in Figure 2b). If there happens to be a mismatch, a fault has been detected and recovery by some means must be initiated. If the register values match, the store proceeds normally. Note that there exists a small *window of vulnerability* between the load and the move and between the checks and the store. While a change to a

<sup>1</sup>Loads from incorrect addresses often have deleterious effects, perhaps by throwing segmentation faults or by performing extraneous reads from side-affecting I/O mapped addresses.

register r2 between instructions 1 and 4 in Figure 2b will be detected by the checking branch instruction, a change before 1 or after 5 will go undetected. These windows of vulnerability will be revisited in the next section, where we propose to explore lowcost hardware support and flexible hybrid hardware-software fault tolerance techniques in addition to software-only techniques. For now, let’s consider the effect of optimizing this code in the compiler.

In our prior work, SWIFT was created as a pass which could be performed at any point in relation to the other optimizations. Originally, the idea was to apply SWIFT early among the other optimizations (in particular instruction-level parallelism (ILP) optimizations) to allow them to minimize SWIFT’s performance impact. As the rest of Figure 2 demonstrates, the optimizer does an excellent job at minimizing the performance degradation, but it does so by eliminating the enhanced reliability! Figure 2c shows the SWIFT code from Figure 2b after copy propagation and dead code elimination have been applied. Together, these optimizations have removed the redundant copy of r2 from the code. The next step (Figure 2d) removes the redundant computation, and the final step (Figure 2e) rightly removes the now dead branches. The resulting code in Figure 2e is identical to the code in Figure 2a. In retrospect, this result is expected, as reliability is gained through redundancy while many performance optimizations work by removing redundancy. Interestingly, despite not having applied any post-processing optimizations, the additional SWIFT code (a redundant copy of each instruction, roughly double the register pressure for the redundant state, and additional code to perform checks) degrades performance by only 43% (geometric mean) on a wide-issue machine (Itanium 2) across a suite of 30 SPEC, MediaBench, and UNIX benchmarks when compared to the highly optimized unprotected code.

While 43% is much less than the more than 2x performance degradation one might expect, optimizations (ILP optimizations in particular) should be applied to further reduce this number. To make this possible, we propose to develop a novel reliability-preserving and type-preserving optimization infrastructure. The key components of this research include first identification of all common optimizations that can have an impact on reliability — we are concurrently aware that common subexpression elimination, copy propagation and register allocation pose problems, but further research may reveal others. Second, we will formulate and prove correct variants of these conventional optimizations, providing theoretical guarantees that the new algorithms preserve typing and reliability. Third, we will implement the dataflow analyses and transformations within the context of our type intermediate languages, propagating typing information through the transformations and type checking afterwards to guarantee our theory has been implemented correctly in practice. Fourth, we will evaluate the effectiveness of the optimizations and measure the true, unavoidable cost of software-implemented fault tolerance. Understanding the costs of fully-optimized software-implemented fault tolerance will further inform future research decisions on the appropriate trade-off between software and hardware protections.

While we will tame existing *performance optimizations*, new *reliability optimizations* will also be developed. For example, in C, Booleans are often implemented using 32-bit or 64-bit integer registers holding the values 1 or 0. With only a hamming distance of one between the two values, we are asking for trouble. New analyses and optimizations can be employed to replace the values used with 0xFF..FF and 0, increasing the hamming distance and the reliability without an impact on performance. In cases where the values consuming more bits, such as arbitrary 8-bit chars in 32- or 64-bit registers, masking all unused bits immediately prior to use can improve reliability with negligible performance cost. In addition redundancy within the register can be created by replicating the char 4 or 8 times across adjacent bit ranges. Arithmetic codes can also be used through instructions which preserve their properties. Another already identified optimization involving changes to instruction scheduler to reduce the window of vulnerability is described at the start of the next section. While we cannot list all the optimizations we will invent, we plan to allow the code and the compiler to guide us in our search for them. Moreover, as new optimizations prove fruitful, they may necessitate rethinking and redesign of our type systems. We are ready to take on such challenges as they arise in our research.

### 3.4 Software-Modulated Fault Tolerance Techniques

While the work described to this point provides assurance that the code is formed correctly for the purposes of fault tolerance, the resulting code reliability still depends on the quality of the software-modulated fault tolerance (SMFT) technique employed. Consider, again, the example in Figure 2b. As mentioned earlier, there is a window of vulnerability between the load and the move (instruction 1) and between the checks (instructions 3 and 4) and the store. This window of vulnerability is a consequence of SWIFT, the SMFT technique employed, having no means to correctly create redundant loads and store. (Recall that even loads may have side-effects in I/O and cannot be executed twice unless proven safe in this respect and others [56].) The best that can be done without hardware support is to keep the window of vulnerability small. In general, by keeping the live ranges of uniquely held values (r2 in this case) small,

cosmic radiation is given a smaller target in time. In general, the type system can help the compiler enforce live range constraints to reduce the window of vulnerability. Similar constraints can be placed on any registers deemed to carry sensitive information.

With hardware support, a different approach can be taken with the window of vulnerability. With ISA support for special loads and stores, the window of vulnerability can be eliminated without great hardware expense [57]. Redundant load and store instructions can be supported by pairing them into a single transaction before interacting with the memory system to avoid unwanted side-effects and other problems. Full coverage of architectural state is achieved with ECC covering the memory system and the load/store queue, and the redundant load/store with redundant registers covering the pipeline and register files.

In general, the cost of redundancy can be reduced if a good balance of responsibility is struck between hardware and software. In the proposed work, we will not limit ourselves to software-only solutions. For example, we may find that creating redundancy in programs by duplicating instructions in the code is not the most efficient means. Compiler decisions about which code segments or registers to make redundant could be passed to the hardware through a side table. The resulting software-modulated, hardware-implemented fault tolerance technique may have desirable properties such as good backwards compatibility (simply ignore the fault tolerance side-table). As another example, we may find that only minor extensions to existing ISAs are necessary for large gains. New instructions designed to check the value in multiple corresponding registers would reduce the performance cost of the checking branches.

While we do intend to explore new SMFT techniques involving hardware, special emphasis will be placed on new software-modulated, software-implemented fault tolerance techniques. For example, since future generations of many microchips will contain multiple processor cores, we will explore opportunities created by having multiple threads available for software-only SMFT.

### **3.5 Hardware Design for Software-Modulated Fault Tolerance**

Transistors in today's microprocessors are sized according to design rules established to ensure that the chips produced work properly. Since even one bad transistor can make the chip unusable, these design rules are set with a margin of safety to ensure a high yield. As transistors continue to shrink in size, they will become more susceptible, and transient fault rates will rise. One way to counter this trend is to oversize some, but not all transistors, resulting in chips with components that vary in terms of their dependability.

We believe, and our previous research experience supports the fact, that non-uniform policies on transistor reliability can have a substantial impact on processor design and performance since not all transistors have the same importance. For example, as PIs Martonosi and Clark demonstrated, designing weaker memory cells in the branch predictor may be beneficial in terms of cost (size of each cell) and power savings even if these cells forget their value over time [28]. We intend to apply similar reasoning and implementation techniques as we study optimizations that trade reliability in these components for lower cost and power consumption.

We also believe we will be able to make gains by carefully navigating the hardware-software divide. Transistors that affect architectural state must normally operate without fault for correctness. However, in the presence of software-modulated fault tolerance (SMFT), these transistors fall into one of two categories. First, there are those transistors which can be protected by SMFT and those which cannot. While those which cannot be protected must be made robust, the others may be designed with a smaller margin of safety to lower power consumption and reduce cost. While we can only speculate at this point about the full range of possibilities for combining hardware and software to detect and recover from faults, our multi-disciplinary team has the skills to be able to identify opportunities that arise in the early years of our proposal as we study the effectiveness and performance of software techniques and the knowledge to implement and evaluate new solutions in the latter years. In order to facilitate hardware design in this space, we propose to develop design tools that extend PI August's Liberty Simulation Environment [70] specifically for fault tolerance and reliability. In particular, we will extend Liberty's type systems for hardware design with reliability constraints, based on principles of information flow closely connected to our compiler typed intermediate languages (see section 3.1). These tools will help designers determine the level to which interconnected microarchitectural components must be protected. Moreover, such tools are an important research contribution in and of themselves as we will make them freely available to other researchers in this area.

Research Task	Start	Finish	Notes
Fault-tolerant Typed Intermediate Language (TIL/FT)			
Theory of TIL/FT	-	Year 1	Work in progress
Practical TIL/FT design and implementation	Year 1	Year 2	Section 3.2
Type-preserving compiler implementation	Year 2	Year 3	Section 3.2
TIL/FT evolution (incorporate new software/hardware FT)	Year 3	Year 4	Section 3.2
Fault-tolerant Typed Assembly Language (TAL/FT)			
Theory of TAL/FT (software-only FT)	Year 1	Year 2	Section 3.2
Practical design and implementation of TAL/FT	Year 2	Year 3	Section 3.2
TAL/FT evolution (incorporate new software/hardware FT)	Year 3	Year 4	Section 3.2
Reliability Specifications			
Source-level theory of reliability	Year 2	Year 3	Section 3.1
Implement reliability directives; extend TIL/FT	Year 2	Year 3	Section 3.1
Implement reliability directives; extend TAL/FT	Year 3	Year 4	Section 3.1
Evaluation and evolution	Year 3	Year 4	Section 3.1
Reliability-preserving optimization			
Optimization theory	Year 1	Year 4	Section 3.3
Design, implement, preserve typing, evaluate	Year 2	Year 4	Section 3.3
Extend TIL/FT, TAL/FT typing as necessary	Year 2	Year 4	Section 3.3
Novel, reliability-specific optimizations	Year 3	Year 4	Section 3.3
New instruction-level software-modulated fault tolerance techniques			
Implement a set of SMFT techniques	-	Year 2	Work in progress
Develop hybrid/hardware SMFT techniques	Year 2	Year 3	Section 3.4
Hardware optimization			
Extend Liberty design language and types for reliability	Year 1	Year 3	Section 3.5
Non-uniform hardware optimization	Year 1	Year 4	Section 3.5

Table 1: Timeline for investigation and completion of research tasks.

## 4 Timeline

The broad scope and ambitious goals of our proposal will require substantial commitment and planning to ensure success. Table 1 outlines the timeline on which the various sub-tasks will be carried out. Many of the research components will take place in parallel, though there are some dependencies. For instance, we plan first to focus on developing the central typed intermediate languages that form the core of our compiler and to compile assuming a uniform reliability policy. From that point, in parallel, we will develop optimizations over the intermediate representation, a typed, fault-tolerant assembly language, and mechanisms for source-level reliability specifications. All the while, we will concurrently study hardware optimization strategies, new instruction-level fault-tolerance techniques and new opportunities to shift the reliability back and forth between hardware and software to optimize cost and flexibility. Each of the latter activities will inform the former activities on compiler design. All of software implementation work will exploit the existing (untyped) implementation infrastructure built by David August (Co-PI) [56, 57], thereby greatly improving our productivity. Our research on new instruction-level, software-modulated fault tolerance techniques will exploit David August’s (Co-PI) fault-injection infrastructure and simulators [56, 57].

## 5 Existing Approaches

Over the years, many techniques using double or triple hardware redundancy have been proposed to detect or recover from transient faults [11, 26, 31, 67, 80, 81]. While these heavyweight techniques might have their place in mission-critical applications, they are completely infeasible for commodity systems.

To ease the hardware expense of redundancy, techniques have been proposed which utilize resources already found on most processors such as simultaneous multithreading [55, 59, 60, 74] and chip multiprocessors [23, 41]. These techniques, when applied universally, typically claim many resources that could otherwise be applied to other work, thus degrading performance substantially.

Others [51, 53] have proposed making small modifications to the processor core to implement redundancy. Again, these techniques present an insupportable burden on processor performance. Furthermore, adding even more complexity to already baroque out-of-order execution structures is likely to push design complexity beyond the bounds of feasibility.

Software techniques are attractive from both the chip designers' and the consumers' perspectives because they can provide high fault coverage with *zero* hardware cost. Several software-only fault-detection methods have been proposed for memory [64], control flow [47, 50, 73], and for redundant execution [15, 25, 46, 48, 49, 54]. However, software-only techniques typically incur a performance penalty too large to be practical.

Only our work has begun to explore the capabilities of software modulation [58]. All other works have been applied uniformly across the entire system, whereas our work focuses on applying custom levels of protection across different portions of the system.

**Existing type-preserving compilation techniques.** The process of propagating typing information from source language down through compiler intermediate and target languages is known as *type-preserving compilation*. Java compilers were amongst the first to exploit this new technology. However, they only propagate typing information a very short way through compilation, down to the level of the Java virtual machine language. Moreover, the central motivation for development of the typed JVM was *security*, not reliability. The goal was to allow code consumers, such as Web browsers, to download, type check and securely execute potentially malicious applets supplied by unknown and trusted sources.

In the mid-nineties, at approximately the same time as Java was being developed, researchers at Carnegie Mellon first suggested repeatedly type checking compiler intermediate languages for the purpose of improving compiler reliability, as well as security, and developed the TIL [68, 35]. During this exciting period, many other researchers were also developing new, flexible type systems for their optimizing compilers and intermediate languages [61, 22, 18], including Co-PI Appel and his students [62].

The next critical development came when Walker (PI) and his collaborators were the first to develop not just typed compiler intermediate languages, but a Typed Assembly Language (TAL) [39, 40, 36, 38]. This innovation required new techniques for pushing typing information through all optimization phases and the entire code generation process. The major impact was that TAL could serve as a verifiable *compiler output*. Since type checking occurred after compilation was complete, the typechecked code was the code that actually ran on user machines. Consequently, provided the type checker was implemented correctly (software an order of magnitude smaller and more reliable than the compiler itself), there could be no bugs that compromised safety or security. While TIL and TAL both advocated using types to improve compiler reliability, they both assume perfect hardware and provide no guarantees in the presence of transient faults.

Proof-carrying code (PCC) [44, 43] is a close relative of TAL and strives to meet many of the same goals. The original PCC system used Hoare logic, adapted for machine code, to verify the safety of programs. In principle, PCC can be used to verify any program property, but in practice, there must be some way to generate proofs automatically, and the only robust, general and scalable mechanism for generating proofs of program properties is type-preserving compilation. Hence, like TIL and TAL, practical PCC systems use type-preserving compilation, and in fact, encode type systems quite directly in their underlying logics. Since the initial development of PCC and TAL, the distinctions between the two have become increasingly blurred as researchers have developed more and more flexible verification systems that explicitly combine types, logic and proofs [19, 79, 63, 17, 1, 27, 33]. Nevertheless, like TIL and TAL, all of these systems assume perfect hardware and hence none are correct in the presence of transient faults.

The most recent major development in type-preserving compilation is Appel's (Co-PI) Foundational Proof-Carrying Code [4]. Using type-directed compilation reduces the trusted computing base (TCB), that fragment of the software whose correctness is sufficient to guarantee system correctness, from 100s of thousands of lines of code to 15 to 30 thousand lines of code. Using type-directed compilation with Foundational Proof-Carrying Code reduces the trusted computing base to its barest minimum, a mere 2-4 thousand lines, which could be verified by hand. However, like all the other approaches we have discussed, FPCC assumes perfect hardware, and does not guarantee correctness in the presence of transient faults.

## 6 Broader Impact

**Summer Schools on Security and Reliability.** For the last two years Walker (PI) has organized summer schools on advanced topics in security and reliability, held at the University of Oregon. [7, 8] This year he is on the steering committee planning yet another summer school to be held in the same location. [6] These summer schools provide a unique opportunity – there is nothing else like it in North America – for students to come together for 10 days to study an important topic or collection of topics in this broad area. The schools have seen between 40 and 60 students ranging from a brilliant and unique high school student we had last year to undergraduate students, graduate students, post doctoral fellows, industrial researchers and university faculty. The majority are graduate students in their first or second year. We have also recruited students from underrepresented groups including women, underrepresented minorities and the disabled.

To summer school program each year has included 10 of the foremost lecturers in their field to teach the Summer School students the foundations they need to begin research in the area. Former lecturers have included Martin Abadi, Tom Ball, Ed Felten, Kathleen Fisher, Robert Harper, Peter Lee, Catherine Meadows, Francois Pottier, and many other top researchers. Each lecturer has given between two and four lectures, 80 minutes long on average, on their topic. At all times, material was presented at a tutorial level with the goal of helping graduate students and researchers from academia or industry understand the critical issues and open problems confronting the field.

The PI proposes to continue his leadership role in educational outreach by planning new summer schools in the upcoming years and complementing funds received from NSF by looking for matching funds from other institutions and industry. In the past, funding of the summer schools has been through NSF and a variety of other sources obtained by the PI and his collaborators including ACM SIGPLAN, INRIA (France), Microsoft, and Intel. One of the next summer schools will be on cross-disciplinary topics relevant to this grant: Security and Reliability at the Hardware-Software boundary.

**Integrating Research and Education at Princeton.** Princeton has a strong tradition of integrating research and education. At the graduate level, each of the PIs mentors a number of graduate students who will be supported by funds from this grant. Both first-year graduate courses and more advanced seminars will incorporate elements of the research and related areas directly into the coursework.

At the undergraduate level, the PIs will seek opportunities to include elements of their research into the classroom in their computer science and electrical engineering courses on programming, programming languages, compilers, computer organization, and microprocessor design. Princeton also has a very active undergraduate research program with many students doing multiple junior year and senior year independent research projects. In total, the PIs have mentored approximately 68 undergraduate student research semesters of research over the past 5 years. Last year, Walker's advisee, Rob Simmons, won the Princeton Computer Science Department undergraduate thesis award. Many of the PIs students, excited by their taste of research at Princeton, have gone on to graduate school at other top institutions around the country.

**Participation of Underrepresented Groups.** While computer engineering research areas have difficult time developing relationships with students of minority-represented groups, other science related fields encourage such students by having concrete examples of challenges in their fields. In general, aspects of computer design and programming have specific barriers that prevent an individual's creativity to prosper since often the specific problem is not presentable within the first years of engineering education. Simply stated, the concepts of computer design are not hands-on and not easily approachable. This proposal will support efforts by the PI to change this situation, to bring the fascinating aspects of computer science and design to the masses through a variety of means, potentially including a television program.

The proposal also includes efforts in recruiting under-represented groups of students to the computer science field by working with several university organizations. The PI actively participates as a member of the Princeton University Department of Computer Science Diversity Taskforce which looks for ways to address diversity issues at the department, school, university, state, and national levels. The taskforce does this in part by working with other groups, such as the Society of Women in Engineering (SWE) and the National Society of Black Engineers (NSBE). Efforts involve reaching potential engineers and scientists in high schools through Princeton's High School Honors program.

**Benefits to Society.** The proposed work will have an immense impact on society by simultaneously improving the productivity and reliability of general purpose and embedded applications. New ideas and tools will foster new ideas and methods designed to handle the diminishing reliability of the substrate. In addition to publications generated and tools released, dissemination will include technology transfer using existing ties with the microprocessor industry

(including Intel, IBM, AMD, Sun Microsystems, HP, Microsoft, Xilinx, Freescale, Motorola). We will work closely with our industry affiliates to ensure practical constraints are handled, easing any future technology transfer.

## 7 Results of Prior Support

**Foundational Proof-Carrying Code** Recent work at Princeton involving PIs Appel and Walker, supported in part by NSF Grant CCR-0208601: “Collaborative Research: High-Assurance Common-Language Runtime,” develops sound type systems for machine-level programs and trustworthy implementations of them via the idea of Foundational Proof-Carrying Code (FPCC).

One of the most challenging aspects of developing a safe type system at the assembly language level is safely reasoning about memory allocation, deallocation, pointers and aliasing – this is where low-level type systems truly diverge from their higher-level counterparts. To deal with these complexities, Walker extended his earlier work on typed assembly language (TAL) [40, 38] with rich new substructural logics. These substructural logics can encode sophisticated properties of memory that allow for safe stack-allocated data [2] and heap or region-allocated data [1]. In addition to studying memory safety properties, Walker has shown how to use related type-theoretic techniques to enforce adherence to very general software protocols [32]. He has also written a chapter of a new textbook on type systems [75].

Both TAL and Necula’s PCC [44] eliminate the compiler—hundreds of thousands of lines of code—from the trusted base of the safety mechanism; but each of these systems still has a fairly complex trusted checker (thirty or forty thousand lines of code for Necula’s PCC). There is no formal proof of the safety of the *entirety* of Necula’s PCC system nor Walker and Morrisett’s TAL type checker. In fact, there is not even a complete formal specification of the safety property being checked! Appel’s FPCC project has rectified this problem by specifying a nontrivial safety property for real machine code, and guaranteeing that machine-language programs respect the property using the *smallest possible* trusted computing base. Moreover, he has shown it is possible to do so *without any extra work by the application programmer*. Both Appel and Walker’s experience defining type and proof systems at the machine level will be invaluable when pursuing the research in the current proposal.

**Secure and Reliable Programming Languages** The goal of PI Walker’s CAREER award (NSF CCR-0238328 CAREER: Programming Languages for Secure and Reliable Component Software Systems) is to develop new programming language technology that will improve the security and reliability of component software systems. More specifically, Walker and his students have begun to develop new techniques for defining, implementing and reasoning about run-time enforcement of program properties. For example, he developed a rich new theory of security monitors as formal automata that transform untrusted program behavior as they execute [29]. This theory allows security architects to model a variety of different sorts of run-time enforcement mechanisms, to prove that certain mechanisms can or cannot enforce various security properties, and to compare the power of different classes of security monitors. Recently, Walker has used the theory to prove the surprising new result that powerful run-time program monitors can enforce certain kinds of liveness properties [30]. In addition, he has studied the implementation of program monitors using well-defined, type-safe aspect-oriented extensions of Java [12] and ML [76, 20, 21].

**Compiler Optimization** PI August’s CAREER award CCR-0133712 (*Systematic Design Space Exploration*) focused in part on the development of the IMPACT compiler. IMPACT is an aggressively optimizing ILP research compiler currently widely used in both academia and industry. IMPACT was released under the direction of Wen-mei Hwu and John Gyllenhaal with NSF support (CCR-9809494 *Sharing the IMPACT ILP Compiler Technology with US Researchers*). While a graduate student at the University of Illinois, PI August was funded in part by NSF grant CCR-9308013 under the direction of Wen-mei Hwu. August’s work in predication and predicate analysis using IMPACT is well documented [9, 10, 66]. August and his students have already implemented several fault detection schemes in the IMPACT compiler. This reliability research received the *Third International Symposium on Code Generation and Optimization Best Paper Award* [56]. This work led to several related compiler-based fault detection and recovery systems [58, 57, 16]. The compiler reliability optimizations developed were made freely available. In response, a number of outside research and industry groups, including Intel, the University of Texas, the University of Illinois, North Carolina State University, and the University of Florida are using this framework for reliability research.

**Architectural Fault Modeling** The proposed work will require microarchitectural modeling of future computer systems. PI August's CAREER award CCR-0133712 (*Systematic Design Space Exploration*) involved the initial development of the Liberty Simulation Environment. The additional support of CNS-0305617 *Structural and Composable Performance Simulation of Complex Systems* facilitated the release of LSE to the public. Versions of LSE have been used at Princeton University, Rice University, University of Texas at Austin, Carnegie Melon University, University of Illinois at Urbana-Champaign, University of California at Berkeley, University of California at Los Angeles, Intel Corporation, and Infineon Corporation. LSE was downloaded over 1000 times in 2005, and new international modeling standards are being based on LSE technology. In areas including concurrent high-level structural specifications, front-end analysis and compilation of such specifications, the results of this work are well documented [71, 69, 72, 70].

LSE also provides part of a new high-speed fault injection framework [57]. To experimentally test the reliability of a program with partial redundancy, faults are injected. Timing simulation from LSE used with a fault distribution model determines a set of architectural-level changes necessary to model the program-level effects of each fault. Used in conjunction with native hardware, microarchitectural fault modeling can be performed at near native speeds. For the first time, this system allows researchers to inject the thousands of faults necessary to achieve high confidence in their system reliability measurements.

**Microarchitectural and Circuit-level Work** While the prior work by PIs Martonosi and Clark has focused on power efficiency, observations made during this work apply directly to managing unreliable components. Prior funding (NSF CNS-0410937: Adaptive, Power-Efficient Processors for Sensors and Embedded Systems) supports research in real-time power management in microprocessors and sensor networks. This work focused on techniques for microprocessors, mainly on microarchitectural methods that monitored the synchronization queues that lie between independently-clocked domains in a so-called "GALS" processor (Globally Asynchronous, Locally Synchronous). The novelty of their approach was to apply formal control-theoretic techniques to this problem. (Previous approaches had been essentially ad hoc.) They designed, analyzed and simulated an efficient controller that managed the frequency setting of the clocked region at the receiving end of the queue, and showed how the desired queue fullness (an input parameter of the controller) expressed the trade-off between performance and energy savings. Simulations of a raft of benchmarks showed excellent results (significant power savings with barely diminished performance) [77]. Extensions to this work enable responding very quickly to workload behavior changes and enable applications to multicore networks [78].

With other prior support (Designing "Real-Power" Systems: Static and Dynamic Techniques for Managing Power/Performance Tradeoffs. NSF ITR CCR-0086031), PIs Martonosi and Clark developed power-adaptive computer systems. The core idea of this work is to recognize that in many domains, the central problem is not to minimize power consumption at all costs, but rather to create systems that responsively trade-off performance and power in ways that improve performance while keeping to the system's power budget. The experiences garnered through this research will be directly useful to us as we develop resource management layers and policies for the devices and networks discussed here.

## References

- [1] Amal Ahmed, Limin Jia, and David Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science*, pages 33–44, Ottawa, Canada, June 2003.
- [2] Amal Ahmed and David Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New Orleans, January 2003.
- [3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [4] Andrew W. Appel. Foundational proof-carrying code. In *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE, 2001.
- [5] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science*.
- [6] Zena Ariola, Jeff Foster, Dan Grossman, David Walker, and Steve Zdancewic (Summer School Steering Committee). Summer school on language-based techniques for concurrent and distributed software, July 2006. <http://www.cs.uoregon.edu/activities/summerschool/summer06/>.
- [7] Zena Ariola, David Walker, and Steve Zdancewic (Summer School Organizers). Summer school on software security: Theory to practice, June 2004. <http://www.cs.uoregon.edu/activities/summerschool/summer04/>.
- [8] Zena Ariola, David Walker, and Steve Zdancewic (Summer School Organizers). Summer school on reliable computing, July 2005. <http://www.cs.uoregon.edu/activities/summerschool/summer05/>.
- [9] D. I. August. *Systematic Compilation for Predicated Execution*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 2000.
- [10] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.
- [11] Todd 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.
- [12] Lujo Bauer, Jarred Ligatti, and David Walker. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming language design and implementation*, pages 305–314, June 2005.
- [13] Robert 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.
- [14] Robert 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.
- [15] 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.
- [16] Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery methods. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, June 2006.
- [17] Karl Cray and Joseph C. Vanderwaart. An expressive, scalable type theory for certified code. In *ACM SIGPLAN International Conference on Functional Programming*, September 2002.
- [18] Karl Cray and Stephanie Weirich. Flexible type analysis. In *ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

- [19] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston, January 2000.
- [20] Daniel S. Dantas and David Walker. Harmless advice. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2006.
- [21] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In *ACM SIGPLA International Conference on Functional Programming*, September 2005.
- [22] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *ACM International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.
- [23] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109. ACM Press, 2003.
- [24] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, May 2003.
- [25] John G. Holm and Prithviraj 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.
- [26] 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.
- [27] Limin Jia, Frances Spalding, David Walker, and Neal Glew. Certifying compilation for a language with stack allocation. In *IEEE Symposium on Logic in Computer Science*, pages 407–416, June 2005.
- [28] Philo Juang, Kevin Skadron, Margaret Martonosi, Zhigang Hu, Douglas W. Clark, Philip W. Diodato, and Stefanos Kaxiras. Implementing branch predictor decay using quasi-static memory cells. *ACM Transactions on Architecture and Code Optimization*, 1(2):180–219, 2004.
- [29] Jarred Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *International journal of information security*, 2004.
- [30] Jarred Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Tenth European Symposium on Research in Computer Security*, September 2005.
- [31] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors—a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.
- [32] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *ACM SIGPLAN International Conference on Functional Programming*, August 2003.
- [33] Vijay S Menon, Neal Glew, Brian R Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, January 2006.
- [34] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in los alamos national laboratory’s ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.
- [35] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.
- [36] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

- [37] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28–52. Springer-Verlag, 1998.
- [38] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based Typed Assembly Language. *Journal of Functional Programming*, 12(1):43–88, January 2002.
- [39] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998.
- [40] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.
- [41] Shubhendu S. Mukherjee, Michael Kontz, and Steven 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.
- [42] Andrew Myers and Barbara Liskov. Jflow: Practical mostly-static information flow control. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 226–241, January 1998.
- [43] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.
- [44] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, October 1996.
- [45] 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.
- [46] Nahmsuk Oh and Edward J. McCluskey. Low energy error detection technique using procedure call duplication. In *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*, 2001.
- [47] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. In *IEEE Transactions on Reliability*, volume 51, pages 111–122, March 2002.
- [48] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. ED<sup>4</sup>I: Error detection by diverse data and duplicated instructions. In *IEEE Transactions on Computers*, volume 51, pages 180 – 199, February 2002.
- [49] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in superscalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.
- [50] Joakim Ohlsson and Marcus Rimen. Implicit signature checking. In *International Conference on Fault-Tolerant Computing*, June 1995.
- [51] Janak H. Patel and Leona Y. Fung. Concurrent error detection in alu’s by recomputing with shifted operands. *IEEE Transactions on Computers*, 31(7):589–595, 1982.
- [52] Francois Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
- [53] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 214–224. IEEE Computer Society, 2001.
- [54] Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Marco Torchiano. A source-to-source compiler for generating dependable software. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.

- [55] Steven K. Reinhardt and Shubhendu 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.
- [56] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.
- [57] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu 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.
- [58] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. In *ACM Transactions on Architecture and Code Optimization (TACO)*, volume 2, December 2005.
- [59] Eric 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.
- [60] 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.
- [61] 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.
- [62] 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.
- [63] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *ACM Symposium on Principles of Programming Languages*, London, January 2002. ACM Press.
- [64] Philip P. Shirvani, N.R. Saxena, and Edward J. McCluskey. Software-implemented EDAC protection against SEUs. In *IEEE Transactions on Reliability*, volume 49, pages 273–284, 2000.
- [65] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo 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.
- [66] J. W. Sias, W. W. Hwu, and D. I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proceedings of 33rd Annual International Symposium on Microarchitecture*, pages 112–123, December 2000.
- [67] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM’s S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.
- [68] 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.
- [69] Manish Vachharajani, Neil Vachharajani, and David I. August. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.
- [70] Manish Vachharajani, Neil Vachharajani, Jason A. Blome and Sharad Malik David A. Penry, and David I. August. The liberty simulation environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems*.

- [71] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.
- [72] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture*, December 2004.
- [73] 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.
- [74] T. N. Vijaykumar, Irith Pomeranz, and Karl 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.
- [75] David Walker. *Substructural type systems*, chapter 1. MIT Press, January 2005. In *Advanced topics in types and programming languages*, Benjamin Pierce, ed.
- [76] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ACM International Conference on Functional Programming*, Uppsala, Sweden, August 2003.
- [77] Qiang Wu, Philo Juang, Margeret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
- [78] Qiang Wu, Philo Juang, Margeret Martonosi, and Douglas W. Clark. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, 2005.
- [79] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *International conference on functional programming*, Florence, September 2001.
- [80] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.
- [81] Y.C. 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.
- [82] James. F. Ziegler and Helmut Puchner. *SER - History, Trends, and Challenges: A Guide for Designing with Memory ICs*. 2004.