VERIFYING APPLICATION-SPECIFIC FAULT TOLERANCE VIA FIRST-CLASS FAULT MODELS

By Michael Carbin, Jamieson Career Development Assistant Professor of Electrical Engineering and Computer Science; Member, Computer Science and Artificial Intelligence Laboratory (CSAIL)

Due to the aggressive scaling of technology sizes in modern computer processor fabrication, modern processors have become more vulnerable to errors that result from natural variations in processor manufacturing, natural variations in transistor reliability as processors physically age over time, and natural variations in these processors' operating environments (e.g., temperature variation and cosmic/environmental radiation).

Large distributed systems composed of these processors — such as emerging designs for exascale supercomputers — are anticipated to encounter errors frequently enough that traditional techniques for building high-reliability applications will be too resource-intensive (in terms of time, storage, and energy consumption) to be practical. Applications will, instead, need to be architected to *execute through errors*[1].

Traditional Fault Tolerance

Researchers have long sought methods to enable reliable computation on unreliable computing substrates. For example, in the 1950s, vacuum-tube-based computing systems experienced vacuum-tube failures as frequently as every 8 hours[2]. In response, the industrial and academic community sought to resolve this issue both by designing more reliable computing substrates (modern CMOS transistors) and by designing fault-tolerance mechanisms. Of these latter techniques, popular methods include:

1) n-modular redundancy to implement majority voting (where $n > 2$),

2) dual-modular (2-modular) redundancy (DMR) to enable error detection and subsequent restart, and

3) algorithm-based fault-tolerance methods to provide low-overhead application-specific detection and correct schemes.

A major aspect of the design of such mechanisms is the trade-off between the overhead (in performance, memory consumption, and energy consumption) of these techniques, the frequency and distribution of hardware faults, and the coverage of a specific error-detection and correction scheme. For example, standard methods for DMR duplicate the entire execution of a computation and check if the two executions of the computation agree on their results. This technique introduces significant computational and energy overhead.

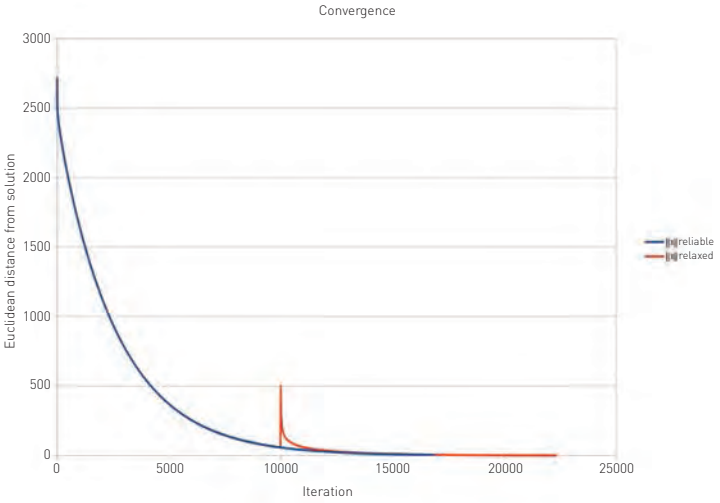

Figure 1: Reliable versus Unreliable Execution of Jacobi Iterative Method for Oil

In contrast, algorithm-based fault-tolerance techniques — such as those for linear algebra — produce lightweight checksums that can be used to validate whether the computation produced the correct results. For some applications, these checksums are exact, enabling the exact error detection of capabilities of DMR with lower overhead. However, for other applications, these checksums either are not known to exist or, at best, compromise on their error coverage.

Application-Specific Fault Tolerance

Modern large computing systems have begun to operate a point in the trade-off space between performance/energy and error rates that traditional, application-oblivious fault-tolerance techniques are too resource-intensive to deploy at scale for large numerical computation. In response, researchers have begun to expand on historical results for algorithm-based fault tolerance — alternatively, application-specific fault tolerance — to identify new opportunities for low-overhead mechanisms that can steer an application's execution to produce acceptable results: that is, results that are within some tolerance of the result expected from a fully reliable execution.

Such techniques include selective n-modular redundancy in which a developer either manually or with the support of a fault-injection tool identifies instructions or regions of code that do not need to be protected for the application to produce an acceptable result, as determined by an empirical evaluation. Another class of techniques are fault-tolerant algorithms that, through the addition of algorithm-specific checking and correction code, are tolerant to faults.

For example, Figure 1 shows the results of executing a self-stabilizing iterative solver (Jacobi iterative method) for a system of linear equations that corresponds to an oil reservoir simulation problem. The graph presents the norm of the error of the solution vector (y-axis) as a function of the number of iterations (x-axis) for a reliable execution (in blue) and an unreliable execution (in red) that encounters a fault on iteration 10,000. The unreliable execution eventually converges to the correct solution. Moreover, it recovers quickly relative to the algorithm's overall convergence from a solution vector of similar error (as evidenced by previous iterations).

While these techniques and algorithms are promising, a major barrier to implementing either of these techniques is that their results either rely on empirical guarantees or — for algorithm-specific techniques — hinge on the assumption that the fault model of the underlying computing substrate matches the modeling assumptions of the algorithmic formalization.

Verifying Application-Specific Fault Tolerance with Leto

In our recent research, Brett Boston and I have developed Leto[3], a verification system that supports reasoning about unreliably executed programs. Leto enables a developer to build confidence in an application-specific fault-tolerance mechanism by 1) enabling a developer to programmatically specify the behavior of the computing substrate's fault model and 2) enabling a developer to verify relational assertions that relate the behavior of the unreliably executed program to that of a reliable execution.

Leto enables a developer to specify the behavior of the underlying hardware system as a program that Leto automatically weaves into the execution of the main program. In addition, Leto enables a developer to specify relational assertions that, for example, constrain the difference in results of the unreliable execution of the program from that of its reliable execution.

First-Class Fault Models: Leto enables a developer to programmatically specify a stateful fault model. For example, a common fault model that application developers use is the single-event-upset model. In this model, at most one fault can occur during the execution of the program. While simple, this model can capture real fault models in which it is possible for errors to happen during execution, but with small probability.

Figure 2 presents a specification in Leto of a single-event upset model that affects only addition operations. This model includes a boolean valued state variable that records whether a fault has occurred during the execution of the program. The model then exports two versions of the addition operator. Line 4 specifies the reliable implementation of addition, while Line 6 specifies an unreliable version. For each addition operation

```
model {
   bool upset = false;

   operator +(x1, x2) modifies () ensures (result == x1 + x2);

   operator +(x1, x2) when (!upset) modifies (upset)
      ensures (x1 + x2 - E < result < x1 + x2 + E && upset == true);
}
```

Figure 2: Leto Fault Model Specification

in a program, Leto dynamically makes a non-deterministic choice between the set of operation implementations that are currently *enabled* in the model as indicated by their *guards* evaluating to *true*. Namely, an operation's guard is the optionally specified boolean expression that occurs after the **when** keyword. For these two versions of addition, the reliable version is always enabled and the unreliable version is enabled only if upset — indicating that a fault has yet to occur in the program.

Relational Verification: Verifying an application that has been protected with an application-specific fault-tolerance mechanism typically requires reasoning about two types of properties of the resulting application: *safety properties* and *accuracy properties*.

Safety properties are standard properties of the execution of the application that must be true of a single execution of the application. Such properties include, for example, memory safety and the assertion that the application returns results that are within some range. For example, a computation that computes a distance metric must, regardless of the accuracy of its results, return a value that is non-negative. In Leto, a developer specifies safety properties with the standard assertion statement, **assert e**, as typically seen in verification systems. For example, to assert that the result of a distance metric is non-negative, a developer may write in the program the statement **assert 0 <= x**, where **x** is the result of the metric.

Accuracy properties are properties of the unreliable or *relaxed* execution of the application that relate its behavior and results to that of a reliably executed version. Accuracy properties are *relational* in that they relate values of the state of the program between its two semantic interpretations. For example, the assertion **-epsilon < x<o> - x<r> < epsilon** in Leto specifies that the difference in value of **x** between the program's reliable execution (denoted by **x<o>**) and relaxed execution (denoted by **x<r>**) is at most epsilon.

Key Insights: Leto relies on an Asymmetric Relational Hoare Logic[4] as its core program logic. Relational Hoare Logics are a variant of the standard Hoare Logic that natively refer to the values of variables between two executions of the program. Leto's use of a relational program logic serves two goals: 1) it gives a semantics to accuracy properties and 2) it enables tractable verification of safety properties.

As an example of the latter, proving the memory safety of an application outright can be challenging for many applications. However, application-specific fault-tolerance mechanisms can typically be designed and deployed such that it is possible to verify that for any given array access or memory access, errors in the application do not *interfere* with the accessed address. Such properties are typically easier to verify for a protected

application than verifying the safety of the memory access outright.

Leto therefore enables developers to tractably verify a strong *relative* safety guarantee: if the original application satisfies all of the specified safety properties, then relaxed executions of the application with its deployed application-specific fault-tolerance mechanisms also satisfy these safety properties.

Results: We have used Leto to verify key correctness properties on several self-stabilizing algorithms: Jacobi, Self-stabilizing Conjugate Gradient, and Self-stabilizing Steepest Descent. Our results show that is possible to verify the key invariants required to prove that these algorithms' self-stability guarantees hold for their implementations. In general, Leto enables developers to specify and verify the rich fault-aware properties seen in applications with application-specific fault-tolerance mechanisms.

Future Approximate Computing Systems

As we continue to scale the size of our systems to include larger collections of increasingly less reliable components, our methods for architecting software systems will need scalable and verifiable methods to manage the uncertainty in the underlying execution substrates of the systems. Leto, along with other work in my Programming Systems Group[5, 6, 7], directly provides new programming languages, methodologies, and systems that enable developers to reason about unreliable, continuous, and probabilistic computation.

References

[1] S. Amarasinghe, et al. ExaScale Software Study: Software Challenges in Extreme Scale Systems. DARPA IPTO, Air Force Research Labs, Technical Report (2009).

[2] J. Von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. Automata Studies, 35 (1956).

[3] B. Boston and M. Carbin. Verifying Application-Specific Fault Tolerance via First-Class Models. In submission.

[4] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving Acceptability Properties of Relaxed Nondeterminisic Approximate Programs. PLDI (2012).

[5] E. Atkinson and M. Carbin. Towards Correct-by-Construction Probabilistic Inference. NIPS Workshop on Machine Learning Systems (2016).

[6] E. Atkinson and M. Carbin. Towards Typesafety to Explicitly-Coded Probabilistic Inference Procedures. In submission.

[7] B. Sherman, L. Sciarappa, M. Carbin, and A. Chlipala. Overlapping Pattern Matching for Programming with Continuous Functions. In preparation.