Michael Carbin *Research Statement*

My research focuses on developing programming systems (programming languages, compilers, and runtime systems) that deliver improved performance or resilience by changing the underlying semantics of the program. Although this approach violates the traditional contract that the programming system must preserve the program's semantics, my research demonstrates that giving the programming system the freedom to change the semantics enables new methods for improved performance and resilience that are still sound and principled.

Motivation and Overview

Improving program performance and resilience are long-standing goals. Traditional approaches include a variety of transformation, compilation, and runtime techniques that share the common property that all executions of the resulting program are still legal executions of the original program. However, my research demonstrates that giving programming systems the freedom to change the semantics of the program can open up new and otherwise unavailable opportunities:

- **Performance.** I have worked on a number of techniques that trade the quality of an application's results for increased performance. The experimental results show that aggressive techniques – for example transforming loops to skip some (or all) iterations – can yield up to a four-fold improvement in an application's performance with acceptable changes in the quality of its results [Hoffmann et al., ASPLOS, 2011; Carbin et al., PEPM, 2013].
- **Resilience.** I have worked on a number of techniques that modify the semantics of the program to make it more resilient to programming errors [Perkins et al., SOSP, 2009; Long et al., ICSE, 2012]. The experimental results show that if the program encounters an error that threatens its stability or security, these techniques enable the runtime system to modify the program's execution to steer around the problem and still execute acceptably. For example, it is often possible to enable an application that is stuck in an infinite loop to produce an acceptable output by simply exiting the loop and continuing on with the remaining execution of the program [Carbin et al., ECOOP, 2011; Kling et al., OOPSLA, 2012].

Changing the semantics of programs raises a number of new and fundamental questions. For example, what is the probability that the resulting program will produce the same result as the original program? How much do the results differ from those produced by the original program? And is the resulting program safe and secure?

My research provides the first programming models and reasoning systems for answering these questions. My work on *quantitative reliability* makes it possible to reason about the probability that a program will produce the same result after some of its operations have been replaced with alternate versions that may produce different results with some probability [Carbin et al., OOPSLA, (Best Paper Award), 2013]. My work on *relaxed programs* makes it possible to effectively verify a program's safety and accuracy after its semantics has been broadened to enable additional executions [Carbin et al., PLDI, 2012].

Taken together, my work demonstrates that giving programming systems the freedom to change the semantics of the program can open up new, simple, and effective ways to boost performance and/or enhance resilience and still verify that the resulting behavior is acceptable.

Quantitative Reliability

Hardware architectures have traditionally provided a fully reliable digital abstraction. However, as software becomes increasingly dominated by approximate computing applications that have a natural resilience to noise and inaccuracies (e.g., multimedia processing, machine learning, and big data analytics), this reliable abstraction may no longer be necessary for all components of an application. Motivated in part by this observation, the computer architecture community has begun to investigate new designs that improve performance by breaking this reliable abstraction. The goal is to reduce the cost of implementing a reliable abstraction on top of physical materials and manufacturing methods that are inherently unreliable. For example, researchers are investigating designs that incorporate aggressive device and voltage scaling techniques to provide low-power ALUs and memories. A key aspect of these components is that they forgo traditional correctness checks and instead expose timing errors and bitflips with some non-negligible probability [1–8].

Concept. Rely is a new programming system that provides verified compilation for unreliable hardware architectures. Rely provides a programming language and a program analysis that, together, enable a developer to control a program's quantitative reliability [Carbin et al., OOPSLA, (Best Paper Award), 2013]. Rely's

programming language enables a developer to write programs that use unreliable ALU operations and allocate data in unreliable memories. For each function in the program, the developer can then write a *quantitative reliability specification* that identifies a lower bound on the probability that a function's implementation will produce the correct result. Given the probability with which each hardware operation executes correctly, Rely's program analysis then verifies that the function's implementation satisfies its specification.

Approach. Rely's program analysis system uses a novel assertion logic to characterize the quantitative reliability of intermediate values along paths through a function. The assertion logic works with the distribution of possible states of the unreliable execution to identify the probability that each intermediate value has the same value as if it were computed fully reliably. To verify a function's specification, the analysis uses the assertion logic to check that every path through the function (with sound handling of loops) produces a result that is at least as reliable as the specification requires.

Rely's analysis is fast and fully automatic. The analysis uses a novel simplification procedure to reduce the verification problem from one that considers all paths through the function to one that only considers the set of least reliable paths. Rely's experimental results show that this approach yields up to a four orders of magnitude reduction in the number of verification conditions, resulting in analysis times of less than one second for a benchmark set of computational kernels from approximate computing applications.

Relaxed Programs

Transformations that improve performance and/or resilience by changing the semantics of the program often operate by identifying points at which the program makes choices about how to organize its computation. Examples of these choices include the number of executed iterations of a loop, the specific algorithm for a computation, or the placement of program data in reliable or unreliable physical memories. These choices expose a range of alternate options that a transformation or runtime system can manipulate to adapt a program's performance, accuracy, and/or resilience.

Concept. Building on the concept of these choice points, I have developed a programming language and verification system for *relaxed programs*. A relaxed program is a standard program augmented with additional annotations that expose the choices that the programming system can use to configure the program's execution [Carbin et al., PLDI, 2012]. These choices can be specified by a developer or even automatically synthesized by a program transformation. For example, a transformation that skips loop iterations may add an annotation to a `for` loop that exposes the choice to run the loop for fewer iterations. This annotation-based approach makes it possible to use the same program text to reason about both the original program (by ignoring the annotations) and the relaxed program (by including the annotations' effects). This conjoined representation of the original and relaxed programs exposes their structural correspondence. The verification system then leverages this correspondence to enable effective relational verification of the relaxed program.

Approach. The programming language provides constructs for writing standard imperative programs augmented with these additional annotations. The annotations are specified as small declarative programs that characterize nondeterministic modifications to the program's state. For example, an annotation may truncate a loop's execution by nondeterministically skipping its remaining iterations. By using declarative programs for annotations (as opposed to a fixed annotation language), the programming language can encode a wide variety of choices and transformations.

The verification system includes a specification language that enables developers to specify not only standard assertions (e.g., memory safety), but also assertions that relate outputs and intermediate values of the relaxed program to those of the original program. For example, an assertion may state that the output of the relaxed program must be within 10% of that of the original program.

The verification system uses a novel relational Hoare Logic that exploits the structural correspondence between the original and relaxed programs. For example, it is possible to prove the memory safety of the relaxed program by assuming that the original program is memory safe and then verifying that the relaxed program performs a subset of the memory accesses of the original program. This style of proof establishes that the choices introduced for the relaxed program do not interfere with the reasoning that establishes a property for the original program. This proof style can therefore lower the complexity and cost of verifying a relaxed program by making it possible to reuse proofs and assumptions associated with the original program.

I have used an interactive theorem prover (Coq) to formalize the programming language and verification system. This additional step makes it possible to obtain fully machine-checked verifications of relaxed programs.

Summary and Future Directions

Giving the programming system the freedom to change the semantics of the program can deliver significant performance and resilience gains. However, to fully realize the benefits of this approach, it is critical to develop a sound methodology for reasoning about the variety of programs these systems produce. My work on both quantitative reliability and relaxed programs is the first to present a general methodology for reasoning about the behavior of these programs. Going forward, my work opens up new opportunities in approximate computing and emerging software development concerns.

Approximate Computing Frameworks. In traditional compiler frameworks, each optimization pass provides the modular guarantee that it will preserve the semantics of the program. This guarantee enables a compiler to sequentially compose optimization passes and still produce a semantically equivalent program at the end of the optimization process. For optimizations that change the behavior of programs, however, this compositional reasoning no longer holds. For example, optimizations that target unreliable hardware can deliver a range of trade-offs between the reliability and performance of the program. An open question is then how to automate the exploration of this trade-off space and preserve sufficient structure throughout the process to perform effective end-to-end verification.

My current work on quantitative reliability and relaxed programs is a good starting point. Building on this research, I will investigate a general compilation framework by exploring a wide variety of techniques for compiling to unreliable hardware. To support this effort, I will build new collaborations in the hardware community to increase the variety of unreliable hardware designs for which Rely can provide verified compilation.

Compiling programs to unreliable hardware can also produce a broader understanding of program approximation in general. For this reason, I will build collaborations with the numerical analysis, theory, and database communities that will address how programming systems can incorporate both algorithmic and data approximation. For example, I plan to investigate techniques such as replacing code with sublinear algorithms and representing program data with sparse, multiresolution data representations.

Emerging Software Development Concerns. The programming language community's goal of making software easier to write has been partially realized through communities like GitHub and Stack Overflow. By making it easy to access and adapt software components, these communities have brought about an ecosystem in which software components move rapidly from project to project and from domain to domain. As these forces continue to push software development, the vast majority of the components of future software systems will have uncertain provenance and operation. Software developers will therefore understand less of their systems' overall behavior than they currently do today.

Understanding software is currently one of the primary ways we gain confidence that our software provides some guarantee, such as security or functional correctness. As we continue to use collaborative software communities, building confidence in software will become even more important because these communities will face the same epidemiological challenges as human communities: bugs and security vulnerabilities will spread among programs. A critical research question will therefore become how to build confidence in these poorly understood systems.

Semi-automated program verification will solve this problem for core components of the software stack (e.g., compilers, operating systems, and standard libraries and data structures). These components have strong logical characterizations of correctness that are amenable to verification. For these systems, developers will specify interfaces for individual components and then verify that their implementations respect these specifications. Verified implementations will enable developers to reuse components with confidence – and without understanding their exact provenance or operation.

On the expansive periphery of the software ecosystem, however, where software is one-off, quickly developed, and often user-facing, correctness is less well-defined and resilience is a primary objective. For this software, we will need new techniques that build confidence through resilience. To this end, I will investigate how relaxed programs can serve as a platform for exposing and manipulating global system behaviors to create resilience. I will also investigate how to connect these behaviors with developers' limited understanding in a way that enables developers to distinguish between the set of behaviors they have built into their software and the set of behaviors that emerge from resiliency mechanisms. I will also investigate how to change software interfaces to coordinate software resiliency mechanisms with end-users' domain-specific goals. This approach has the potential to make poorly understood software systems fast, secure, and usable.

Referenced Publications

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Programming Languages Design and Implementation*, PLDI, 2012.

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin Rinard. Verified integrity properties for safe approximate program transformations. In *Workshop on Partial Evaluation and Program Manipulation*, PEPM, 2013.

Michael Carbin, Sasa Misailovic, Michael Kling, and Martin Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object-Oriented Programming*, ECOOP, 2011.

Michael Carbin, Sasa Misailovic, and Martin Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA, (Best Paper Award), 2013.

Hank Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA, 2012.

Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *International Conference of Software Engineering*, ICSE, 2012.

Jeff Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, SOSP, 2009.

External References

[1] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. ISCA '10.

[2] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. MICRO, 2003.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. ASPLOS, 2012.

[4] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. DATE, 2010.

[5] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: saving dram refresh-power through critical data partitioning. ASPLOS, 2011.

[6] S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. DATE, 2010.

[7] K. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 2005.

[8] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. PLDI, 2011.