Research in Program Repair and Approximate Computing: A Retrospective

Martin C. Rinard
MIT EECS & CSAIL
Cambridge, MA
rinard@csail.mit.edu 0000-0001-8095-8523

***Abstract*—This paper and accompanying talk trace the trajectory of my research in program repair and approximate computing. The prevailing value system in the field at the time focused on program correctness as a fundamental goal. This research, in contrast, was driven by a new perspective that emphasized acceptable (but not necessarily fully correct) survival through errors and the automatic identification and exploitation of performance versus accuracy tradeoff spaces implicitly present in computations coded to operate at only a single point in this space.**

Because the research challenged the prevailing value system at the time, it met with some skepticism despite empirical results highlighting its effectiveness. The following quote from an anonymous reviewer may give some idea of the reaction:

> ***"The basic idea—to assist incorrect programs in their efforts to emit incorrect output—is an abomination and if adopted would likely usher in a new dark age."***

As the research progressed, we gained a deeper understanding of the reasons behind the surprising — at least to us — phenomena we observed. We were able to formalize this understanding to generate source code patches and obtain performance, accuracy, and acceptability guarantees for computations that leveraged our techniques, bringing the research full circle to once again focus on reasoning statically about program behavior but with different reasoning techniques and guarantees.

Finally, I discuss lessons learned and future relevance of the principles, perspectives, and concepts that this research pioneered.

***Index Terms*—Program Repair, Approximate Computing**

I. Introduction

The citation for the 2025 ACM SIGSOFT Outstanding Research Award reads: "for fundamental contributions in pioneering the new fields of program repair and approximate computing." In this paper and accompanying talk I'll tell the story of this research — how it started, where it went, and what I ultimately learned from doing it. Figure 1 presents (linked) selected techniques and systems developed as part of the program repair research; Figure 2 presents the same for the approximate computing research.

The research itself started with a change in perspective. My research to that point had focused on formal semantics, program analysis, program verification, and parallel computing, which saw the computation as a rigid logical artifact whose semantics should be precisely specified and preserved. Consistent with the dominant value system in the field at the time, if an error was detected, the right response was to stop the computation rather than produce an incorrect output. While I enjoyed this research when I did it (and am still proud of this research today), over the years I had become increasingly aware of limitations of this perspective and was ready for a something new.

I therefore decided to view computations as flexible biological systems and see where that led. An important (some might say only) prerequisite for success as a biological organism is to survive. So I focused on techniques that enabled computations to survive, which meant the computation continued to execute, ideally to produce acceptable (but not necessarily perfect) results. Inverting the goal, I identified reasons why the computation would die and (initially) started with two: memory errors (such as null pointer or out of bounds access errors) and (with Brian Demsky) data structure consistency violations (my research group had been working on inferring and verifying data structure consistency properties so this was an obvious target). Instead of attempting to correct the defects that caused the errors, which would have put the research back into the mainstream at the time, we focused on recovery techniques for surviving these errors when they occurred:

- **Data Structure Repair:** This technique started with an (inferred or specified) data structure consistency specification and traversed the data structure to find and fix any data structure consistency violations by enforcing the specification. One notable aspect was that there was no guarantee that the data structure would be correct (because, for example, the error may have destroyed data that should have been present in the data structure). The guarantee was instead that the data structure would be consistent, which we hoped would be enough for the computation to survive.
  Whether this was a good idea or not was an empirical question — one could imagine situations where it would work well and situations where it would not work at all. So we acquired some computations with data structure corruption errors and tried it. The results were surprising even to us — the computations exhibited a level of resilience that we did not anticipate. And the reasons for this resilience were varied and illuminating. Without repair, the inconsistency typically caused the computation to crash or terminate. In some cases the repair simply filled in missing intermediate data structure nodes, enabling subsequent data structure operations to execute

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00257

Data Structure Repair
Automatic Patch Generation
ClearView
Kali, SPR, Prophet, Genesis
Code Transfer
CodePhage
CodeCarbonCopy
Execution Integrity Repair
Failure-Oblivious Computing
Recovery Shepherding
Cyclic Memory Allocation, Infinite Loop Escape
Filtered Iterators
Program Inference and Regeneration

Fig. 1. Program Repair: Techniques and Systems

Task Based Techniques
Task Skipping, Early Phase Termination
Topaz
Loop Perforation
Quality of Service Profiling
Accuracy, Reliability, Acceptability Verification
Chisel, Rely
Nondeterministic Acceptability
Displays and Devices
Ishihara, Crayon, DaltonQuant,
Warp, Rake, Lax
Dynamic Knobs
Approximate Parallel Computing
QuickStep
Approximate Parallel Data Structures
Reconfigurable Analog Devices
Legno
Jaunt, Arco

Fig. 2. Approximate Computing: Techniques and Systems

correctly. In other cases the repair produced a consistent, mostly correct, data structure missing some data, with the resulting computation providing acceptable service to users potentially with some anomalies (which was much better than no service at all).

- **Failure-Oblivious Computing:** This technique targeted null or out of bounds reads and writes. Dealing with writes was easy — simply discarding the write enabled the program to continue. Dealing with reads was more of a challenge because the program needed a value to continue. Not having a better idea, we decided to hand back a manufactured value. We settled on the sequence 0, 1, 2, 0, 1, 3, 0, 1, 4, ... because we knew (from the value prediction literature in computer architecture) that 0 and 1 were by far the most common values read in computer programs and we thought that if 0 or 1 didn't keep the computation satisfied it might need some specific value and this sequence would (eventually) give it that value.
A challenge here was finding the right context for the technique. We initially started with small computations with inserted defects and it didn't work at all. I was working with a young Cristian Cadar (at the time an undergraduate at MIT) and although he (like almost everyone else I spoke with about the technique) thought it would never work, he suggested we try to use it on computations with buffer overflow security vulnerabilities. The goal was to neutralize the attack (prevent the attacker exfiltrating data or taking over the computation) and keep the computation executing to successfully process subsequent legitimate inputs or requests.
So we got a C compiler that performed bounds checks, implemented the technique, acquired programs from the Unix ecosystem with security vulnerabilities (email programs, an http server, and a file management tool), and tried the technique. It worked great — the computations felt like they were bulletproof as they executed smoothly through malicious inputs that triggered errors. The standard Pine mail agent, for example, could be taken down during initialization when it attempted to load a mail file containing an email with a carefully crafted From field. With failure-oblivious computing, Pine survived to provide full functionality.
As we started looking into the reasons for the overall success of the technique, it became clear that the programs could all be seen as generalized servers processing sequences of requests, with each request triggering off a subcomputation to handle the request. Because these subcomputations were loosely coupled, the technique enabled a subcomputation that encountered an error to complete without corrupting the basic integrity of the computation (because of the bounds checks, any writes that would otherwise have corrupted the computation were discarded) and the computation went on to successfully process subsequent requests. Moreover, errors often occurred when malformed or malicious requests exercised missing input sanity checks, minimizing any concerns about the acceptability of results generated from subcomputations that processed these requests.

It is difficult to convey just how much skepticism this direction initially inspired. The programming language community in particular was deeply committed to correctness as a fundamental goal and reacted very negatively to the concept of executing through errors, especially with manufactured values. The following quote from an anonymous reviewer may give some idea of the intensity of the initial reaction:

> *"The basic idea—to assist incorrect programs in their efforts to emit incorrect output—is an abomination and if adopted would likely usher in a new dark age."*

The computer systems and software engineering communities were also skeptical but also more open to the idea — indeed,

I have consistently found that one of the strengths of the software engineering community is the value it places on new ideas and concepts and its willingness to consider ideas that go against the current value system. Because of this openness, many of the results that I am most proud of have been published in the software engineering literature.

Despite this skepticism, I persisted. The surprising empirical success of the techniques, along with the knowledge that large production software systems were rife with defects yet still worked well, helped keep me going. But this experience helped me more fully appreciate just how powerful an entrenched ideology can be and how resistant fields can be to evidence that contradicts the value system that the field has bought into.

II. Program Repair

At this point I had made the perspective transition and validated the new perspective on two classes of errors. Despite the skeptical initial response and corresponding difficulties publishing the research, I focused on extending the scope to systematically target other basic execution integrity errors, with the ultimate goal of making computations immune to this otherwise fatal class of errors. *Cyclic memory allocation* (with Huu Hai Nguyen) eliminated memory leaks by statically preallocating a fixed size circular buffer for all objects allocated at potentially leaky allocation sites. This technique was particularly controversial because, unlike previous techniques, which did not interfere with error-free computations, cyclic memory allocation had the potential to introduce new errors into otherwise error-free computations by overlaying multiple live objects into a single allocation slot within the circular buffer. Even in such cases, however, the technique promoted successful continued execution by preserving the type safety of objects allocated at that site.

One empirical insight from this research is how archaic, no longer relevant functionality acquired over the lifetime of a software system can place the entire computation at risk — many of the memory leaks we targeted occurred in obscure, no longer relevant but still network exposed functionality that could be successfully targeted by outside attackers. This fact minimized concerns that anomalies caused by overlaying live objects could impair the useful functionality of the system. Daniel Dumitran extended this technique to cyclically allocate other leakable resources such as file descriptors. *Detecting and escaping infinite or long running loops* (with Michael Kling, Sasa Misailovic, and Michael Carbin) ensured that the continued execution did not become unresponsive.

One particularly notable project was a collaboration led by Jeff Perkins that included Michael Ernst, Saman Amarasinghe, and a host of students and postdocs. *ClearView* worked with stripped x86 binaries to learn invariants over registers and memory locations, then generate patches that eliminated errors by enforcing the learned invariants. This substantial engineering effort produced a system that worked with a community of running computations to detect errors, evaluate multiple candidate patches for these errors in parallel, identify successful patches, then apply those successful patches throughout the community without restarting or otherwise perturbing the running computations. In multiple adversarial Red Team exercises, ClearView enabled successful continued execution in the face of otherwise successful security attacks. I was later told that when the sponsor saw the results, they said it was the closest thing to a miracle they had ever seen. Although it was straightforward to translate the generated binary patches back to source-level patches, the research did not emphasize that capability.

III. Approximate Computing

This direction emerged as a fortuitous offshoot of my program repair research. For my PhD research I had developed a language, Jade, for parallel and distributed computing. Jade structured the computation as sets of tasks and I had identified a common pattern in Jade computations, specifically that the final result was usually computed by combinining contributions from many parallel tasks. A standard error recovery technique would detect tasks that failed for one reason or another, then reexecute the task. But given how many tasks contributed to the final result, I wondered just how important the contribution from any one task could be. Inspired by the success of our previous program repair techniques, I decided to instead just *discard failed tasks* (and their contributions to the final result). One particularly appealing aspect of this approach was that it tolerated deterministic software errors, for which reexecution was pointless because reexecuted tasks would just fail again and the computation would never make forward progress.

To my surprise, I found that it was often possible to discard results from half or more of the tasks, failed or not, with minimal changes to the final computed result![1] This fact opened up a new research direction: *improving performance by discarding tasks*. I developed models that made it possible to improve performance by purposefully navigating the resulting performance versus accuracy tradeoff and also used the technique to eliminate poor parallel efficiency caused by waiting for long running tasks at parallel barriers.

Unfortunately, few computations come with explicit task boundaries. To make the technique broadly applicable, we had to find another class of subcomputations to skip. Because the tasks in Jade programs often just encapsulated multiple iterations of time consuming loops, an obvious target was loop iterations in sequential programs. The project involved Stelios Sidiroglou-Douskos and Sasa Misailovic, who modified the LLVM compiler to implement *loop perforation*: i.e., to skip iterations of time consuming loops, using profiling to explore the resulting loop perforation space to find loops that offered appealing performance versus accuracy tradeoffs. Hank Hoffmann supplied the final missing piece when he suggested applying the technique to the Parsec benchmark suite. The

[1] With the very important caveat that some tasks, for example tasks that built data structures accessed by the rest of the computation, had to execute without error for the computation as a whole to succeed. This turned out to be a recurring pattern in approximate computing. We called regions of the computation that had to execute without error critical regions and found combinations of profiling, fault injection, input fuzzing, and influence tracing to be an effective way to identify these critical regions.

technique was useful to Hank because his research involved applying control theory to manage resource fluctuations and he needed a way to modulate the resources required to produce desired results. Because the technique exposed a controllable performance versus accuracy tradeoff space implicitly (and often unexpectedly) present in computations coded to operate at a single point in this space, it gave him this capability.

IV. Living In The Comfort Zone

Over the course of the research our analysis indicated that errors were often triggered because an anomalous input violated some constraint that the code implicitly assumed but did not check. Typical inputs essentially never triggered such errors — any such missing checks had been encountered during testing or normal use and the defect corrected. This observation led to the concept of a *comfort zone* — a region of inputs similar to those the code has seen before and for which it is almost certain to deliver expected and acceptable behavior. Applying the basic philosophy of this research direction, I manually developed *rectifiers* that processed inputs to move them into the comfort zone. The results highlighted the effectiveness of these rectifiers in protecting computations against inputs that triggered errors while still delivering value available in such inputs to users.

This initial success motivated *input rectification*, a technique that automatically learns constraints characterizing typical inputs that the computation processes successfully, then enforces those constraints on new inputs to eliminate the ability of the new input to trigger the error. This research, with Fan Long, Vijay Ganesh, Michael Carbin, and Stelios Sidiroglou-Douskos, produced rectified inputs that the computation could safely process instead of discarding. The empirical results showed that, when applied to image files, the technique preserved much of the useful information in images that would have otherwise triggered errors (or been discarded by input filters). One particularly appealing aspect of this research was how it generalized a program repair goal of preserving value present in computations that encounter errors to an input repair goal of preserving value present in inputs that, without input rectification, trigger errors.

V. Displays and Devices

When Phillip Stanley-Marbell joined my research group, he took the research in a new direction, exploring *energy reducing approximations* targeting sensors, embedded systems, communication, and displays. The resulting approximations often exploited the fact that the target computations and hardware platforms often worked with imprecise or noisy data (both input and output) and were therefore inherently approximate (even if not explicitly engineered to exploit this approximation). This research showed that, for these computations on these platforms, an appropriately managed increase in the amount of approximation could often generate significant energy savings while preserving overall end to end utility. A particularly compelling project targeted displays, leveraging the flexibility of the human visual system to modify images to reduce display power dissipation while preserving acceptable display color accuracy.

In another project, Jurgen Cito and Julia Rubin worked with Phillip to reduce energy consumption in mobile devices by identifying and throttling recurrent advertising and analytics requests — a nice example of how identifying and targeting redundant activities can reduce resource consumption.

VI. Building on Understanding

While our first results were empirical, we always prioritized understanding the reasons behind the phenomena we observed. This understanding first focused on the interaction between the specific computations we worked with and the techniques we deployed on those computations. Building on our understanding of these interactions, we were ultimately able to generalize and formalize that understanding in the form of static analysis systems and programming language constructs that made it possible to reason about and obtain guarantees for the behavior of the resulting computations. We also developed systems for automatically generating source-level patches that corrected code defects responsible for triggered errors. Interestingly enough, these efforts brought the research full circle back to static analysis and verified guarantees, but with a different perspective, new properties, and new analysis and verification techniques.

A. Filtered Iterators

Our analysis of the reasons why our initial program repair techniques worked so well produced the insight that computations are often (implicitly) structured as servers that process sequences of requests. Even when such computations apparently process a single unified input (such as an input file or stream of bytes), the first step is to (potentially recursively) subdivide the input into separate input units, each of which triggers off a skippable subcomputation. A capstone result in this line of research, with Jiasi Shen, produced a programming language construct, *filtered iterators*, that made this pattern explicit in the code. The empirical evaluation demonstrated striking improvements in the overall robustness, reliability, and even correctness of computations coded with this construct. One key finding was that input units that trigger errors are often malformed and trigger an error because they exercise a missing input sanity check. In these cases the correct response is often to discard the input unit, with the filtered iterator construct therefore producing fully correct behavior. One particularly appealing aspect is the elimination of explicit error-handling code — because the programming language implementation detects and skips subcomputations that encounter execution integrity errors, there is no need to include checks for these errors in the code. The resulting code simplification makes the mainline structure of the computation more apparent and easily understood.

This research touched on two issues, error detection and efficient atomic execution, that I feel could easily benefit from more attention. The filtered iterator research targeted universal execution integrity errors (out of bounds access, divide by

zero, null pointer access) and supported explicit aborts but did not address silent data corruption errors. Sara Achour's Topaz system, which used anomaly detection to identify and discard tasks that produced unacceptably inaccurate results, highlighted one creative and productive way to identify and deal with otherwise silent computational errors. I always viewed this research as opening up new directions that would benefit from further exploration.

It became evident early on that maintaining the integrity of the computation was a prerequisite for the success of both program repair and approximate computing. Since we were aspiring to execute through errors, a potential threat to this integrity was errors that damaged this integrity beyond recovery. Using atomic execution to prevent the effects of encountered errors from propagating beyond aborted tasks is one way to prevent this propagation. Jiasi's filtered iterator system implemented a straightforward atomic task execution mechanism. Here I think there is a program analysis opportunity to minimize the state management overhead required to obtain efficient atomic execution.

B. Reasoning About Approximate Computations

Once we understood the reasons why approximate computations were successful, we were in a position to develop formal reasoning systems to provide reliability and accuracy guarantees (here accuracy is the distance between the result that the computation produces and the corresponding fully accurate result; reliability is the probability that the computation will produce an acceptably accurate result). Michael Carbin and Sasa Misailovic, with contributions from Deokhwan Kim, Sara Achour, and Zichao Qi, led the development of *relational program logic and verification systems* (Rely and Chisel) for specifying and verifying reliability and accuracy requirements, with the accuracy specified relative to an ideal error and noise free computation. This research targeted an approximate hardware platform with both reliable and more energy efficient unreliable components. Because the specifications identify a range of acceptable computations, they enable an optimization algorithm that automatically maps computations and data to reliable and unreliable hardware components to minimize energy consumption while still providing the specified accuracy and reliability guarantees.

Our investigation of the reasons behind the success of task and loop iteration skipping indicated that they worked because they targeted forms of redundancy implicitly present in the computations they successfully optimized. Collaborations with Sasa Misailovic, Dan Roy, Zeyuan Allen Zhu, and Jonathan Kelner formalized the reasons for this success and generated probabilistic accuracy guarantees for approximate computations that leveraged these and other forms of redundancy.

Although neural networks are inherently approximate, research with Yichen Yang showed that it is possible to leverage a model of the state space of the world and the observation process that produces neural network inputs to specify and verify the correctness of neural networks that perform perception tasks. Kai Jia developed efficient SAT based reasoning techniques for verifying the robustness of binarized neural networks. Leveraging perspectives from previous approximate computing research, this research worked with definitions of correctness and robustness appropriate for the underlying approximate neural network model of computing.

C. Large Language Models and Program Repair

One currently prominent topic is large language models for automatic patch generation. Based on recent experience, large language models are powerful pattern matchers but also, as demonstrated in research with Charles Jin, show intriguing signs of spontaneously inducing models of the reality reflected in their training data. Recent research with Rem Yang illustrates how this dual capability can place the large language model in a dilemma when attempting to reason about code. The idea is to take code that is in distribution, apply mutations that produce plausibly correct but out of distribution code, then ask the large language model to reason about the code. In this situation the large language model sometimes exhibits the capability to both 1) revert the mutation to reason about the original in distribution code and 2) reason successfully about the mutated code as written. Moreover, we have seen chain of thought dialogue in which the model, when given a mutated version, states that it thinks there is a defect in the mutated version and the correct version is the original version; however, because it was instructed to work with the code exactly as written, the model states that it is following instructions to generate the correct result for the mutated version. Notably, only the most recently released large language models exhibit this dual capability — earlier models are much more biased towards reverting the mutation and much less successful at reasoning about the mutated version (while also being less successful at reasoning about the original version). This fact suggests that an ability to reason, at least about the semantics of code, may only now be emerging in the latest large language models.

Here is what I think this may mean for automatic patch generation and software engineering tasks more generally. I expect large language models to be especially effective when the code is in distribution and pattern matching can successfully solve problems. For example, I expect large language models to perform well in generating patches that correct missing checks in common programming patterns as long as the training data contains examples that include these missing checks. But we have also observed situations in which the language model may mistakenly think the provided code is incorrect and generate patches that introduce coding defects into the correct provided code. We have seen this happen, for example, when the language model mistakenly thinks the correct code has an off by one error.

One consequence of these kinds of disparities in the ability of large language models to operate successfully in distribution versus out of distribution could easily be selective pressure that drives the software engineering community towards the use of coding patterns that appear commonly in the training

data, with a corresponding homogenization of code across the software ecosystem.

Large language models also dramatically reduce the cost of obtaining in distribution code. Automatic patch generation is a technology motivated in part by the need to protect the investment present in existing code. By reducing the cost of new code, large language models may open up a compelling alternative to automatic patch generation, specifically wholesale regeneration of new code instead of patching old code. Jiasi Shen's *program inference and regeneration* research pioneered this direction for database backed applications and (with Varun Mangalick) for applications that use key/value storage systems. Later research expanded the scope of regeneration to target software supply chain vulnerabilities. In addition to Jiasi, this later research included contributions from Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Julian Dai, Evangelous Lamprou, and Grigoris Ntousakis. Replacing legacy code with regenerated code could accelerate the homogenization of the sofware ecosystem, with the code base becoming increasingly in distribution over time. Automatic patch generation, in contrast, forces the language model to work with code that it may not have generated and may be less able to work with successfully.

One particularly appealing aspect of regeneration is that it enables large language models to work with code that is fully within distribution where they operate most successfully, promoting the application of large language models to reduce the cost and improve the accuracy and effectiveness of a range of software development and maintenance tasks (refactoring, verifying documentation accuracy, functionality augmentation), not just defect elimination. This dynamic could create a positive feedback effect that promotes a rapid replacement of potentially out of distribution existing software with in distribution updated or new software (either automatically generated or coded by human developers) throughout the entire software ecosystem.

Regenerated code needs some kind of a specification, even if (or especially if) the specification is partial. Starting with an existing (potentially defective, insecure, or obsolete) implementation, Jiasi's research used *active learning* to uniquely identify a corresponding (correct, secure, up to date) program in a domain specific language, then regenerate a new implementation in a target language or for a target implementation platform. Large language models open up the possibility of doing a direct translation from old source code (but run the risk of transferring defects or vulnerabilities along). The flexibility of large language models opens up the possibility of working with code from multiple different systems, code written in different programming languages, or multimodal specifications that combine code, natural language, pictures, diagrams, or any other forms of information. The ability of large language models to fill in missing pieces can enable them to work with partial specifications, including partial multimodal specifications, with the language model completing (or even fully regenerating) the specifications, code, or both.

VII. Source Level Program Repair

The early program repair research focused on dynamic techniques for trecovering from encountered errors. Fan Long's contributions to this approach set him up to pursue a different goal: generating source patches that corrected code defects to eliminate errors before they occurred. One approach would have been to translate the dynamic recovery techniques into source level patches in response to detected errors (like ClearView did for binary patches except for source code instead of binaries), and I think this approach would have been largely effective. However, Fan chose to targeted code defects exposed by incorrect behavior on one or more test inputs.

The research started with the finding that existing test inputs alone were inadequate for identifying correct (or even acceptable) patches — the *Kali* system showed that patches that simply deleted code involved in errors were often able to produce correct behavior on the available test inputs. This finding changed the methodology in the field — credible evaluations of automatic patch generation techniques now focus not just on test input behavior, but also include an analysis (typically performed by humans) of patch correctness.

Fan's subsequent research built on his deep understanding of software to deploy a range of sophisticated techniques — constraint solving, machine learning, effective search techniques — that delivered productive search spaces of potential patches. One key finding here was that there were often many more plausible patches (patches which produced correct results for all of the test inputs) than correct patches. Here Fan's *Prophet* system effectively deployed machine learning to identify and prioritize correct patches within the much larger set of plausible patches. Prophet (and its predecessor SPR) worked with a set of patch patterns (each of which captures a program transformation that generates the patch) that generated the space of patches. This line of research ultimately produced (with Peter Amidon) *Genesis*, a system that learned patch patterns from successful human patches.

This research had an enormous impact on the field:

- **Inadequate Test Suites:** It showed that the test suites in the benchmark sets at the time were completely ineffective at distinguishing correct patches from plausible patches.
- **Correct Patches Are Obtainable:** It showed that it was nevertheless possible to identify and obtain correct patches for a substantial number of the code defects that occurred in practice.
- **Techniques for Obtaining Correct Patches:** It pioneered key techniques for obtaining these correct patches; these techniques included effective patch search spaces generated by patch patterns, the use of machine learning to automatically obtain these patch patterns, and the use of machine learning to identify correct patches within the resulting automatically generated patch spaces.

With help from Fan Long and Eric Lahtinen, Stelios Sidiroglou-Douskos developed *CodePhage*, which pioneered a different approach, horizontal code transfer, to automatic patch

generation. CodePhage transferred missing input sanity checks from binary donor applications into source-level patches in recipient applications. This research leveraged multiple development efforts that targeted similar functionality and aimed to identify and integrate all of the sanity checks from developers working across multiple development projects. This line of research also produced a code transfer system, CodeCarbonCopy, that transferred complete pieces of functionality between applications.

These projects were two more examples of the research coming full circle, in this case back to the goal of eliminating errors by correcting coding defects at the source level.

VIII. From Approximate to Analog

All throughout the approximate computing research, I had been looking for hardware that could provide a compelling platform for the approximate optimizations we could support. Phillip Stanley-Marbell's embedded devices and battery driven displays were one example of such a platform but I was looking for more. I found it when I ran into Rahul Sarpeshkar at a Divali party at Arvind's house. I asked him what he did for research, he said analog computing, and we had the platform.

Sara Achour identified dynamical systems as an appropriate software abstraction and took on the project of compiling dynamical systems onto the configurable analog device that Rahul had at the time. With support from Yannis Tsividis the project later moved on to target the Sendyne HCDCv2. These devices presented a challenging but also very fruitful model of computation that motivated the development of completely new reasoning capabilities and optimization algorithms. A major challenge was that the software for such devices is directly exposed to, and must effectively manage, problematic physical phenomena such as noise, bandwidth limits, sampling rate limits, current and/or voltage range limits, quantization errors, and manufacturing variability that can cause analog components to deviate significantly from their ideal behavior. Dealing with all of these issues was a major challenge and Sara's thesis presenting techniques to effectively manage all of these phenomena while still generating efficient computations was a blockbuster that came in over 500 pages long!

For my PhD thesis I had used experimental hardware (the Stanford DASH machine) and therefore had some exposure to the associated frustrations and challenges. I can still remember the day the DASH machine finally become reliable enough to finish the Jade computations in my thesis. I stayed late into the night getting numbers, hardly being able to believe the machine was finally working. But the analog devices Sara was working with were next level challenging beyond anything I had previously worked with and it took a special researcher to get reliable results.

IX. What I Learned

Computation Characteristics: The most immediate findings identified and explored previously unsuspected characteristics that the research surfaced in the computations we worked with:

- **Inherent Resilience:** One of the most surprising findings was the implicit resilience inherently present in so many computations. Once this resilience was unlocked via the application of the simple recovery strategies that failure-oblivious computing pioneered, computations became remarkably robust against attacks or other events that exercised coding defects.

  So why did software (at the time) have the reputation of being so brittle and why was the emphasis on eliminating as many coding defects as possible? One explanation may involve the mindset of successful software developers — during development, terminating the execution as close as possible to the defect promotes rapid defect identification and correction (a major focus of many developer activities). Moreover, continuing after encountering an error (especially with manufactured values) takes the computation outside of its anticipated execution envelope, which can make developers nervous about the behavior that the continued execution may produce.

  This mindset, along with the alluring possibility of obtaining a defect free artifact that will never fail because of wear (a major concern for most other engineering disciplines), can easily produce computations that are not designed to tolerate errors. Note here the divergence between the mindsets and interests of developers and users — users (who typically have no ability to fix defects) are often better off with the service that even somewhat degraded continued execution can provide.
- **Tradeoff Spaces, Overprovisioning, and End to End Optimization:** Another surprising finding was the implicit, and in many cases unsuspected, presence of an underlying performance versus accuracy tradeoff space in computations coded to operate at a single point in this tradeoff space. And that this tradeoff space could be unlocked by simple, automated techniques such as skipping loop iterations.

  I attribute the presence of this tradeoff space to computational redundancy, often redundancy present in multiple contributions that combine to deliver a final result. Examples of patterns that exhibit this kind of redundancy include search space enumeration (where the approximation skips some of the points in the search space), search metrics (where the search is driven by a metric where the approximation delivers a less accurate but still serviceable metric), and iterative improvement (where the approximation executes fewer iterations of the improvement loop). This redundancy exists because the computations are arguably overprovisioned, for example because the developer is satisficing[2] instead of optimizing or because the computation does not need the full accuracy provided by utilized libaries (which are engineered to operate successfully across a range of contexts including contexts that require more accuracy). The success of our ap-

[2]A portmanteau introduced by Herb Simon meaning "to pursue the minimum satisfactory condition or outcome" (www.merriam-webster.com).

proximate computing research highlights how automated techniques can optimize end to end across the scope of the entire computation.

- **Critical and Forgiving Regions:** At the start of the research it became immediately clear that computations had two kinds of regions: critical regions that had to execute without perturbation for the computation to execute acceptably (an early prominent example was code that builds data structures that the remaining computation uses) and forgiving regions that could tolerate substantial perturbations without threatening the survival or acceptable accuracy of the computation. We found that identifying and targeting forgiving regions was a prerequisite for the success of approximate computing.

General Research Advice: In retrospect, we did several things right that should generalize to other research efforts. These are more about the research process than any specific research finding.

- **Adopt A Guiding Perspective:** This research started with a change in perspective, specifically seeing computations as flexible biological organisms whose existence is worth preserving. This new (at the time) perspective surfaced ideas and techniques not perceivable within the prevailing perspective. An advantage of this approach is that it helps guide the research when the best direction may be uncertain so that you can sustain a long arc research program that produces a coherent body of work with a unified message that becomes realized over time.
- **Invert the Value System:** Fields often become captured by an entrenched value system. Researchers that internalize this value system can become blind to directions that go against this value system. One way to become creative is to pick an implicit assumption that has become entrenched in the field (for example, that computations should execute correctly and stop when they encounter an error), negate that assumption, and see where it leads. The resulting perspective shift can often surface directions that others, trapped in the dominant value system, cannot see. Should you choose to do this, be prepared for a skeptical reaction. Fields can become fully committed to an ideological orthodoxy and react strongly against challenges to that orthodoxy. Moreover, there are often very good reasons for this orthodoxy, with the rationale supporting the orthodoxy well developed over time and internalized by the field. The rationale for new ideas, on the other hand, is typically not nearly as developed and the initial rationale may even be just wrong. If you encounter this situation one encouraging aspect is that you are not alone — researchers in all areas of science and technology who have successfully pioneered new ideas consistently report encountering substantial pushback and initial rejection of the idea when it first appears.
- **Opposing Perspectives and Adversarial Thinking:** New ideas and directions can be fragile — they often need time, development, and refinement to realize their potential. Working on a new idea can be discouraging because it can be difficult to come up with a rational argument for why it will eventually succeed. Especially at the early stages of the research the only way forward is often to just suspend disbelief and keep going.
  On the other hand, to develop the idea you need to be able to identify and acknowledge the drawbacks, then choose between several options: 1) fix the drawbacks, 2) wait and hope a fix becomes apparent as the research progresses, 3) accept the drawbacks as inherent in the idea and move forward, or 4) consider the drawbacks fatal and abandon the idea (which is unfortunately the right decision in some cases).
  This situation sets up an opposition between two prerequisites for success: believing in the idea despite all of the evidence (currently) against it versus acknowledging and dealing with the drawbacks. Adversarial thinking (alternating between challenging and defending the idea) can be a useful mental tool for effectively managing this opposition. Challenges often surface issues that need to be fixed; defending against the challenges can often clarify thinking and surface new directions that strengthen the research. Successfully balancing these opposing perspectives, which can be difficult since they require you to successfully balance two very different mindsets, is one key to success.
- **Talk to People:** From the very beginning of the research I spoke to as many people as I could. I found that just articulating an idea often clarified how I thought about it and generated new insights and directions. While much of the reaction (at least at the start) was skeptical, hearing and responding to different perspectives helped me develop the ideas in positive directions. Sometimes people provided missing pieces that helped the idea succeed — Cristian Cadar's identification of security vulnerabilities as a good target for failure-oblivious computing and Hank Hoffmann's identification of the Parsec benchmarks as a good target for loop iteration skipping were both critical to the overall success of those two projects.
  Note that you should talk to a much wider circle of people than your collaborators, colleagues in the field, or even technical professionals — I often found the fresh perspective of and basic questions from people not familiar with the research or even the technical context in which the research took place to be enormously valuable in clarifying my understanding of what I was doing and why. Indeed, I found that propositions people in the field found very difficult to accept were often seen as common sense by people outside the field — in some cases because of their perspective as users of technology rather than creators of technology, in others because of their expertise in fields with very different contexts and value systems.
- **Pay Attention to Unexpected Findings:** The approximate computing research started with an unexpected finding (that it was possible to acceptably skip large numbers of tasks in Jade computations) encountered in

a research program directed towards an entirely different goal (simplifying error recovery). The resulting approximate computing research highlights how paying attention to and appropriately following up on these kinds of surprising results can often generate productive new research directions.

- **Good Collaborators:** Finally, this research was fortunate in the quality and quantity of collaborators it manged to attract. These collaborators refined and extended the ideas, proved them out in multiple contexts, and identified new and productive directions. Be fortunate in your choice of collaborators!

X. Missed Opportunities

In retrospect, there are two directions that I think we would have done well to pursue but didn't. The first is generating source-level patches earlier in the research. These patches would have translated the dynamic recovery techniques for detected errors into source-level patches that implemented the recovery actions for those errors. This would have been relatively straightforward to do and would have broadened the scope of the research and eliminated the need for a nonstandard runtime environment (in retrospect, a barrier to adoption).

The second direction is applying approximate computing to neural networks. As the approximate computing research was winding down, neural networks started their rise to the central role they now play in computing. Neural networks, and the results they produce, are fundamentally approximate to begin with and would have been a compelling target for optimizations that exploited further approximation opportunities to improve performance, energy efficiency, or other goals. Indeed, many optimizations that have since been applied to neural networks target redundancy profitably exploitable by skipping or otherwise removing computation. Examples include distillation, pruning, and quantization (some pioneered by Michael Carbin and Jonathan Frankle after Michael's graduation). This fact highlights the generalizable validity of the basic premise of our approximate computing research.

While there are a variety of reasons we did not pursue these directions at the time, one reason is the fog of research. Once the research was in full swing we were in the middle of it with many promising directions available. In this situation it can be difficult to fully optimize the directions one chooses to pursue.

XI. The Present

Developments have validated the basic perspective that this research advocated. As software has become ever more integrated into the basic functions of society, continued meaningful execution, even at the cost of some local anomalies if the alternative is denial of service, has only become more important. And consistent with our initial findings, preserving basic data consistency and computation integrity (which our techniques emphasized) is a prerequisite for this meaningful continued execution. One basic principle that we advocated was that aspiring to software perfection could be counterproductive and acceptability was often a more realistic and productive goal.

As systems continue to grow in complexity and scale, something is always going to be wrong somewhere, and the importance of this principle continues go grow. Current systems are now request driven and designed to tolerate and encapsulate errors in request triggered computations, often by discarding computations that encounter errors. Indeed, many of the systems upon which our society depends operate successfully because of the implicit resilience inherent in systems that work with this basic structure.

Systems that work with noisy data are inherently approximate, which often opens up ways to further exploit this approximation to promote other goals (such as energy consumption, increased system scope, or better responsiveness). The two dominant categories of data today, sensor data from the physical world and unstructured data from the Internet, are both noisy and systems that process this data (most recently large language models) are inherently approximate and often leverage that approximation in service of other goals. Approximation and approximate computing are now pervasive across the entire field of computer science, with many optimizations built on the basic approach of successfully navigating performance versus accuracy tradeoffs exposed by skipping partially redundant computations.

XII. The Future

The software engineering context has changed significantly since this research was initiated, most prominently with the advent of code generated by large language models. The basic model of computation that these large language models implement is approximate — they are trained on noisy, unstructured data and produce probabilistically generated results. Optimizations that target resource consumption, in both training and inference, heavily exploit the flexibility that approximation brings. New programming models like probabilistic programming are also inherently approximate. Approximate computing, including exploiting approximation opportunities available in redundant computations for resource optimization and error tolerance, is now central to computing and will only become more important in the future.

At the same time, traditional computing systems that work with digital data will remain an important foundation of our baseline computing infrastructure. These systems are driven by cascading sequences of requests at scale. While I expect code generated by large language models and the use of safer programming languages to eliminate many of the basic execution integrity errors that our early research targeted, things will continue to go wrong in various ways, with resilience in the form of continued execution in the face of these largely unanticipated events only growing in importance. Whether the code is generated by large language models, human developers, or some combination, I therefore expect acceptability based error tolerance techniques to continue to

play a major role in ensuring the overall success of our computing infrastructure now and in the future.

Reduced Cost of In Distribution Software: Large language models will significantly reduce the cost of obtaining in distribution software at scale. Reductions in the cost of resources have often driven transformational societal change, even if there is a delay in understanding how to best utilize the resource. Software has traditionally been a relatively expensive resource that required specialized expertise to obtain. Libraries have captured developer effort and expertise in reusable form for decades; I expect language models to provide a much more fluid, flexible automated mechanism for accessing and building on encapsulated expertise. One consequence may be a reduced commitment to an existing code base, with maintenance replaced by regeneration (and a concomitant reduction in the value of technologies such as automated patch generation that are designed to preserve and extend the value present in a given code base). As part of this process we may see software designs, programming languages, and coding patterns converge to a consensus around those most in distribution for large language models. For an early example of the incentives that will drive this convergence, we have seen large language models tasked with correcting errors incorrectly introduce errors into correct but out of distribution code, disincentivizing the use of out of distribution code. This consensus will emerge organically via the interaction between exising training sets and new code (from either human developers or large language models) entering subsequent training sets.

More generally, large language models may reduce the cost of reducing domain expertise to code, changing the balance between the value of domain expertise and software development expertise. In my experience one of the most valuable talents that software developers bring to the development of software that is deeply embedded in a domain is the ability to generalize from specific problems in the domain to concepts and processes that generalize across the entire domain. Domain experts often think primarily in terms of the problem in front of them, while software that operates within the domain must work more generally across the domain. If language models can generalize, with the generalization process potentially drawing on their background knowledge, this may reduce the cost of going from specific problems in the domain to general concepts and processes across the domain.

One of the more potentially impactful aspects of language models is their ability to capture the full range of knowledge and expertise present in their training data, then make this knowledge and expertise flexibly available. The ability to automatically generate unlimited amounts of code with automated feedback immediately available makes software a particularly appealing target because it enables essentially unlimited training data and learning.

Requirements, Specifications, and Designs: In addition to capturing code, large language models capture associated information about requirements, specifications, and designs. Because aspects of these elements appear repeatedly across multiple software systems, we may be looking at a future in which language models play a major role in defining requirements, generating specifications, and providing designs that implement these specifications in addition to providing code. Because of the remarkable fluency that large language models exhibit with this kind of information, we can expect to see language models deliver polished artifacts in all of these areas. This opens up the possibility of generating verifiable connections between requirements, specifications, designs, and software, which could help promote cohesion across the software system and address current concerns about the correctness of software obtained from language models.

XIII. Further Reading

For more on the following topics, see the referenced web pages, whch contain summaries of the research and links to relevant papers:

- **Broad Research Overview:** https://people.csail.mit.edu/rinard/research/
- **Acceptability-Oriented Computing:** https://people.csail.mit.edu/rinard/research/AcceptabilityOrientedComputing/
- **Approximate Computing:** https://people.csail.mit.edu/rinard/research/ApproximateComputing/
- **Automatic Patch Generation:** https://people.csail.mit.edu/rinard/research/AutomaticPatchGeneration/
- **Code Transfer:** https://people.csail.mit.edu/rinard/research/CodeTransfer/
- **Data Structure Consistency:** https://people.csail.mit.edu/rinard/research/DataStructureConsistency/
- **List of Papers:** https://people.csail.mit.edu/rinard/paper/

XIV. Selected Papers

Automatic Patch Generation: The following three papers present automatic patch generation techniques. The first two generate source level patches; the third generates binary patches that can be applied to running programs without restarts or service interruptions.

Fan Long, Peter Amidon, and Martin C. Rinard (2017). "Automatic inference of code transforms for patch generation". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, pp. 727–739. URL: https://doi.org/10.1145/3106237.3106253.

Fan Long and Martin C. Rinard (2016b). "Automatic patch generation by learning correct code". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, pp. 298–312. URL: https://doi.org/10.1145/2837614.2837617.

Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory T. Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard (2009). "Automatically patching errors in deployed software". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, pp. 87–102. URL: https://doi.org/10.1145/1629575.1629585.

Data Structure Repair: The following paper is the capstone paper in the data structure repair line of research.

Brian Demsky and Martin C. Rinard (2006). "Goal-Directed Reasoning for Specification-Based Data Structure Repair". In: *IEEE Trans. Software Eng.* 32.12, pp. 931–951. URL: https://doi.org/10.1109/TSE.2006.122.

Approximate Computing: The following two papers present loop perforation (skipping loop iterations) as an optimization and as a performance profiling technique for finding approximate regions of the program that deliver large performance improvements at the cost of small losses in accuracy. The third paper presents task skipping as an error tolerance and approximate optimization technique.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard (2011). "Managing performance vs. accuracy trade-offs with loop perforation". In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, pp. 124–134. URL: https://doi.org/10.1145/2025113.2025133.

Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin C. Rinard (2010). "Quality of service profiling". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, pp. 25–34. URL: https://doi.org/10.1145/1806799.1806808.

Martin C. Rinard (2006a). "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks". In: *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*. ACM, pp. 324–334. URL: https://doi.org/10.1145/1183401.1183447.

Error Recovery and Error Tolerance: The following papers present input rectification, which automatically enforces learned or manually implemented input constraints to move inputs that would otherwise trigger errors into the comfort zone of the computation (familiar inputs for which the computation is almost certain to deliver acceptabile behavior), and failure-oblivious computing, which discards out of bounds writes and manufactures values for out of bounds reads, enabling computations to execute through otherwise fatal errors to successfully process subsequent inputs and requests.

Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin C. Rinard (2012). "Automatic input rectification". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, pp. 80–90. URL: https://doi.org/10.1109/ICSE.2012.6227204.

Martin C. Rinard (2007b). "Living in the comfort zone". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, pp. 611–622. URL: https://doi.org/10.1145/1297027.1297072.

Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee (2004). "Enhancing Server Availability and Security Through Failure-Oblivious Computing". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association, pp. 303–316. URL: http://www.usenix.org/events/osdi04/tech/rinard.html.

Color Approximation for Energy Efficient Displays: The following paper presents a technique for manipulating displayed images to optimize energy consumption while minimizing the impact on human perception.

Phillip Stanley-Marbell, Virginia Estellers, and Martin C. Rinard (2016). "Crayon: saving power through shape and color approximation on next-generation displays". In: *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. ACM, 11:1–11:17. URL: https://doi.org/10.1145/2901318.2901347.

Compiling to Reconfigurable Analog Devices: The following paper presents the Legno compiler, which compiles dynamical systems onto a physical (as opposed to simulated) reconfigurable analog device.

Sara Achour and Martin C. Rinard (2020). "Noise-Aware Dynamical System Compilation for Analog Devices with Legno". In: *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, pp. 149–166. URL: https://doi.org/10.1145/3373376.3378449.

XV. Papers By Topic

Automatic Patch Generation

Fan Long, Peter Amidon, and Martin C. Rinard (2017). "Automatic inference of code transforms for patch generation". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, pp. 727–739. URL: https://doi.org/10.1145/3106237.3106253.

Fan Long and Martin C. Rinard (2016a). "An analysis of the search spaces for generate and validate patch generation systems". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, pp. 702–713. URL: https://doi.org/10.1145/2884781.2884872.

Fan Long and Martin C. Rinard (2016b). “Automatic patch generation by learning correct code”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, pp. 298–312. URL: https://doi.org/10.1145/2837614.2837617.

José Pablo Cambronero, Jiasi Shen, Jürgen Cito, Elena L. Glassman, and Martin C. Rinard (2019). “Characterizing Developer Use of Automatically Generated Patches”. In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*. IEEE Computer Society, pp. 181–185. URL: https://doi.org/10.1109/VLHCC.2019.8818884.

Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin C. Rinard (2015). “Automatic error elimination by horizontal code transfer across multiple applications”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, pp. 43–54. URL: https://doi.org/10.1145/2737924.2737988.

Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin C. Rinard (2017). “CodeCarbonCopy”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. Ed. by Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman. ACM, pp. 95–105. URL: https://doi.org/10.1145/3106237.3106269.

Fan Long and Martin C. Rinard (2015). “Staged program repair with condition synthesis”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, pp. 166–178. URL: https://doi.org/10.1145/2786805.2786811.

Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard (2015). “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. ACM, pp. 24–36. URL: https://doi.org/10.1145/2771783.2771791.

Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory T. Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard (2009). “Automatically patching errors in deployed software”. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, pp. 87–102. URL: https://doi.org/10.1145/1629575.1629585.

Data Structure Repair

Brian Demsky and Martin C. Rinard (2006). “Goal-Directed Reasoning for Specification-Based Data Structure Repair”. In: *IEEE Trans. Software Eng.* 32.12, pp. 931–951. URL: https://doi.org/10.1109/TSE.2006.122.

Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard (2006). “Inference and enforcement of data structure consistency specifications”. In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. ACM, pp. 233–244. URL: https://doi.org/10.1145/1146238.1146266.

Brian Demsky and Martin C. Rinard (2005). “Data structure repair using goal-directed reasoning”. In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM, pp. 176–185. URL: https://doi.org/10.1145/1062455.1062499.

Brian Demsky and Martin C. Rinard (2003a). “Static Specification Analysis for Termination of Specification-Based Data Structure Repair”. In: *14th International Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA*. IEEE Computer Society, pp. 71–84. URL: https://doi.org/10.1109/ISSRE.2003.1251032.

Brian Demsky and Martin C. Rinard (2003b). “Automatic detection and repair of errors in data structures”. In: *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*. ACM, pp. 78–95. URL: https://doi.org/10.1145/949305.949314.

Approximate Computing

Michael Carbin, Sasa Misailovic, and Martin C. Rinard (2016). “Verifying quantitative reliability for programs that execute on unreliable hardware”. In: *Commun. ACM* 59.8, pp. 83–91. URL: https://doi.org/10.1145/2958738.

Sara Achour and Martin C. Rinard (2015). “Approximate computation with outlier detection in Topaz”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, pp. 711–730. URL: https://doi.org/10.1145/2814270.2814314.

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard (2014). “Chisel: reliability- and accuracy-aware optimization of approximate computational kernels”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. ACM, pp. 309–328. URL: https://doi.org/10.1145/2660193.2660231.

Sasa Misailovic, Deokhwan Kim, and Martin C. Rinard (2013). “Parallelizing Sequential Programs with Statistical Accuracy Tests”. In: *ACM Trans. Embed. Comput. Syst.* 12.2s, 88:1–88:26. URL: https://doi.org/10.1145/2465787.2465790.

Martin C. Rinard (2013). “Parallel Synchronization-Free Approximate Data Structure Construction”. In: *5th USENIX Workshop on Hot Topics in Parallelism, HotPar’13, San*

Jose, CA, USA, June 24-25, 2013. USENIX Association. URL: https://www.usenix.org/conference/hotpar13/workshop-program/presentation/rinard.

Michael Carbin, Sasa Misailovic, and Martin C. Rinard (2013). "Verifying quantitative reliability for programs that execute on unreliable hardware". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, pp. 33–52. URL: https://doi.org/10.1145/2509136.2509546.

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard (2013). "Verified integrity properties for safe approximate program transformations". In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*. ACM, pp. 63–66. URL: https://doi.org/10.1145/2426890.2426901.

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard (2012). "Proving acceptability properties of relaxed nondeterministic approximate programs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, pp. 169–180. URL: https://doi.org/10.1145/2254064.2254086.

Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin C. Rinard (2012). "Randomized accuracy-aware program transformations for efficient approximate computations". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM, pp. 441–454. URL: https://doi.org/10.1145/2103656.2103710.

Sasa Misailovic, Stelios Sidiroglou, and Martin C. Rinard (2012). "Dancing with uncertainty". In: *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability, RACES@SPLASH 2012, Tucson, Arizona, USA, October 21, 2012*. ACM, pp. 51–60. URL: https://doi.org/10.1145/2414729.2414738.

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin C. Rinard (2011). "Dynamic knobs for responsive power-aware computing". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. ACM, pp. 199–212. URL: https://doi.org/10.1145/1950365.1950390.

Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard (2011). "Probabilistically Accurate Program Transformations". In: *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. Vol. 6887. Lecture Notes in Computer Science. Springer, pp. 316–333. URL: http://hdl.handle.net/1721.1/73897.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard (2011). "Managing performance vs. accuracy trade-offs with loop perforation". In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, pp. 124–134. URL: https://doi.org/10.1145/2025113.2025133.

Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin C. Rinard (2010). "Quality of service profiling". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, pp. 25–34. URL: https://doi.org/10.1145/1806799.1806808.

Martin C. Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou (2010). "Patterns and statistical analysis for understanding reduced resource computing". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, pp. 806–821. URL: https://doi.org/10.1145/1869459.1869525.

Martin C. Rinard (2007a). "Using early phase termination to eliminate load imbalances at barrier synchronization points". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, pp. 369–386. URL: https://doi.org/10.1145/1297027.1297055.

Martin C. Rinard (2006a). "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks". In: *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*. ACM, pp. 324–334. URL: https://doi.org/10.1145/1183401.1183447.

Error Recovery and Error Tolerance

Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard (2014). "Automatic runtime error repair and containment via recovery shepherding". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, pp. 227–238. URL: https://doi.org/10.1145/2594291.2594337.

Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin C. Rinard (2012). "Automatic input rectification". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, pp. 80–90. URL: https://doi.org/10.1109/ICSE.2012.6227204.

Martin C. Rinard (2007b). "Living in the comfort zone". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, pp. 611–622. URL: https://doi.org/10.1145/1297027.1297072.

Michael Kling, Sasa Misailovic, Michael Carbin, and Martin C. Rinard (2012). "Bolt: on-demand infinite loop escape in unmodified binaries". In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. ACM, pp. 431–450. URL: https://doi.org/10.1145/2384616.2384648.

Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard (2011). "Detecting and Escaping Infinite Loops with Jolt". In: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. Vol. 6813. Lecture Notes in Computer Science. Springer, pp. 609–633. URL: http://hdl.handle.net/1721.1/73898.

Martin C. Rinard (2012). "Obtaining and reasoning about good enough software". In: *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*. ACM, pp. 930–935. URL: https://doi.org/10.1145/2228360.2228526.

Michael Carbin and Martin C. Rinard (2010). "Automatically identifying critical input regions and code in applications". In: *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*. ACM, pp. 37–48. URL: https://doi.org/10.1145/1831708.1831713.

Huu Hai Nguyen and Martin C. Rinard (2007). "Detecting and eliminating memory leaks using cyclic memory allocation". In: *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*. ACM, pp. 15–30. URL: https://doi.org/10.1145/1296907.1296912.

Martin C. Rinard (2006b). "Automated Techniques for Surviving (Otherwise) Fatal Software Errors". In: *Proceedings of the Workshop on Verification and Debugging, V&D@FLoC 2006, Seattle, WA, USA, August 21, 2006*. Vol. 174. Electronic Notes in Theoretical Computer Science 4. Elsevier, pp. 113–116. URL: https://doi.org/10.1016/j.entcs.2006.12.033.

Martin C. Rinard, Cristian Cadar, and Huu Hai Nguyen (2005). "Exploring the acceptability envelope". In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM, pp. 21–30. URL: https://doi.org/10.1145/1094855.1094866.

Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu (2004). "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)". In: *20th Annual Computer Security Applications Conference (ACSAC 2004), 6-10 December 2004, Tucson, AZ, USA*. IEEE Computer Society, pp. 82–90. URL: https://doi.org/10.1109/CSAC.2004.2.

Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee (2004). "Enhancing Server Availability and Security Through Failure-Oblivious Computing". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association, pp. 303–316. URL: http://www.usenix.org/events/osdi04/tech/rinard.html.

Displays and Devices

Phillip Stanley-Marbell and Martin C. Rinard (2017). "Error-Efficient Computing Systems". In: *Found. Trends Electron. Des. Autom.* 11.4, pp. 362–461. URL: https://doi.org/10.1561/1000000049.

Phillip Stanley-Marbell and Martin C. Rinard (2018). "Perceived-Color Approximation Transforms for Programs that Draw". In: *IEEE Micro* 38.4, pp. 20–29. URL: https://doi.org/10.1109/MM.2018.043191122.

Phillip Stanley-Marbell and Martin C. Rinard (2020). "Warp: A Hardware Platform for Efficient Multimodal Sensing With Adaptive Approximation". In: *IEEE Micro* 40.1, pp. 57–66. URL: https://doi.org/10.1109/MM.2019.2951004.

José Pablo Cambronero, Phillip Stanley-Marbell, and Martin C. Rinard (2018). "Incremental Color Quantization for Color-Vision-Deficient Observers Using Mobile Gaming Data". In: *CoRR* abs/1803.08420. URL: http://arxiv.org/abs/1803.08420.

Phillip Stanley-Marbell and Martin C. Rinard (2016). "Reducing serial I/O power in error-tolerant applications by efficient lossy encoding". In: *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 62:1–62:6. URL: https://doi.org/10.1145/2897937.2898079.

Phillip Stanley-Marbell, Virginia Estellers, and Martin C. Rinard (2016). "Crayon: saving power through shape and color approximation on next-generation displays". In: *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. ACM, 11:1–11:17. URL: https://doi.org/10.1145/2901318.2901347.

Phillip Stanley-Marbell, Pier Andrea Francese, and Martin C. Rinard (2016). "Encoder logic for reducing serial I/O power in sensors and sensor hubs". In: *2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, August 21-23, 2016*. IEEE, pp. 1–2. URL: https://doi.org/10.1109/HOTCHIPS.2016.7936231.

Phillip Stanley-Marbell and Martin C. Rinard (2015a). "Efficiency Limits for Value-Deviation-Bounded Approximate Communication". In: *IEEE Embed. Syst. Lett.* 7.4, pp. 109–112. URL: https://doi.org/10.1109/LES.2015.2475216.

Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin C. Rinard (2016). "Battery-aware transformations in mobile applications". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, pp. 702–707. URL: https://doi.org/10.1145/2970276.2970324.

Julia Rubin, Michael I. Gordon, Nguyen Nguyen, and Martin C. Rinard (2015). "Covert Communication in Mobile Appli-

cations (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, pp. 647–657. URL: https://doi.org/10.1109/ASE.2015.66.

Phillip Stanley-Marbell and Martin C. Rinard (2015b). "Lax: Driver Interfaces for Approximate Sensor Device Access". In: *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/stanley-marbell.

Program Inference and Regeneration

Jiasi Shen and Martin C. Rinard (2021). "Active Learning for Inference and Regeneration of Applications that Access Databases". In: *ACM Trans. Program. Lang. Syst.* 42.4, 18:1–18:119. URL: https://doi.org/10.1145/3430952.

Jiasi Shen and Martin C. Rinard (2019). "Using active learning to synthesize models of applications that access databases". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, pp. 269–285. URL: https://doi.org/10.1145/3314221.3314591.

Martin C. Rinard, Jiasi Shen, and Varun Mangalick (2018). "Active learning for inference and regeneration of computer programs that store and retrieve data". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018*. ACM, pp. 12–28. URL: https://doi.org/10.1145/3276954.3276959.

Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard (2021). "Supply-Chain Vulnerability Elimination via Active Learning and Regeneration". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, pp. 1755–1770. URL: https://doi.org/10.1145/3460120.3484736.

José Pablo Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard (2019). "Active learning for software engineering". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*. ACM, pp. 62–78. URL: https://doi.org/10.1145/3359591.3359732.

Compiling to Reconfigurable Analog Devices

Sara Achour and Martin C. Rinard (2020). "Noise-Aware Dynamical System Compilation for Analog Devices with Legno". In: *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, pp. 149–166. URL: https://doi.org/10.1145/3373376.3378449.

Sara Achour and Martin C. Rinard (2018). "Time Dilation and Contraction for Programmable Analog Devices with Jaunt". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. ACM, pp. 229–242. URL: https://doi.org/10.1145/3173162.3173179.

Sara Achour, Rahul Sarpeshkar, and Martin C. Rinard (2016). "Configuration synthesis for programmable analog devices with Arco". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, pp. 177–193. URL: https://doi.org/10.1145/2908080.2908116.

Emergent Meaning in Large Language Models

Charles Jin and Martin C. Rinard (2024a). "Latent Causal Probing: A Formal Perspective on Probing with Causal Models of Data". In: *CoRR* abs/2407.13765. URL: https://arxiv.org/abs/2407.13765.

Charles Jin and Martin C. Rinard (2024b). "Emergent Representations of Program Semantics in Language Models Trained on Programs". In: *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. URL: https://openreview.net/forum?id=8PTx4CpNoT.

Verifying Neural Networks

Kai Jia and Martin C. Rinard (2020). "Efficient Exact Verification of Binarized Neural Networks". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. URL: https://proceedings.neurips.cc/paper/2020/hash/1385974ed5904a438616ff7bdb3f7439-Abstract.html.

Yichen Yang and Martin C. Rinard (2019). "Correctness Verification of Neural Networks". In: *CoRR* abs/1906.01030. URL: http://arxiv.org/abs/1906.01030.

Kai Jia and Martin C. Rinard (2021). "Verifying Low-Dimensional Input Neural Networks via Input Quantization". In: *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*. Vol. 12913. Lecture Notes in Computer Science. Springer, pp. 206–214. URL: https://arxiv.org/abs/2108.07961.

ACKNOWLEDGMENTS

I dedicate this paper and accompanying talk to Jeff Perkins and Stelios Sidiroglou-Douskos, research scientists who played a major role in the program repair and approximate computing research that this award recognizes. I acknowledge contributions from the following collaborators on the approximate computing and program repair research discussed in this paper: Sara Achour, Anant Agarwal, Saman Amarasinghe, Peter Amidon, Jonathan Bachrach, William S. Beebee, Jr., Achilles Benetopoulos, Jose Cambronero, Cristian Cadar, Michael Carbin, Jurgen Cito, Julian Dai, Thurston Dang, Brian Demsky, Daniel Dumitran, Anthony Eden, Michael Ernst, Virginia Estellers, P.A. Francese, Vijay Ganesh, Elena Glassman, Michael I. Gordon, Philip Guo, Kai Jia, Charles Jin, Shivam Handa, Henry Hoffmann, Jonathan A. Kelner, Youry Khmelevsky, Deokhwan Kim, Sunghun Kim, Michael Kling, Evangelous Lamprou, Eric Lahtinen, Sam Larsen, Tudor Leu, Fan Long, Varun Mangalick, Stephen McCamant, Sasa Misailovic, Huu Hai Nguyen, Nguyen Nguyen, Grigoris Ntousakis, Carlos Pacheco, Jeff Perkins, Zichao Qi, Dan Roy, Julia Rubin, Rahul Sarpeshkar, Alizee Schoen, Frank Sherwood, Jiasi Shen, Stelios Sidiroglou-Douskos, Phillip Stanley-Marbell, Greg Sullivan, Yannis Tsividis, Nikos Vasilakis, Weng-Fai Wong, Jerry Wu, Rem Yang, Yichen Yang, Karen Zee, Zeyuan Allen Zhu, and Yoav Zibin.

I acknowledge valuable feedback on drafts of this paper from Saman Amarasinghe, Ann McLaughlin, Logan Engstrom, Sasa Misailovic, Cristian Cadar, Srini Devadas, Joel Emer, Daniel Jackson, and Rem Yang. Idan Orzach provided invaluable assistance with LaTex.

I also acknowledge related work in program repair and approximate computing performed by others, including research that substantially extends the reach and sophistication of these techniques. Because this paper is intended to tell the story of the research that took place in my research group, I omit a treatment of related work in these (now very large and successful) research areas.

The research discussed in this retrospective was supported by a variety of sponsors as acknowledged in the papers listed in Section XV.