My research areas of interest are programming languages and software engineering, with a focus on program analysis and software testing. For my doctoral dissertation, I developed and applied static program analyses to make error handling in large software systems more reliable [3, 5, 7, 6, 8]. My analyses aim to help developers understand how run-time errors propagate through software and find error-propagation bugs that could lead to serious problems including system crashes, security vulnerabilities, silent data loss, and data corruption. In addition to finding hundreds of confirmed bugs in real-world systems such as the Linux kernel and Mozilla Firefox, my error-propagation analysis also found a critical bug in heavily tested code used for space missions by the NASA/JPL Laboratory.

Most recently, I have been exploring a different application domain: numerical programs. Testing and debugging numerical programs pose unique challenges due to the use of floating-point arithmetic. In general, it is difficult to reason about floating-point programs, and developers often resort to using the highest available floating-point precision whenever possible. This can severely impact program performance. To alleviate this problem, I developed PRECIMONIOUS, a dynamic analysis tool to assist developers in tuning the precision of their floating-point programs to improve performance [9]. I am currently designing and implementing a framework for developing dynamic program analyses to improve the reliability of numerical programs.

1 Research Mission and Philosophy

My research aims to design and build tools to help developers write more reliable and efficient software by developing scalable, precise, and informative program analyses.

I believe developers constitute an important source of domain knowledge when building tools. Unfortunately, as tool designers, we are far from exploiting developers' knowledge and expertise. Developers should be kept in the loop when designing program analysis tools. To make a tool attractive for developers to use, it should scale to real software, require minimum effort to be used (e.g., it should be easy to configure and require minimal program annotations), produce few false positives and rank bugs by importance, and finally, be as informative as possible to make it easier for developers to understand, reproduce, and fix bugs.

2 Static Program Analysis to Find Error-Propagation Bugs

One of the goals of my research is to use program analysis to improve software reliability. This work focuses on error handling in systems software and user applications. Bugs in software error handlers are some of the most pervasive, dangerous, and difficult to detect bugs. My dissertation [4] aims to automatically find error-propagation bugs in large software systems. Incorrect error propagation is a longstanding problem in many application domains, including systems software. C is still the preferred language for systems programming, and it does not provide explicit exception-handling support. Consequently the return-code idiom is commonly used in large C programs, including operating systems. Run-time errors are represented as simple integer codes, where each integer value represents a different kind of error. These error codes propagate through conventional mechanisms such as variable assignments and function return values. Despite having exception-handling support, many C++ applications also adopt the return-code idiom. Unfortunately, this idiom is error-prone and effort-demanding. My work applies static program analysis to understand how error codes propagate through software that uses the return-code idiom, and finds different kinds of error-propagation bugs. Many of these bugs can lead to serious problems such as system crashes, security vulnerabilities, data corruption, and unexpected results.

The main component of my framework is an interprocedural, flow- and context-sensitive static analysis that tracks errors as they propagate, with specializations for the systems under analysis. For example, the analysis recognizes high-level error-handling patterns found in Linux, which makes it possible to distinguish handled from unhandled error codes. I formulated and solved the error-propagation problem using weighted pushdown systems
A WPDS is a dataflow engine for problems that can be encoded with suitable weight domains, computing the meet-over-all-paths solution. Solving the WPDS reveals the set of error codes each variable might contain at each program point. I designed and implemented program analyses to find a variety of error-propagation bugs.

**Error Propagation Analysis and Dropped Unhandled Errors** [3, 5] This analysis focuses on finding error-code instances that vanish before proper handling is performed. Unhandled errors are commonly lost when the variable storing the unhandled error value (a) is overwritten with a new value, (b) goes out of scope, or (c) is returned by a function but not saved by the caller. I applied the analysis to five widely-used Linux file systems, including ext3 and ReiserFS, where the analysis found 312 confirmed error-propagation bugs and a critical bug in heavily tested code used by the NASA/JPL Laboratory for space missions. This work was done in collaboration with colleagues in the area of file systems.

**Defective Error/Pointer Interactions in the Linux Kernel** [7] Linux error codes are often temporarily or permanently encoded into pointer values as they propagate. Error-valued pointers are not valid memory addresses, and therefore require special care. Misuse of pointer variables that store error codes can lead to system crashes, data corruption, or unexpected results. This analysis finds three classes of error-valued pointer bugs in Linux file systems and device drivers: (a) bad pointer dereferences, (b) bad pointer arithmetic, and (c) bad pointer overwrites. The analysis uncovered 56 bugs in 52 different Linux file system implementations and 4 Linux device drivers.

**Error Code Mismatches Between Code and Documentation** [6] Inaccurate documentation can mislead programmers and cause unexpected failures. In this work, I considered whether the manual pages that document Linux kernel system calls match the real code’s behavior regarding returned error codes. This analysis finds the sets of error codes that file-related system calls return and compares these to Linux manual pages to find errors that are returned to user applications but not documented. The analysis uncovered hundreds of error-code mismatches for 52 Linux file-system implementations, and 42 file-related system calls.

All the analyses above [8] are scalable, precise, and informative. To achieve scalability, I devised two extremely effective optimizations that allow the analyses to run 24 times faster (under 5 minutes on average), while requiring 75% less memory. One of these optimizations consists of filtering out program variables that cannot possibly contain error codes. To achieve precision, I incorporated developers’ domain knowledge to reduce the number of false positives. Finally, I made the analyses informative by producing sample execution paths that demonstrate each bug reported.

To illustrate the generality of my analysis techniques, I also analyzed user applications such as the Mozilla Firefox web browser and the SQLite library, which is extensively used in widely-deployed applications such as Mozilla Firefox, Chrome, Skype, and Dropbox [4]. Firefox is written in C++, but uses the return-code idiom extensively. The feedback from developers is rewarding and motivating:

"I think this is an excellent way of detecting bugs that happen rarely enough that there are no good reproduction cases, but likely hit users on occasion and are otherwise impossible to diagnose." - Andreas Dilger (ext4)

"Ew, this [bug] is hard to figure out." - Matthew Wilcox (FS)

"Thank you for helping to improve JFS!" - David Kleikamp (IBM JFS)

"So that is a nice find." - Jan Harkes (Coda)

3 Dynamic Program Analysis of Numerical Programs

Another goal of my research is to apply program analysis to improve program performance. This work focuses on improving the performance of floating-point programs. Reasoning about floating-point programs is difficult given the large variety of numerical errors that can occur. One common practice followed by programmers without an advanced background in numerical analysis is using the highest available floating-point precision. While more robust, this can degrade program performance significantly. To alleviate this problem, I developed **Precimoniou**s, a dynamic program analysis tool to assist programmers in tuning the precision of floating-point
programs [9]. **Precimonious** systematically searches for a type assignment for floating-point variables so that the resulting program (1) produces an accurate enough answer given an error threshold, and (2) is faster with respect to the original program. Evaluation on several widely used functions from the GNU Scientific Library (GSL), the NAS Parallel Benchmarks, and other numerical programs shows performance improvements as high as 40%. We are in the process of developing alternative search techniques with the purpose of producing results closer to the global minimum.

Currently, I am also designing and implementing a framework for developing general-purpose dynamic program analyses. The main goal of this program-analysis infrastructure is to facilitate the development of dynamic analyses to improve the reliability of numerical programs. Our approach is based on shadow execution. We envision implementing several dynamic analyses targeting floating-point exceptions as well as symbolic execution. For example, we are working on formulating a dynamic analysis that aims to detect invalid exceptions and NaN (not a number) propagation, which can lead to problems such as infinite loops. This work is being conducted in collaboration with colleagues in the area of numerical analysis.

## 4 Future Research Agenda

The high-level goal of my future research is to broaden the impact of program analysis in three main aspects: (1) understanding and improving error handling through specification inference, testing, and language design, (2) expanding the use of program analysis to other less explored areas such as program performance and energy saving, and (3) developing new scalable, precise and informative general-purpose program analyses techniques.

### 4.1 Understanding and Improving Error Handling

The first step towards understanding error handling is to understand how errors propagate. In my dissertation [4], I designed and developed an error-propagation analysis, and found hundreds of error-propagation bugs in both systems software and user applications that use the return-code idiom. The next step consists of understanding how errors are handled. If an error is correctly propagated, we should ensure it is also handled correctly. To this end, there are many areas left to be explored to better understand and improve error handling. In the near future, I plan to explore inferring error-handling specifications in systems software, using these specifications to drive testing of error handling code, and proposing language support to prevent the introduction of error-handling related bugs in software.

**Inferring Error-Handling Specifications** Error handling in systems software is particularly interesting and challenging due to the size and complexity of these code bases. For example, Linux has millions of lines of code, and defines over a hundred run-time errors. Errors are propagated through long call chains, and error handling code is intertwined with the system’s main functionality. Furthermore, a same run-time error may be handled differently depending on the context. The code base is so large and complex, that it is impossible for a single developer to understand the whole system in detail, which makes it extremely difficult, if not impossible, to determine whether errors are handled correctly in each particular situation. I propose to use program analysis to automatically infer error-handling specifications. I plan to first focus on Linux file systems, for which multiple implementations are available. The first two challenges consist of distinguishing between handled and unhandled errors, and identifying error-handling code. My error-propagation analysis already solves the first problem, and I believe it can be used to identify error handling code. The remaining challenges consist of formulating how these specifications will look, and leveraging the multiple file system implementations to infer them. At the high-level, I believe these specifications will initially consist of a sequence of function calls.

**Testing Error Handling Code** Error-handling specifications can be used in a variety of scenarios. My main motivation behind inferring error-handling specifications is to have a better understanding of error handling, and use this knowledge to drive the testing of error-handling code in systems software. We will be inferring the specifications from the code we want to test, thus there is a chance these specifications might describe buggy error-handling patterns. I see the specifications as a starting point to provide an abstraction of how the code handles errors. We will drive execution to the parts of the program related to error handling, and use the
information captured in the specifications to re-create the context possibly by injecting errors. We will then check whether the expected behavior, e.g., error notification, is observed.

**New Error-Handling Language Constructs** Language designs for exception management have been under consideration for decades. Static verification of proper exception management has its own difficulties. For example, C++ exception throwing declarations are explicitly checked at run time only, not at compile time. Java’s insistence that most exceptions be either caught or explicitly declared as thrown is controversial. Frustrated Java programmers are known to pacify the compiler by adding blanket catch clauses that catch and discard all possible exceptions. Thus, while exceptions change the error handling problems in interesting ways, they certainly do not solve them. I would like to expand the scope of my work by studying error handling in modern programming languages that provide exception-handling mechanisms. I recently started looking into exception handling in Java programs [1]. My ultimate goal is to make programs more reliable by providing language support to avoid these problems in the first place. I believe it is time to learn from both successes and mistakes and significantly change the way we deal with program errors.

4.2 Improving Program Performance and Saving Energy

**Precimonious** constitutes a first step towards using program analysis to tune the precision of floating-point programs to improve performance. The following paragraphs briefly describe a few potential research directions in applying program analysis to improve program performance and save energy.

**Inputs for Numerical Programs** Precimonious relies on the user to provide a set of representative program inputs to be used during the precision tuning process. No guarantees are provided for untested inputs. In general, generating meaningful inputs for numerical programs is a difficult task because it requires a deep understanding of the program under analysis. Popular metrics used in testing such as code coverage are not sufficient when generating inputs for numerical programs. Input generation techniques in this context could have a large impact not only in precision tuning, but also in testing of numerical programs. I plan to leverage my expertise in dynamic test generation [2] to explore this problem in the near future.

**Precision Tuning Beyond Scientific Programs** Floating-point precision tuning is not just applicable to scientific programs. Programs in other domains can also potentially benefit from precision tuning. Some examples include GPU applications and machine learning. I plan to explore other areas in which precision tuning can impact both performance and energy consumption.

4.3 Designing Scalable, Precise, and Informative Analysis Tools

I see several research future directions towards making program analysis more effective in practice. Here are some areas I would like to explore in the near future.

**Scalable Tools** The key scalability limitation for today’s program analysis tools is memory. In previous work [5, 7, 6], I have achieved scalability by applying effective program optimizations. In collaboration with a colleague in databases, I recently developed a new program analysis technique [10] that leverages both memory and disk in a variety of configurations, dramatically reducing resource requirements. In the past, I also explored incrementality to speedup dynamic test generation [2]. In the near future, I plan to develop new program analysis techniques that run extremely fast, even in code bases with millions of lines of code, so that developers can afford to use them any time during software development. For example, the developer could run the analysis before check-in time, preventing the introduction of program errors earlier during the software development process. Researchers have developed various approaches to achieve scalability, e.g., the use of program summaries. In the same spirit, I would like to explore ways to reduce the amount of work to perform when applying program analysis. An interesting observation is that, although software changes rapidly, most code remains unchanged from one program revision to another. Thus, results from a previous analysis run could be updated accordingly rather than computed from scratch. This incremental approach presents interesting challenges in change impact analysis, how to store the analysis information, and how to use previous results to compute new analysis results.
Precise and Informative Tools  We need to invest more time in developing techniques to automatically infer domain-specific knowledge from programs to develop more precise and informative tools. My proposed work on automatically inferring error-handling specifications touches on this (see section 4.1). However, I also believe we should exploit developers’ domain knowledge to make tools more precise. In this regard, I would like to collaborate with colleagues in human computer interaction to study what is the most effective way to gather domain knowledge from users, and incorporate it into analysis tools to improve their precision. For example, developing interactive program analysis tools that let users specify program specifications using natural language, or graphical interfaces. I would also like to explore ways to make tools more informative. For example, what diagnostic information an analysis should produce, and what is the best way to present it to the user. In previous work [5, 7, 6], my analyses produce program traces that illustrate how error codes propagate in a format that facilitates the navigation of relevant source code. Some ideas to make analysis tools more informative include using existing or new visualization techniques and providing ways to easily navigate both the source code and the program documentation. In general, I believe program analysis tools often lack good program interfaces, making them less useful and attractive to developers.

References


