Research Statement

Baris Kasikci

Building correct, secure, and efficient software is a complex endeavor, given the new architectures, platforms, and interfaces that emerge on a regular basis. This complexity makes it hard for developers to write good software, and thus nullifies some of the benefits of new technology trends. *My research objective is to help developers maximally reap the benefits of technological advances by aiding them in quickly writing good software.*

In the future, I will work on techniques to address problems such as security vulnerabilities and privacy issues, which have become more relevant than ever due to the surge of large-scale distributed applications. I will also build techniques to help developers cope with software and system design challenges of computing platforms that are becoming mainstream such as Internet of Things and heterogeneous computing systems.

In my dissertation, I worked on addressing the challenges posed to software development by a prominent trend of the past ten years, namely the shift of microprocessor design from single-core to multi-core architectures. As hardware became increasingly parallel, concurrency became mainstream to build software that leverages parallel hardware. However, the transition to multi-core hardware happened at a more rapid pace than the evolution of associated techniques and tools, which made it difficult to write programs that employ concurrency and are both efficient and correct. Concurrency bugs are often hard to reproduce and fix, and can cause massive losses. I experienced the difficulties of concurrency firsthand as a software engineer designing and developing concurrent real-time software for four years prior to my Ph.D. *In my dissertation, I developed techniques for the detection, classification, and root cause diagnosis of bugs, with a particular emphasis on concurrency bugs.* Some of the tools and techniques I developed are being used in major technology companies such as Microsoft and Intel.

In the rest of this statement, I will first summarize my dissertation research, and then I will explain my future research directions.

Dissertation Research

In my dissertation, I designed and implemented: *RaceMob* [4, 6], the first high-accuracy data race detector that can be used always-on in production; *Portend* [5, 7], the first high-accuracy data race classifier that can predict the consequences of data races under various memory models; *Gist* [2, 3, 8], the first technique to accurately determine root causes of in-production failures without relying on custom hardware or runtime checkpointing. Many of the techniques I developed require efficient runtime instrumentation and sampling. In this regard, I also developed *Bias-Free Sampling* [1], a technique that allows sampling rarely executed code—where bugs mainly reside—without over-sampling frequently executed code. In the next four sections, I briefly describe each project in more detail. I then talk about the approach I take to problem-solving.

RaceMob: Data Race Detection

Data races are among the worst concurrency bugs, having caused massive material losses and losses to human lives, and thus, many techniques have been developed to detect data races to eventually fix them.

However, data race detection that is both accurate and efficient has long been an open problem, mainly because of the tension between accuracy and efficiency: accurate data race detection requires gathering runtime execution information, which hurts efficiency. In particular, purely *static* data race detectors work offline without access to the runtime execution information (e.g., variable addresses). Therefore, static detectors are efficient (i.e., don't incur runtime performance overhead) but have low accuracy. On the other hand, purely *dynamic* data race detectors monitor execution information, therefore they can accurately detect data races, but they incur a lot of runtime performance overhead.

Furthermore, some data races only manifest in production, which makes it more challenging to efficiently detect them without perturbing user experience. Prior work did not attempt to detect data races in-production, where the challenges of data race detection are exacerbated.

In order to solve the in-production data race detection problem, I developed RaceMob [4, 6], a technique that combines in-house static program analysis with in-production dynamic analysis. RaceMob first statically

detects *potential* data races, and then dynamically validates them using a dynamic data race detection technique. RaceMob's static data race detection is complete: it does not have false negatives (i.e., it does not miss true data races). However it has many false positives (i.e., reports that do not correspond to real data races), but that is compensated by RaceMob's real-user crowdsourcing that dynamically determines whether the potential data races are real data races. RaceMob's mixed static-dynamic data race detection achieves low overhead and a high degree of accuracy.

In my RaceMob work, I showed that we can relax the requirement of always tracking synchronization operations for accurate data race detection by carefully determining points in the execution where tracking should start and end. Combined with crowdsourcing, this deviation from conventional wisdom allowed RaceMob to lower the runtime overhead of data race detection while remaining accurate.

Using RaceMob's mixed static-dynamic approach and its novel dynamic data race detection scheme, I showed that we can perform data race detection that is more accurate than commercial data race detectors (e.g., Google ThreadSanitizer) with negligible overheads of around 2%. Such low overheads make in-production data race detection feasible, thereby allowing developers to fix data races and pave the way to more reliable and secure concurrent software.

Portend: Data Race Classification

Another long-standing problem in concurrency has been the accurate identification of the consequences of data races to classify them in terms of their severity. Modern software has a lot of data races, therefore developers need to prioritize the fixing of data races to efficiently use their time[1].

I showed that the abstraction level of the classification criteria that prior work employed harmed classification accuracy. I then developed Portend [5], the first technique to accurately classify data races based on their consequences. Portend is a data race classifier that explores multiple paths and schedules of a program to accurately identify the consequences of data races. Unlike what prior work suggested, I showed that low-level effects of data races on programs' memory state is not an appropriate criterion for classification, and looking at higher-level effects of data races such as externalized program state yields up to 89% higher accuracy.

I also explored the effects of the memory model (such as instruction reordering) on data races. These effects can be caused by multiple layers in the system stack such as the hardware (store buffer reordering) or the compilers (instruction reordering). I developed symbolic memory consistency modeling [7], a technique that can be used to model various memory models (e.g., weak memory) in a principled way in order to perform data race classification under those memory models.

Gist: Root Cause Diagnosis of Bugs

Detecting and classifying concurrency bugs allows developers to discover the bugs that exist in their programs, and understand the priorities of fixing these bugs. However, in order to fix a bug, developers need an explanation of how the bug manifests, that is, how their program reaches the failure (e.g., what are the failure-inducing inputs, thread schedules, etc.). This explanation is useful, given that developers traditionally seek such an explanation when debugging a program by reproducing failures to determine their root cause. Determining the root cause becomes even harder if failures recur only rarely in production.

I developed a technique called Gist that produces a high-level execution trace called the *failure sketch*. A failure sketch includes statements that lead to a failure and highlights the differences between the properties of failing and successful program executions. I showed through a series of experiments on real-world software (e.g., the Apache web server) that these differences effectively point developers to the root causes of failures, and that Gist can identify these differences accurately and efficiently: Gist built failure sketches with 96% accuracy with an average runtime performance overhead of below 4%. Although I mainly focused on concurrency bugs in my evaluation (because finding the root causes of such bugs is very hard), Gist is applicable to non-concurrency bugs too.

[1]Some language semantics (e.g., C++) consider all data races as bugs, whereas others don't (e.g., Java). Regardless, modern concurrent software is ridden with data races, and developers need to better understand the consequences of data races.

I transferred some of Gist's technology to an Intel product during my 2015 internship at Intel. For this, I built a tool that uses static analysis to determine which statements operate on variables of a given data type during a failing run. I showed that this tool reduces by an order of magnitude the number of statements a developer needs to look at while debugging. This tool is now being used and maintained at Intel.

Bias-Free Sampling

Many of the techniques I developed rely on gleaning execution information from in-production runs, which is challenging to do without undue runtime performance overhead. Most of the time, execution information of interest (e.g., buggy statements) resides in cold code (i.e., rarely executed code), and therefore, monitoring and sampling cold code is particularly useful. Alas, which parts of the code are cold is not known a priori, therefore it is challenging to sample cold code without over-sampling hot code.

To overcome the challenge of sampling cold code efficiently, I developed a technique called Bias-Free Sampling (BfS) together with researchers from Microsoft Research. BfS allows sampling the machine instructions of a dynamic execution independently of their execution frequency by using breakpoints. The BfS overhead is, therefore, independent of a program's runtime behavior and is fully predictable: it is merely a function of program size. BfS forms the foundation of dynamic monitoring in my root cause diagnosis work, which uses hardware breakpoints for a component of its dynamic analysis.

BfS is very efficient: it incurs 1–6% overhead for all the Windows 8 System Binaries. My work on BfS had an impact on Microsoft products. Using BfS, we built a coverage measurement tool for both native and managed (i.e., C#) code, which is now used internally at Microsoft.

Approach to Research

I have taken a collaborative and interdisciplinary approach when developing the techniques in my dissertation. I collaborated with researchers and engineers from Microsoft and Intel for my work on Gist and BfS. I combined techniques from programming languages, operating systems, computer architecture, and software engineering to solve systems problems. Looking back at the work I have done so far, there are three principles that shaped my approach to problem solving:

First, I take a systems-oriented approach. This means that I take a holistic view of a given problem, identifying the constituents of the problem at multiple layers of the system stack. In my research, this approach helped me develop techniques to deal with concurrency issues, which appear in many layers of the system stack, from parallel hardware to language memory models to operating system schedulers and to applications.

Second, I frequently employ techniques from programming languages, and in particular, I rely on static and dynamic program analysis. I primarily focus on finding a sweet spot in the balance between static vs. dynamic program analysis to develop efficient and accurate techniques. Static analysis tends to be efficient but inaccurate, whereas dynamic analysis tends to be inefficient but accurate, and a carefully designed mix can be both accurate and efficient. In my research, this mixed approach helped me strike a balance between static analysis, which is done in house and thus can afford heavyweight techniques, and dynamic analysis, which is done in production with minimal runtime performance overhead.

Third, I don't shy away from unconventional approaches. In my research, adopting an unconventional approach and rethinking the assumptions of prior work allowed me to solve some key open problems, such as low overhead data race detection and accurate data race classification. For example, as opposed to common belief, I showed that it is possible to accurately detect data races without tracking synchronization operations all the time.

Future Research Plans

In the future, I plan to continue working on helping developers build more reliable, secure and efficient programs. In particular, I will focus on system security and privacy, primarily in the context of large-scale distributed systems. I will also work on helping developers cope with the challenges of software and system design in the context of Internet of Things and heterogeneous computing systems. Below, I describe my short term and long term research goals.

Security Today, one of the most challenging problems in computer systems is building secure software. I am currently extending my work on root cause diagnosis to encompass security vulnerabilities. In the short term, I plan to look into using hardware support for sampling execution path profiles from user executions to detect control flow hijack attacks. In the long term, I would like to answer more general questions such as: Can we identify *good* execution paths versus *bad* execution paths (e.g., ones that lead to security vulnerabilities and failures) using techniques from machine learning? Can we take this approach further and automatically infer properties (e.g., performance behavior) about paths relying on statistical techniques and help developers better structure code based on such properties? What are the meaningful boundaries of programs to monitor when gathering path information? Can we have intelligent strategies to sample paths of programs (e.g., strategies that do better than random sampling)? I believe that there are a lot of similar hard and interesting questions to answer in order to enable developers to build secure software.

Privacy Some of the techniques I developed rely on gathering execution information from users, and therefore, have privacy implications. To improve the privacy of users, I initially intend to work on techniques to quantify and limit the amount of execution information extracted from user endpoints. In the long term, I would also like to work on techniques with strong privacy guarantees to anonymize the execution information. For example, one can compute an anonymous *signature* describing the control flow of an execution to provide better privacy. However, it remains to be seen what are the right boundaries (i.e., within the code) for computing such signatures. Effective computation of signatures in the presence of concurrency and non-determinism is also an open question.

Large-scale distributed systems Many of the problems I attacked in my dissertation have incarnations in large-scale distributed systems (e.g., Internet-scale systems). For instance, data races and atomicity violations manifest as process-level races that cause correctness and performance problems as well as resource leaks in distributed systems. I would like to adapt my techniques for the detection and root cause diagnosis of bugs in the context of large-scale distributed systems.

Internet of Things The rising popularity of Internet of Things (IoT) will pose unprecedented challenges for developers. In particular, IoT is about integrating networked devices with *things* around us: the watches we wear, cars we drive, electrical appliances we use, etc. Developers will be faced with the challenge of building software for such highly interconnected and distributed systems. For instance, debugging the software in IoT devices will have to be done in production, because such devices may not be directly accessible by developers. I plan to leverage the techniques I developed in my dissertation such as in-production bug detection, to address the challenges of software development for IoT devices. Moreover, the challenges associated with IoT also present unique opportunities. For instance, the ubiquitous nature of IoT will allow amassing a large amount of data that can be used to help developers improve the quality and efficiency of their programs. Gleaning execution information from such resource-constrained environments will require combining various techniques from program analysis and computer architecture, much like the combination I relied on in my work on root cause diagnosis of software failures.

Heterogeneous Computing Heterogeneous computing systems that mix traditional CPUs with GPUs, FPGAs, and ASICs are becoming increasingly relevant. Heterogeneous systems come with a whole new set of challenges for developers such as non-uniformity in system development, complexities due to co-presence of a variety of programming interfaces, and unpredictable performance and energy behavior. Tackling the challenges of software development for heterogeneous systems will require rethinking the system stack and abstractions. I plan to leverage my interdisciplinary approach to attack these imminent challenges in heterogeneous computing.

References

[1] B. Kasikci, T. Ball, G. Candea, J. Erickson, and M. Musuvathi. Efficient tracing of cold code via bias-free sampling. In *USENIX Annual Technical Conf. (USENIX ATC)*, Philadelphia, PA, June 2014.

[2] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Symp. on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[3] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, M. Musuvathi, and G. Candea. Failure sketches: A better way to debug. In *Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[4] B. Kasikci, C. Zamfir, and G. Candea. Cord: A collaborative framework for distributed data race detection. In *Workshop on Hot Topics in Dependable Systems (HotDep)*, Hollywood, CA, October 2012.

[5] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: Telling the difference with portend. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

[6] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced data race detection. In *Symp. on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[7] B. Kasikci, C. Zamfir, and G. Candea. Automated classification of data races under both strong and weak memory models. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, May 2015.

[8] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea. Automated debugging for arbitrarily long executions. In *Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Ana Pueblo, NM, May 2013.