RESEARCH STATEMENT

Du Li (duli@cs.cmu.edu)

My primary research interest lies in advancing the quality of software by using program language and runtime system approaches. In particular, I am excited to address the program reliability and security challenges brought by the prevalence of multi-core systems. My research mission is to exploit underlying principles of concurrent programming to enable cost-effective approaches to identify program defects, and to automate the process of fault elimination.

Computer systems are becoming more integrated into the fabric of our society. Given society’s reliance on such systems, it is increasingly important that they are perpetually available and failure-free. Meanwhile, multi-core systems are becoming pervasive from high-end servers to low-end portable devices. Multi-threading is the widely adopted way to fully utilize the computation power of multi-core infrastructure. However, it is a well accepted fact that concurrency faults can be expensive to detect and fix in multi-threaded programs. Despite significant advances in validation and verification techniques over the past decades, such faults are still commonplace even for well tested systems. For examples, a race-related fault in a well-tested Nasdaq auction system delayed the Facebook IPO, resulting in financial losses of more than $60 million in 2012. The Blackout of the Northeastern US in 2003 also caused billions of damages and a loss of lives.

We can envision this type of incident will be likely to keep emerging in the future due to the fact that efficiently detecting and fixing concurrency faults is still an open research question. Worse, it has been confirmed that concurrency faults can also compromise software security and make the system vulnerable. Fundamentally, concurrency faults are challenging because they are sensitive to thread inter-leavings, therefore they can go undetected during the in-house testing phase and then appear in the field. A major part of my research vision is aiming at improving the security and reliability of deployed multithreaded software by dynamically removing software defects due to concurrency faults.

My research approach is to study real systems, observe results, and then understand the underlying principles. I believe that it is important to build and deploy real systems, and evaluate proposed solutions on them, which helps me uncover new insights and identify practical problems. As part of my dissertation, I have built race detectors within an industrial grade Java Virtual Machine, Jikes RVM. During my internship at Oracle Labs, I built a program analysis framework and it has been released with Maxine VM. I believe good research work should be based on a solid theoretical foundation. Towards that end, I take a unique approach to improve the quality of software by combining runtime system infrastructure with program analyses, compilers and type theory.

Currently, dynamic concurrency fault detectors suffer unacceptably high overhead, making them infeasible to apply in real-world software systems. My research bridges the gap between the expectation and the practice for concurrency fault detection. In particular, my work investigates strategies for reducing the overhead of race detection leveraging spatial and temporal dimensions of program behavior. In the spatial dimension, we filter out objects that cannot be involved races. In the temporal dimension, we identify the period of time when a method likely participates in races by taking advantage of the runtime information such JIT compiler activities. Further, we are extending our work to the program security area.

Current work

1. Optimization in the spatial dimension of program behavior

1.1 Reduce race detection overhead with stationary analysis

Data races are the most common concurrent bugs, involving two improperly synchronized accesses, one of which is a write. Existing dynamic race detection approaches are prohibitively expensive for deployed systems due to the fact that there are a vast number of memory accesses in a program for race detectors to monitor. In an object oriented program, the lifespan of an object consists of three phases: the first phase is initialization, in which the object can only be accessed by the creator thread, that is, the object is thread local; the second phase starts at the point the object becomes being shared among multiple threads and ends when any thread writes to the object; the last phase is from a thread’s write through the end of the object’s life. Clearly, a race cannot occur during the first and the second phase as the former is thread local and the latter does not have a write. After careful study, we find that a
large number of objects will stay in the second phase forever, we call these objects stationary objects. These objects become read-only after initialization, so they cannot never be involved races. This finding opens up an opportunity for race detectors to skip monitoring accesses to stationary objects without missing data races, while greatly reducing the overhead of race detection. We build a race detector, SOS, based on this insight and it detects up to six times more races in comparison to a state-of-the-art race detector. This paper about SOS appeared at OOPSLA[1].

1.2 Reduce race detection overhead with loop based sampling

Existing sampling-based approaches do not sample uniformly across a program. Conceptually, their approach seeks to uniformly sample time slices during which they monitor for data races. Most programs consist of loop regions. When a program has long running loops, race detectors end up sampling more within those loops, because the program spends more time in those loops, which leads to increased detection overhead, but without a corresponding increase in detection effectiveness. We also discover that when a race exists in a loop, it is detected again and again so monitoring these operations does not necessarily reveal new sources of races. This insight provides an opportunity to reduce the race detection overhead without degrading race detection effectiveness by appropriately reducing monitoring costs within loops. We introduce Loop Iteration Sampling (LIS), a new detection optimization that spends less time to monitor repetitive operations that occur inside loop regions. The algorithm employs a differential sampling approach to significantly and continuously reduce the monitoring efforts in long running loops. When we apply LIS to two existing sampling-based race detectors, Pacer and SOS, we see a 52% and 33% reduction in overhead, respectively. LIS also increases the numbers of detected races by a factor of seven for Pacer and a factor of two for SOS, when we restrict the race detection overhead to less than 100%[2].

2 Optimization in the temporal dimension of program behavior

2.1 Identify timing of races with runtime information

We also investigated the distribution of races in the temporal dimension of program behavior. Our work continues in the spirit of optimization within runtime systems by connecting program behavior with runtime information. We observed that races were appearing very soon after JIT compilation. In other words, a method will race soon after it get compiled by JIT compilers if it ever does. This strongly suggests that baseline compilation activities can serve as an effective trigger for enabling race checking, when the JIT compiler becomes active, and later disabling it, after a certain amount of time when there is no compiler activity. We implemented the feature in a race detector and it can remove about 50% percent of monitoring effort with a modest loss of race coverage.

2.2 Identify timing of races with hybrid analyses

Race detection can be categorized as two classes of technologies. One is the static approaches and the other is dynamic ones. Static approaches are generally complete, that is, they rarely miss races, but they suffer from false positives. In contrast, dynamic race detectors can ensure soundness, but their runtime overhead is high. We explore a hybrid mechanism that can get the best of both worlds.

As stated, we use the runtime information to locate the time window in which a method, say $\tau$, likely involves races. We use static analyses to identify a group of suspicious methods that may race with method $\tau$ if a method satisfies two conditions: 1) a method can execute in parallel with method $\tau$; 2) a method accesses the same shared data that method $\tau$ does. We build a framework to pass on the static analysis results to dynamic race detectors so that race detectors only need to monitor the suspicious methods during a certain time periods. This project is in progress, but we have already gotten promising preliminary results.

Other research contributions

Memory safety for multilingual programs

The use of the Java Native Interface (JNI) can bypass Java boundary checking and exception-handling mechanisms. Furthermore, its use can violate Java’s type safety feature because of the type mismatches between native programs and Java programs. As a result, this integration can cause various security issues, including heap errors that can be dangerous and difficult to detect.

We propose Quarantine, a runtime system that can identify objects accessible by native methods and then migrate these objects to a quarantine area, which is used specifically to host this type of object. The goal of Quarantine is
to create a runtime platform that allows programmers to apply existing heap protection techniques that have been designed for native languages but do not work well in the Java domain. This work appeared at PPPJ in 2011[3].

Future work

Security and reliability for emerging computing environment

Currently, emerging computing environments such as cloud computing and mobile systems are becoming widely used. I would like to focus my research on improving the dependability of software in these environments. In particular, I am interested in various security challenges in these emerging computing environments. There are clear connections with my previous work especially on how the combination of program analysis and runtime system support can be used to isolate and repair anomalies due to security violations. Recently, studies have shown that concurrency faults can lead to exploitable security vulnerabilities. I have started exploring program language and runtime techniques to make concurrent systems more secure and resilient. We plan to extend race detectors by tapping the knowledge of security properties so that the they can automatically distinguish vulnerable races.

In addition, fixing security vulnerabilities is a time sensitive task. So far, most repair techniques involve a considerable amount of human involvement (e.g., analyzing faults and synthesizing patches), which can be unreliable and untimely. We are working on a framework to automate the vulnerability repair procedure. It includes three components: 1) an efficient and precise race vulnerability detection tool, 2) a program analysis and synthesis technique that can automatically generate security patches, 3) a transparent patching mechanism based on the dynamic compilation infrastructure in runtime systems. Our goal is to provide a framework that can detect, defend and fix concurrency security attacks in a fully transparent and instant way.

I participated in writing two proposals based on this idea. One is National Security Agency sponsored Science of Security Lablet seeding project, and the other is National Science Foundation sponsored Secure and Trustworthy Cyberspace project. The former has been funded in 2013, and the latter is under review.

Program models for parallel and concurrent programs

In essence, generic programming languages like Java do not provide sufficient concurrency safety. As such, applications written in such languages are prone to concurrency errors such as data races, atomicity violations and deadlocks that can be difficult to detect, isolate, and repair. Meanwhile existing programming models do not provide an efficient way to utilize the strength of multi-core systems. My future research focus is to evolve programming models to address these issues with multi-core systems. I am interested in working on theoretical and practical aspects of compilers and runtime systems to improve performance and reliability of parallel programs. I would work on formal methods, like type systems and verification, or program language design and implementation, like parallelism and domain specific languages.

Interplays between runtime systems, OS and architecture

I strongly believe that better hardware and software interactions or co-designs are critical to scalable system performance in the multi-core era. Different system layers have different runtime information that can make execution of other software or hardware layers more efficient. Compilers and runtime systems have high level program semantics information that can be used by the operating system or hardware to better manage resources. In the past, my research group has leveraged lock usage information that is accessible inside a Java Virtual Machine to eliminate much of lock contention that occurs due to poor thread scheduling. Currently, we are exploring an approach to improve efficiency of the last level cache by making use of memory management information. It is promising and exciting for me to investigate how to enable two-way communication and coordination between compilers, operating systems, and hardware to improve the efficiency and dependability of parallel programs.

References

[1] D. Li, W. Srisa-an, and M. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. OOPSLA ’11

http://cs.cmu.edu/~duli