Research Statement

Suriya Subramanian

June 11, 2009

My research is in the broad area of software reliability and availability. As society increasingly relies more on software, we require several computer systems such as cell phone base stations, nuclear plants, banks, financial exchanges, and online retailers to operate continuously. These systems are not immune to software updates that are required to fix bugs and add new features. Keeping systems available in the face of software updates is an important research challenge.

Dynamic Software Updating

My work towards my dissertation has been in *Dynamic Software Updating* (DSU) [1]. DSU aims to avoid application downtime by supporting *on-line updates*. A DSU system is able to update a running application to a new version by transforming state to conform to the updated version and executing new code after the update. In our work, we focussed on DSU for managed languages as these languages are becoming increasingly prevalent. The key contribution of our work is achieving DSU by leveraging existing services provided by a managed language virtual machine. We show that DSU is viable by supporting over two years worth of updates to three open-source server applications. Our system JVOLVE is the most-featured best-performing general-purpose DSU system for managed languages.

Jvolve VM We built JVOLVE on top of Jikes RVM, a Java-in-Java research virtual machine by extending the following VM services. JVOLVE modifies the VM's thread scheduler to suspend application threads to perform an update. JVOLVE avoid type errors due to timing of an update by stopping threads at DSU safe points that expand on VM safe points, where it is safe to perform garbage collection (GC) and thread scheduling. DSU safe points further restrict what methods may be active on each thread's stack, depending on the update. JVOLVE also installs return barriers that trigger an update when all unsafe methods have popped off the activation stack. JVOLVE makes a substantial advancement over prior work by supporting updates to several categories of active methods on stack. For this, JVOLVE makes use of on-stack replacement.

JVOLVE makes use of the garbage collector and JIT compiler to efficiently update data and code associated with changed classes. JVOLVE initiates a whole-heap GC to find and update existing object instances to conform to their new version. The adaptive compilation system naturally optimizes updated methods further if they execute frequently. JVOLVE imposes no overhead during steady-state execution, and update time is roughly equal to that of a full heap garbage collection. The zero overhead in steady-state execution is a significant improvement over prior work.

Evaluation. We assessed JVOLVE by applying updates corresponding to one to two years' worth of releases of three open-source multithreaded applications: Jetty web server, JavaEmailServer (an SMTP and POP server), and CrossFTP server. We showed that JVOLVE could successfully apply 20 of 22 updates—the two updates it does not support change a method within an infinite loop that is always on the stack. Applications updated by JVOLVE enjoy the same steady-state performance as on a standard VM and as if started from scratch.

Hybrid Dataflow Graph Execution in the Issue Logic

Prior to JVOLVE, I worked in compiler architecture. To improve the scalability of conventional superscalar instruction windows, I proposed Hybrid Dataflow Graph Execution (HeDGE) in the Issue Logic [2]. HeDGE explicitly maintains data dependences between instructions in a superscalar instruction window. HeDGE's explicit encoding allows the window to be constructed using Random Access Memory (RAMs) instead of

Content Addressable Memory (CAM). This simplifies the wakeup logic, improving overall processor power, energy, and energy-delay.

Future work

In the future, as we continue increasing our reliance on software, I will continue working towards robust software systems. I intend to continue work on Dynamic Software Updating with the aim of pushing it closer to mainstream adoption. I also intend to focus on making dynamic language systems better.

The best of both worlds: Blending static and dynamic languages

Static and dynamic languages have well-known strengths and weaknesses. Static languages such as Java and C# guarantee compile-time type safety, allow easier and better static analysis, and provide high performance using modern VM technology. Dynamic languages such as Python and Ruby are increasingly being used in a wide variety of environments these days. These languages are elegant and expressive, fostering programmers to write concise programs faster. These programs are also easier to read and understand. However, these languages do not guarantee compile-time type safety and force developers to rely on unit tests to improve correctness. Also, these languages are typically interpreted and hence run far slower than static languages.

We desire the productivity benefits of dynamic languages combined with the safety and performance of static languages. We can achieve this with a language design staying close to dynamic languages targeting high performance VMs. In doing so, we have to ensure two key considerations. 1) There is a lot of code out there written in dynamic languages and we need to either maintain backward compatibility or provide an easy migration path to the new language. 2) Several years of research and development has resulted in high quality VMs. We should not duplicate VM research, but look to target existing VMs with as little modification as possible.

I strongly believe that research in programming languages should be guided by intuition gained from writing and understanding programs. From my experience in writing Python programs and building web applications I have observed that most parts of an application do not require or employ fundamental dynamic features such as metaclasses, monkey patching or the use of `eval`. An application can be viewed to be composed of parts that are dynamic and those that are not. With negligible productivity loss, programmers should be able to write most parts of an application in a non-dynamic subset of the language.

We can improve type safety by combining of static type inferencing and programmer annotations for non-dynamic parts of the program, with run-time type checking for the rest. Similarly the common-case non-dynamic parts of an application can be optimally compiled to machine code like in a conventional VM, while still employing a slower implementation using reflection for the less-common dynamic features. We seek a middleground between dynamic language design and static language VMs to get the best of both worlds.

In conclusion, my research will produce analyses and systems that improve the reliability, availability and robustness of future software systems.

References

[1] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), To appear*, Dublin, Ireland, 2009.

[2] Suriya Subramanian and Kathryn S. McKinley. HeDGE: Hybrid Dataflow Graph Execution in the Issue Logic. In *HiPEAC*, volume 5409 of *Lecture Notes in Computer Science*, pages 308–323. Springer, 2009.