

RESEARCH STATEMENT FOR MARIA JUMP

My experiences as a graduate student span two different areas. After spending a couple of years working in computational geometry with Nina Amenta, I settled into research in Programming Languages with Kathryn McKinley. My experiences working on protein crystallography analysis and 3D SuperResolution exposed me to large, memory-intensive programs. During these early days, I spent many hours understanding these programs and how they could be made to use memory more efficiently. My dissertation addresses program understanding and debugging by introducing low-overhead dynamic heap analysis techniques. In this statement, I summarize the contributions of my dissertation and outline my plans for future work.

Dissertation Summary

As the demands on computer systems and software increase so does their sophistication and complexity. To manage this increased complexity, programmers are increasingly turning to managed languages because of the software engineering benefits they provide. The Tiobe Programming Community Index shows that popular languages like C# and Java have grown in popularity since 2001. Managed languages provide features that allow large teams of programmers to collaborate and produce higher quality software faster. Large teams of developers mean that few, if any, understand the entire system making semantic, memory, and concurrency bugs harder than ever to find.

One class of program bugs are heap-based, resulting from the memory model and pointer discipline. Notoriously difficult to detect, heap-based bugs include both memory and semantic errors. Memory errors, many of which are eliminated by automatic memory management, persist in two forms: (i) leaks --- systematically neglecting to null pointers after their last use which can cause the program to exhaust memory --, and (ii) null pointer dereferences. Semantic errors manifest as malformed pointer-based data structures. While dereferencing a null pointer has an immediate effect, most semantic bugs have a delayed result, and often times only become apparent after significant damage has been done to the heap.

Previous work for detecting heap-based anomalies focuses on detection during the development and/or testing phases. Many of the prior tools are only useful during testing since they significantly slow down programs, require special hardware, and/or increase memory consumption by 75% or more. One technique for offsetting these overheads is to offload processing work to a different core in a multicore system. Unfortunately this does not work in systems where memory overheads are high and distinct cores compete for shared resources. Additionally, while software testing does improve software quality by exercising a wide range of program functionality, exhaustive testing is infeasible due to the size of the testing space. The result is that software ships with bugs. Fortunately, the presence of a managed runtime can be leveraged to detect heap-based bugs in deployed software as well as assist in understanding a program's heap-usage during development and testing.

My dissertation begins an exploration of how to exploit the runtime to heighten program understanding and to ease identification of heap-based bugs by introducing two synergistic techniques for gathering heap statistics at very low overhead. The first technique, the *dynamic summary graph* (DSG), compactly summarizes the heap during program execution. By exploiting the underlying runtime and piggybacking on the object scan performed during garbage collection, DSGs can be built by adding less than 1% to total allocation and less than 4% to total time on average. The DSG is one of a family of graphs which summarize the dynamic nature of the heap; not only the objects that comprise the heap but also the relations between them. By capturing points-to relations in the edges, we show ownership properties. The lack of an expected ownership relation is often evidence of potential semantic errors. Points-from relations in the edges allow us to identify why a particular class of objects is considered alive. Heap summaries are most cheaply generated by class, but level of summarization is often too coarse grain. Fortunately, by applying the second technique -- *dynamic object sampling* (DOS) -- to selectively tag objects, the heap can also be summarized by context or even data structure.

Summarization by context or data structure is more difficult since managed runtimes do not correlate objects with their context or data structure. One approach to teach objects their context, for example, is to add space to the object's header resulting in larger objects, more frequent garbage collections, larger program footprints, and slower runtimes -- adding a single word increases the the space overhead by 5-7% and the time overhead by 8-18% on average. Dynamic object sampling (DOS) provides a method for tagging some objects while leaving others unchanged [3]. For some applications, selecting which objects to tag is obvious (nodes of a recursive data structure); for others, it is more challenging. By focusing on fewer objects in the heap we can significantly reduce the overheads of tracking objects characteristics (sampling 6% of objects results in less than 1% space overhead and less than 3% time overhead).

To show the effectiveness of these techniques, we introduce two separate applications of the DSG. The first, Cork, dynamically detects systematic heap growth while using the simplest type of DSG, a class points-from summary graph [1]. The second summarizes both points-to and points-from relations by data structure in an arity summary graph to identify statistical anomalies and to show a correlation between these statistical anomalies and semantic errors in the program's data structures. Next, I will describe these two applications.

Cork is an accurate, scalable, and low-overhead memory leak detection tool for managed languages. Cork uses the simplest type of DSG, a class points-from summary graph (previously called type points-from graph or TPFPG), to summarize, identify, and report systematic heap growth. We show that the TPFPG provides both efficiency and precision. The nodes of the TPFPG represent the volume of live objects of each class. The edges represent the points-from relationship between types weighted by volume. At the end of the each collection, the TPFPG completely summarizes the live-object points-from relationships in the heap. Comparing the TPFPGs over time produces a dynamic slice which identifies the classes contributing most significantly to the systematic heap growth and, by following the points-from edges, identifies the class of the growing data structure. We show that Cork efficiently provides us with enough information to detect systematic heap growth. Cork successfully identifies the only two SPECjvm benchmarks with systematic heap growth as well as identifying the cause of a well-known memory leak in SPECjbb2000, something its developers had not previously done. Finally, Cork positively identified the cause of a documented memory leak in Eclipse allowing for a correction to be submitted back to the community. Cork's accuracy, scalability, and efficiency make it the first system of its kind with overheads low enough to use in deployment.

To study **data structure anomalies**, the DSG takes on a new form, an arity summary graphs (ASG), that summarizes the shape of data structures. In order to gain understanding of the shape of individual data structures, we separate objects by data structure using dynamic object sampling and summarize object arity by data structure. Arity is defined as the number of references pointing to an object (the indegree of the object) and the number of non-null references it has (the outdegree of the object). The nodes of the ASG represent the number of objects of each class with each arity value (e.g., the percentage of **Node** objects whose outdegree=0, are leaves, in a binary tree). The edges in the ASG represent either points-to or points-from relations between objects. At the end of each garbage collection, the ASG summarizes each data structure in a separate summary graph. I am currently working on exploring the usefulness of the ASG for (i) program understanding by studying how the ASG changes over time, (ii) data structure anomaly detection by dynamically checking of arity statistics to previous collected values, and (iii) dynamic verification of ownership invariants. During testing, ASG statistics can be combined from multiple executions to discover heap properties that are stable from different executions of the same program and those properties that vary across every execution. For example, the ASG can identify whether an object which was intended to have a single owner (indegree=1) is ever owned by another object. Additionally the ASG, which maintains class information, can distinguish between iterators and other classes of objects. Profiling to generate statistical invariants during testing and focusing our analysis during the garbage collection allows the ASG to be used to verify arity relationships during production runs.

Future Work

Managed languages such as Java and C# are becoming more and more pervasive due to the software engineering benefits they offer, including dynamic class loading and instantiation of interfaces; polymorphism; and automatic memory management. While these features allow programmers to develop software faster and with fewer errors, they also make achieving high performance and performing program/performance analysis harder. Fortunately, the presence of an underlying runtime provides an opportunity. My research focus is to continue to explore these opportunities and how they can be used to improve program understanding and performance.

While my dissertation shows how the dynamic summary graph can be used to detect heap growth and data structure anomalies, I believe that these techniques can be extended to look at other aspects of the heap. For example, combining the DSG with a write barrier would allow us to monitor data structure changes and allow dynamic detection of candidates for synchronization elimination. Additionally, I would like to explore how the just-in-time compiler can learn from the DSG and vice versa to help programmers understand their programs.

Beyond the DSG, I believe that conventional compiler and programming techniques obscure opportunities for program optimization and understanding in modern languages. For example, unlike in conventional languages, a managed language with a just-in-time compiler dynamically chooses which methods to compile at each optimization level depending upon how frequently they are called. Methods that are called more frequently (hot methods) are recompiled using additional optimizations to improve their overall performance. Some conventional compiler

techniques (e.g. interprocedural pointer analysis) cannot be performed in this scenario since compilation time affects runtime. Current research has used machine learning to learn different optimization strategies based on method features. I plan to examine how the computation of compilation can be exploited to further program understanding. For example, can calculating dominators during compilation to perform code motion provide useful information to the garbage collector to optimize the placement of objects in the heap and can dominator information be provided to the developer to increase program understanding?

In summary, I would like to build a research group whose focus is programming languages and managed runtime systems. In order to be successful, I plan to collaborate with architecture, operating systems, and applications groups.

Publications

- [1] **M Jump** and K. S. McKinley, *Cork: Dynamic Memory Leak Detection*, In Proceedings of the 2007 Principles of Programming Languages, Nice France, January 2007.
- [2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Kahn, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hoskins, **M. Jump**, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Weiderman. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland Oregon, USA, October 2006.
- [3] **M Jump**, S. M. Blackburn, and K. S. McKinley, *Dynamic Object Sampling for Pretenuring*. In Proceedings of the 2006 International Symposium on Memory Management, pp 152-162, Vancouver, British Columbia, Canada, October 2004.