Dynamic Analysis from the Bottom Up

Markus Mock
University of Pittsburgh
Department of Computer Science
6405 Sennott Square, Pittsburgh, PA 15260, USA
mock@cs.pitt.edu

Abstract

Changes in the way software is written and deployed today render static analysis increasingly ineffective. Unfortunately, this makes both traditional program optimization and software tools less useful. On the other hand, this also means that the role and importance of dynamic analysis is continuing to increase. In the future, we believe dynamic analysis will be successful both in program optimization and in software tools. One important ingredient to its success lies in efficient profiling methods. This paper looks at how this goal can be realized by exploiting already existing hardware mechanisms and possibly new ones. We believe that this will lead to software tools that are both effective and minimally invasive, easing their adoption by programmers.

1. Introduction

From its early beginnings, static analysis has been a huge success story. It is routinely used in optimizing compilers to ensure the correctness of code improving transformations. It is also commonly used in programming tools (e.g., smart editors) and software tools designed to facilitate the debugging and evolution of software, for instance, in program slicers. On the one hand, static analysis has been so successful because its use is unintrusive and does not require running the program or any other user intervention, and typically the user is completely unaware of its presence. On the other hand, to achieve practically useful results, typically the whole program, or large parts thereof have to be available to the analysis.

Unfortunately, this traditional model has been eroding over the last years thereby rendering traditional static analysis methods ever less effective. Since software is now routinely deployed as a collection of dynamically linked libraries, and more recently, also as Java bytecode that is delivered dynamically and on demand, static analysis in compilers and other programming tools knows less and less of the finally executing program. This forces it to make conservative assumptions that result in analysis results that are too imprecise to be useful either for program optimization or program "understanding" tasks.

While traditional static analysis is of limited effectiveness in these new dynamic software environments, *dynamic program analysis* [3] will play an increasingly important role to realize tasks that have become inefficient with static analysis alone. Moreover, dynamic analysis will enable new powerful techniques – both in optimization and program understanding – that are impossible to achieve with static analysis alone.

For some time now, dynamic (i.e., run-time) information has been used in optimizing compilers in the form of feedback-directed optimization where run-time information is used to aid the static program optimizer to make better optimization decisions – decisions, that would otherwise have to rely on static heuristics, which generally result in less effective optimization. More recently, run-time information has been exploited in dynamic compilation systems and just-in-time (JIT) compilers to which the complete program is available, which makes their analyses often quite successful [1].

While leveraging dynamic information in such systems has become quite popular, the use of dynamic analysis in software tools designed to assist the software engineer is still in its infancy. While the use of dynamic information in program optimization systems is always confined by the constraint of soundness – a potentially faster but possibly incorrect program has to be avoided –, tools designed to assist a human in a software engineering task are free of this restriction. Moreover, in many cases the results of a static analysis, although sound, may be considerably less useful than the potentially unsound result of a dynamic analysis, for instance, if it overwhelms the user with too much data.

For all the foregoing reasons, we believe that dynamic analysis algorithms, modeled after classical static analyses will be both important and useful in future software de-

velopment environments. Unconstrained by the yoke of soundness, dynamic analysis is likely going to be even more successful in software engineering applications than the promise it has already shown in run-time optimization. Crucial to the wider success of dynamic analysis, however, is the creation of efficient profiling methods to collect dynamic information unintrusively and with little performance overhead.

Therefore, we propose to design dynamic analysis systems "from the bottom up". Currently existing hardware mechanisms can be exploited to make the collection of run-time information more efficient. Software engineers interested in dynamic analysis should also work with hardware designers and compiler writers to participate in the design of new architectures that enable the efficient collection of data that can assist them in building more powerful and versatile dynamic analysis systems.

The rest of this paper is organized as follows: Section 2 discusses two future directions in the application of dynamic analysis. Section 3 looks at how to achieve efficient profiling methods as one essential ingredient in making dynamic analysis successful. Section 4 discusses related work and Section 5 concludes.

2. Future Directions in Dynamic Analysis

As the usefulness of static analyses decreases, dynamic analysis approaches are becoming more attractive. We see several interesting research directions for dynamic analysis in the coming years:

- research on how to effectively exploit run-time information to optimize programs;
- research on the application of dynamic analysis to improve software tools that assist programmers in the understanding, maintenance, and evolution of software; since such tools do not necessarily have to produce sound results this may be the "killer application" for dynamic analysis;
- research on the efficient collection of run-time information; this includes research into combined hardware-software approaches that will lower the cost of collecting run-time information.

In the following two sections, we will briefly discuss the first two items, which represent two broad application areas for dynamic analysis. In Section 3 we will then elaborate on the last point, which is fundamental for the wider success of dynamic analysis.

2.1 Program Optimization with Dynamic Analysis

To achieve good program performance, increasingly run-time information will be necessary to perform effective code-improving transformations. The fundamental constraint for program optimization, though, is soundness, which is at odds with the unsound nature of dynamic analysis. However, we believe that a symbiosis of static and dynamic analysis will not only be effective but in fact crucial for the success of program optimization of future software systems.

Results of static analysis are always conservative approximations of actual run-time behavior; when programs are only partially known, this problem is exacerbated because worst case assumptions have to be made for all unknown code parts. On the other hand, program properties may in fact be true in most, if not all, runs despite the inability of static analyses to demonstrate this. For instance, [9] has shown that the statically computed sets of potential pointer targets in pointer dereferences in C programs are several orders of magnitude larger than the number of actually observed targets. Consequently, optimizing compilers are often not able to allocate variables to registers because of aliases through pointer accesses,[1] even though those accesses at run-time never or almost never overwrite the variable's value.

Fortunately, static and dynamic analysis may be combined in this case to improve what can be done with static analysis alone. One approach consists of generating multiple code versions, one, in which code is optimized aggressively assuming that no aliasing occurs even though the static analysis is not able to ascertain this. The decision when this specialized code should be generated would be based on a dynamic analysis that checks at program execution whether aliasing does occur. A run-time check would then be inserted in the code to select the correct code version and ensure soundness.

Similarly, other program properties that are usually inferred by static program analysis, might be observed at run time. Static analysis would then be used to generate appropriate run-time checks to ensure the soundness of program transformations that depend on the correctness of those properties. Investigating what properties are both useful and efficiently derivable by dynamic analysis, is an interesting research area for the combination of static and dynamic analysis as well as the exploration of synergies arising from that combination.

[1] Alternatively, if they are allocated to registers, after every possibly aliased write through a pointer, the register value has to be reloaded, which may neutralize the benefit of register allocating the variable.

2.2 Improving Software Tools with Dynamic Analysis

Whereas dynamic analysis will generally have to be complemented by static analysis to be applicable in program optimization, software engineering tools may enjoy the benefits of dynamic analysis in many cases even without supporting static analysis. As has been observed by several researchers, in many cases unsound information may be just as useful or even more useful than sound information in software engineering applications.

The key to the usefulness of dynamic analysis again is the (increasing) imprecision of static analyses. While static analyses may provide a sound picture of program properties. this picture may be too complex to be useful in practice. As an example, consider again pointer analysis. A static points-to analysis[2] may compute several hundreds or even thousands of potential pointer targets for a dereference. When a user wants to understand what are in fact the feasible targets, a points-to set of that size will be too large to be examined completely. Moreover, the static analysis does not provide any insights into which of those targets are more likely than others to show up in practice.

On the other hand, a dynamic points-to analysis [9] shows only those points-to targets that have actually occurred at run time. While this set may not be sound, i.e., miss some targets that may in fact be feasible, since dynamic points-to sets are typically very small, they can be much more useful because the enable the programmer to focus on definitely feasible targets, which in addition, may be prioritized by the frequency of occurrence, so that any subsequent task can be focused to examine the more important (more frequent, or more likely) analysis results first. Dynamic pointer information has been used, for instance, to improve program slicing [8].

3 Dynamic Analysis from the Bottom Up: Achieving Efficient Profiling

One of the fundamental challenges for the success of dynamic program analysis lies in the creation of instrumentation and profiling infrastructures that enable the efficient collection of run-time information. Current approaches typically result in significant program slowdowns [9, 5] so that they are confined to offline use. They are also not invisible to the user and typically additional effort is required to integrate them with current software tools. While this may be acceptable when the result is directly consumed by the user of a dynamic analysis tool, when the dynamically derived information is subsequently used to transform a program for instance, faster turnaround times will significantly enhance the usability of tools based on dynamic analysis. Moreover, if dynamic analysis can be performed with minimal overheads during normal program executions, it may become routine and not require any additional effort from software engineers to tap into the generated information.

In our opinion, one particularly promising approach to reduce profiling overhead lies in the collaboration with computer architects. Processor designers dispose of more hardware resources than ever so that it is not unreasonable to expect that additional structures to support efficient dynamic analysis may be placed on chips if they provide a significant enhancement of functionality. Moreover, very simply hardware structures may suffice and can potentially make a big difference in performance. For example, the addition of hardware data watchpoints in modern processors (for example the Intel Pentium), enables a debugger such as `gdb` to monitor all memory accesses to a particular variable (or set of variables) without noticeable performance degradation on the program. When such hardware support is not present, monitoring the contents of a variable becomes often prohibitively expensive – typical software implementations based on trapping after every instruction, result in slowdowns of a factor of 100.

The additional hardware required to support data watchpoints, on the other hand, is minimal. Similarly, for many of the properties we are interested in dynamic program analysis it may be possible to achieve big performance improvements with simple hardware support. Current processors already have many hardware performance counters, which are used in profiling for program performance. Maybe future architectures will have "analysis counters" to assist software engineers in building fast dynamic analysis tools. If we can show that such support is useful for the software community as a whole, we should have a good case for their realization in silicon.

The following sections will look at potential mechanisms to aid in two particular dynamic analysis tasks: points-to profiling and invariant detection.

3.1 Example One: Points-To Profiling

Maintaining a mapping from the current addresses of local and heap-allocated variables to their compile-time names accounts for the major part of the cost of points-to profiling.[3] If the compiler could simply load the monitored addresses into a hardware table and all load and store instructions would automatically be checked against this table (simultaneously updating the associated access statistics), points-to profiling would only add a small amount of extra

[2] A points-to analysis computes for each pointer dereference in a program the set of potential targets accessed by the dereference, called the *points-to set* of the dereference.

[3] The addresses of local variables usually change with each invocation and multiple addresses are usually associated with the same memory allocation site.

work (the initialization of the address table at procedure entry and at each malloc site). In current software implementations, for every load and store instruction tens or hundreds of instructions have to be executed resulting in slowdowns of one to two orders of magnitude [9].

Some current processors, e.g., the Intel Itanium, already support a similar, though more limited hardware structure (the *ALAT* table [7]), which, however, cannot be directly loaded by the compiler (it is manipulated indirectly through special load instructions used for optimization). Therefore, it appears not unreasonable to assume that a more general mechanism similar to the one described above, may eventually be implemented in hardware.

3.2 Example Two: Invariant Detection

Another field where compiler and architecture support can be used to improve the applicability of dynamic analysis is invariant detection. In the Daikon [5] system, invariants are detected offline after a profiling urn of an instrumented program. Obviously, for dynamically updated software this two-phased approach does not directly work since the program needs to be re-instrumented as it is running. Moreover, it may actually be desirable to detect some invariants as the program is running, for instance when invariants represent security-relevant properties.

Arnold and Ryder [2] present an approach to reduce the cost of instrumented code by providing a mechanism to dynamically enable and disable the profiling of selected program parts. Their approach could be combined with Daikon in a run-time system that would automatically instrument dynamically changing code. The dynamically updated code could then be gradually profiled to detect its (local) invariants and as soon as invariants stabilize, profiling would be disabled until the next software update. This would enable invariant detection while a system is running, at potentially very little overhead, so that invariant detection could remain in place even in deployed software.

This would make exciting new applications possible. For instance, software could be shipped with previously detected or specified invariants. As the system runs, these invariants could be compared against those detected in the field. Discrepancies, which might indicate, for example, insufficient testing, could then be reported back to the developer to either correct the software or the invariants.

4. Related Work

PREfix [4] was one of the first tools that tried to overcome the imprecision of static program analysis by a systematic exploration of program execution paths along which certain program properties were checked. It was shown to be very effective in detecting program errors that were not detectable by static analysis. Ernst [5] has focused on detecting likely program invariants, which can then be used to reason about programs or in error detection. The DIDUCE system [6] uses dynamic program analysis to detect unusual program states which are likely to indicate program bugs.

5. Conclusions

Due to changes in the way software is written and deployed today, the effectiveness of static analysis is decreasing. Therefore the importance of dynamic analysis will continue to increase. Consequently, improving the usability of dynamic analysis tools by making them less intrusive and more efficient is one of the main challenges for dynamic analysis researchers today. By designing dynamic analyses from the bottom up, and in collaboration with compiler writers and computer architects, we believe that efficiency and ease of use will be achieved and make dynamic analysis a standard feature of future software systems.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, USA, Oct. 2000.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 168–179. ACM Press, 2001.

[3] T. Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, June 2000.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[6] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM Press, 2002.

[7] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, 2002.

[8] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM International Symposium on the Foundations of Software Engineering*, Charleston, SC, Nov. 2002.

[9] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, Snowbird, UT, USA, June 2001.