Using DDL to understand and modify SimpleScalar

Naoman Abbas Sumant Tambe Jonathan E. Cook
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

1. Introduction

Many legacy systems are built on the technology of dynamic link libraries. Software engineering tools have not exploited this framework very well. Because the linking is delayed until runtime, it seems natural to provide tools access to the linking process, in order to track it, and even to modify it. There is no reason the dynamic linker *must* resolve a symbol to the intended function. To this end we have created DDL (the Dynamic Dynamic Linker) by modifying the Gnu dynamic linker to provide an API for developers to use to customize the linking process. We believe that DDL will be useful for a wide variety of software engineering tasks, not the least of which is the reverse engineering of legacy systems, and their maintenance and modification. In this paper, we present a case study of one such task. This task is the understanding and modification of the SimpleScalar CPU/architectural simulator [1]. We were able to use DDL to understand the behavior of SimpleScalar at a function-call level, and then used DDL to modify the run-time behavior (without modifying the functional source code) to implement the switching between detailed and functional simulation modes (an important extension to the community [2]).

2. DDL: A Dynamic Dynamic Linker

DDL is an extensible dynamic linker that allows powerful control over the linking process, enables the easy construction of runtime monitoring tools, and supports the runtime evolution of dynamically linked programs. DDL is a modification of the Gnu dynamic loader, which is part of the Gnu C library. The fundamental capability that DDL supports is link interception and redirection. When a symbol is being looked up for purposes of linking, our hooks in the dynamic linker perform callbacks into the DDL control library. In this, the hooks built into the dynamic linker do not provide an API to external services but rather they use an API provided by DDL control library. DDL control and the tools that use them are generally passive and event-driven, those events being, for the most part, link requests. DDL also maintains an internal data structure of resolved symbols (functions), and the bindings or links that refer to them. Maintaining this information during the runtime of the program allows DDL to support dynamic program evolution through runtime link modification (e.g., rebinding). Related work can be found in [3].

This work was supported in part by the National Science Foundation under grants CCR-0306457, EIA-9810732, and EIA-0220590. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

3. Understanding SimpleScalar

Since we are focusing on being able to combine the detailed and functional simulations, we concentrate on `sim-outorder` and `sim-safe` rather than on other parts of the SimpleScalar suite. We should note that to use DDL we had to modify SimpleScalar slightly, by changing the build to use shared libraries and by removing some "static" declarations to expose symbols to DDL. We used DDL to generate the dynamic call graphs of both `sim-safe` and `sim-outorder`. DDL allows us to do this easily by using table-based redirection. In this mode, DDL can allow the tool builder to use a single tracing wrapper around all traced function calls. Since these two simulators are related, we then wanted to see the differences in their call graphs, in order to understand them a little better. Figure 1 is a view of the parts of `sim-outorder` that are not in `sim-safe`. We see that in `sim-outorder`, `ruu_dispatch()` calls `mem_access()` exactly the same number of committed loads and

Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)
1095-1350/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

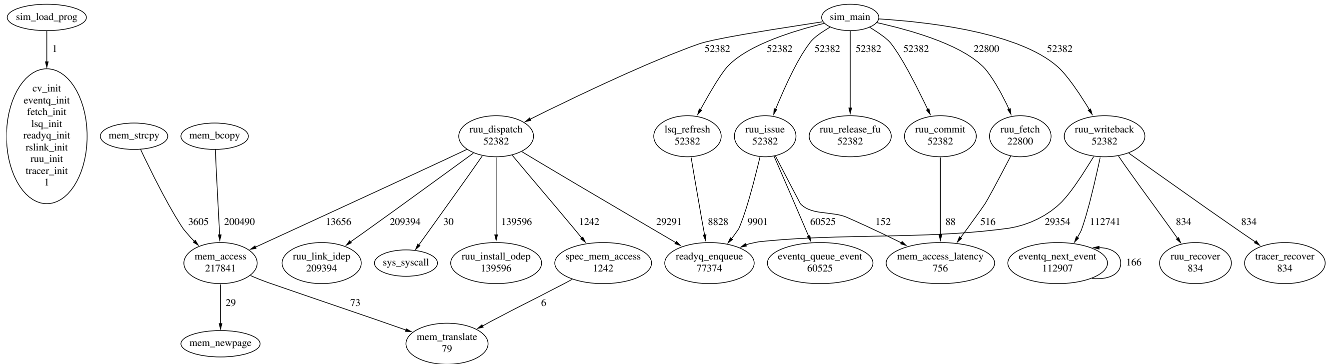

Figure 1. Parts of `sim-outorder` not in `sim-safe`.

stores that the simulator reports , and the number of memory pages used is that same as the number of calls to `mem_newpage()`. The `sim-outorder` simulation calls its main sub-functions each 52,382 times, which is one more than the number of cycles simulated because of the way the simulator finishes the program.

4. Modifying SimpleScalar

As can clearly be seen, the `ruu_*` and `lsq_*` calls, which are called once for each simulated cycle, offer a clean slicing point at which we could expect to modify the behavior with DDL. With DDL we can dynamically modify which functions are called at these call sites. We simply take the calls that enter into the detailed simulation, redirect all but one of them to an empty-bodied function, and then redirect one (we chose `ruu_fetch()`) to a function that performs one functional instruction simulation step. In this manner, the source code for `sim_main()` is not changed but the behavior is switched back and forth between detailed and functional simulation.

Our initial attempts at naively switching were not successful. Essentially, before switching from detailed to functional simulation, we needed to finish simulating all the instructions that were already in the CPU's pipeline. Switching now needed three modes rather than just two: detailed simulation, flushing the pipeline, and functional simulation. Since functional simulation did not do any hardware simulation and thus always finished the instructions it "fetched", there was no patch-up mode needed in going from it back to detailed simulation. We thus accomplished a major change in the functionality of a fairly complex program without changing its functional code. Without any explicit calls to our new code, we were able to use the

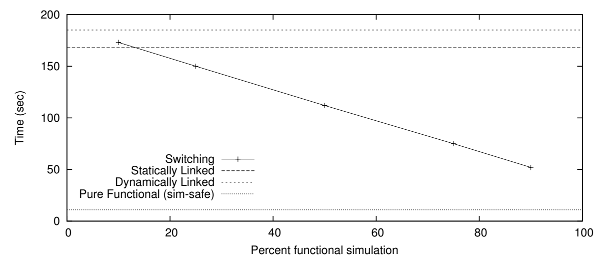

Figure 2. Functional percent variation plot.

DDL framework to modify the system to exhibit new, valuable behavior.

5. Evaluation and Conclusion

Figure 2 shows the performance of the base simulators and our modified simulator over a varying amount of functional simulation with the switching frequency held constant. DDL was successfully used to understand and then modify the behavior of `sim-outorder` without excessive overhead.

References

[1] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. *Technical Report 1342, Computer Sciences Department, Universi ty of Wisconsin*, June 1997.

[2] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. *4th IEEE International Workshop on Workload Characterization*, December 2001.

[3] A. Serra, N. Navarro, and T. Cortes. DITools: Application-level Support for Dynamic Extension and Flexible Composition. In *Proc. 2000 Usenix Technical Conference*, pages 225–238, June 2000.

1095-1350/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY