Using DDL to understand and modify SimpleScalar

Naoman Abbas[†] Jonathan E. Cook[†] Sumant Tambe[†] Jeanine Cook[‡] Ram Prasad[‡]
Department of Computer Science[†]
Department of Electrical and Computer Engineering[‡]
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Abstract

As a case study application of DDL*, our tool for dynamically inspecting and modifying the linking of programs using dynamic link libraries, we investigate the SimpleScalar CPU/architecture simulator. In particular, we gain an understanding of SimpleScalar's behavior in order to modify it to be able to dynamically switch between detailed-but-slow full architectural simulation and functional-but-fast instructions simulation. This application anecdotally shows that* DDL *is useful for reverse engineering and re-engingeering legacy systems.*

1. Introduction

Many legacy systems are built on the technology of dynamic link libraries. In this deployment style, code libraries are linked at run-time by a dynamic linker that runs alongside the actual application. It intercepts initial calls to unlinked functions/methods, finds the correct library and loads it if necessary, resolves the symbol into an address, updates the call link to point directly to the address, and finally jumps to the intended function. All subsequent calls go directly to the target, and thus the overhead of dynamic linking is only paid once.[1]

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. A tool might prefer to have it resolve to a wrapper, which may provide extra result checking, protection from a known bug, or may translate certain argument values. An application developer might prefer to redirect a legacy call to new code that is improved in some way.

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. Some of the authors are already experts in using SimpleScalar, and are investigating methods for improving the speed and efficiency of architectural simulation. One such method is to skip the "uninteresting" parts of a workload that is being simulated by performing simple, fast, functional simulation on those parts, and only performing full architectural simulation on parts that are "interesting" or new.[2] SimpleScalar already has a very limited form of this—it allows one to "fast forward" through an initial part of the workload, but once it starts the full architectural simulation, it does not have the ability to stop.

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 source code[3]) to implement the switch-

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.

1 There are other models of dynamic linking, but this is the most widely used mechanism.

2 In this paper we are not attemping to define what "interesting" means. We are only trying to build the mechanisms that would allow someone to switch simulation modes.

3 We did modify the source code for other purposes, as will be explained later, but we did not modify it to explicitly invoke

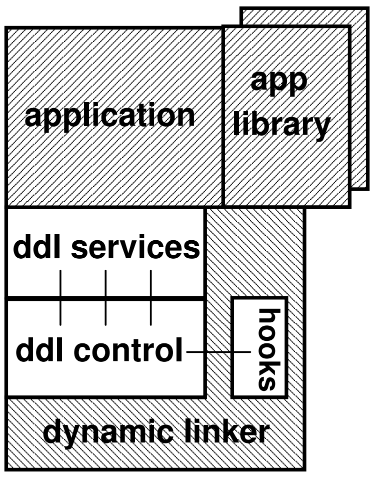

Figure 1. DDL **system architecture.**

ing between detailed and functional simulation modes.

Section 2 presents an overview of our DDL tool, Section 3 presents our reverse engineering of the behavior of SimpleScalar, and Section 4 presents how we used DDL to modify its behavior. Section 5 quantitatively evaluates the new SimpleScalar behavior, and the Sections 6 and 7 wrap up the paper with discussion on related work and conclusions.

2. DDL: API Access to the Dynamic Linker

Here we present a brief overview of DDL, a framework of modifications and extensions to the dynamic linker that allow us to have dynamic control over the linking process, and to implement the full range of desired features listed above. DDL 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. Our current tests have only been on the Gnu/Linux platform, although the Gnu libraries (and the dynamic linker) are ported to many other platforms.

Figure 1 shows the high-level system architecture that DDL implements. The shaded portions indicate parts of the system that DDL does not modify. The application and application libraries are not modified, at the source or binary level, and the bulk of the system dynamic linker is unmodified.

our new functionality.

At the lowest level, we added hooks into the dynamic linker itself so that we could interact with the linking process. On top of these hooks we built useful service abstractions so that tool builders would not need to start from scratch. Further, we implement a application services level that provides even higher level interaction for some types of common services—one such service is scripting language support.

The fundamental capability that DDL supports is link interception and redirection. This allows DDL and the tools that use it to peer into the dynamic linking process, and control it. 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.

With the link interception, we can maintain 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 us to support dynamic program evolution through runtime link modification.

In order to modify a link, we simply need to change the address that is in its jump table entry to be the address of some other function. All the subsequent calls through that link will be directed to the new function. Note that jump tables are allocated per shared object (the main program and other shared libraries), and so these calls are from all the call sites in the shared object whose link we just modified. Thus, the granularity of program evolution is at the shared object level.

Once the link is initially resolved, however, the linker is never going to be called for that particular link again. Therefore, we have to have some way of regaining control over the execution of the program in order to perform runtime link modification. We currently employ the OS's signal mechanism to accomplish this. Our installed signal handler reads a remapping specification, and rebinds the links as directed. Once the application resumes execution, these new bindings will have immediate effect (when they are used).

Although we have much future work to make this type of program evolution generally useful, such as concerns about state corruption or migration, interference between existing calls to the old bindings and new calls to the new bindings, recursion, and the like, there are some applications where the current functionality can be used already.

# of instructions executed	49475
# of loads and stores	13655
code size in bytes	188416
data and bss in bytes	41984
initial stack size (bytes)	16384
# pages allocated	29
size of memory allocated	232k
1st level page table misses	75
# page table accesses	536256

Table 1. `sim-safe` results from simulating `test-math`

# of instructions committed	49475
# of instructions executed	54901
# of loads committed	8508
# of stores committed	5147
# of loads & stores committed	13655
# of loads executed	9406
# of stores executed	5492
# of loads & stores executed	14898
# of branches executed	8259
# of cpu cycles	52381
1st level page table misses	79
# page table accesses	716030

Table 2. SimpleScalar results from simulating `test-math`

3. Understanding SimpleScalar

SimpleScalar is actually a family of simulators. While most people think of its full detailed simulator, `sim-outorder`[4], when they hear "SimpleScalar", its distribution also includes a basic functional simulator, `sim-safe`, a cache simulator, and other simulator configurations. Since we are focussing on being able to combine the detailed and functional simulations, we concentrate on `sim-outorder` and `sim-safe`.

The workload we used for the understanding phase was a very small one, the `test-math.c` that is included in the distribution. It simply calls various math library functions and checks the results, in a linear sequence and directly in `main()`. The top-level data that `sim-safe` outputs for this workload is shown in Table 1, and the (condensed) output from `sim-outorder` is shown in Table 2. This data will help us understand our later data.

4 Then name comes from the fact that it simulates modern processor designs that can do out-of-order instruction execution.

We should note that to use DDL we had to modify SimpleScalar slightly. This is because it is not generally built using shared libraries. When one downloads, installs, and builds SimpleScalar, it builds statically linked single executables for each simulator, even though there is code sharing between them. This is because when doing large simulations that can take days or weeks, every bit of performance is needed, and statically linked code is slightly faster (by one jump instruction for every call) than dynamically linked code.

Our first modification was to change the `Makefile` to build the simulator using shared libraries. This was a trivial change which needed no modification of source code. Next, we noted that each simulator, apart from code shared with other simulators, was implemented in a single file and used `static` declarations on many functions (`sim-outorder.c` is 4605 lines long). To gain access to the calls within this file, we removed the `static` designation and created a dummy shared library with code that "called" these functions, although the calls are never executed. This trick makes the linker route even local calls through the dynamic link jump tables, so that DDL can process calls local to a shared object.

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, rather than creating a unique wrapper for every function. The single wrapper relies on the C calling conventions to call each traced function without knowing its type signature. The only restrictions are that we know the maximum number of argument bytes that any traced function will use, and that all traced functions return an integer-sized return value. SimpleScalar obeyed these restrictions.

Figure 2 is a view of the important parts of the dynamic call graph of `sim-safe`. We clipped some of the initialization calls for the sake of readability. The numbers in the nodes and on the edges represent the number of calls of a function, in total and per parent, respectively. `sim-safe` is a fairly simple program, and indeed its main simulation engine is within a single function, `sim-main`, only making a few calls for memory referencing and for system calls from the simulated workload.

Figure 3 is a view of the dynamic call graph of `sim-outorder` (again, with some initialization calls removed). As can be seen, `sim-outorder` is a much more complex application. It also uses a function `sim-main` as the top-level simulation function, but it clearly farms

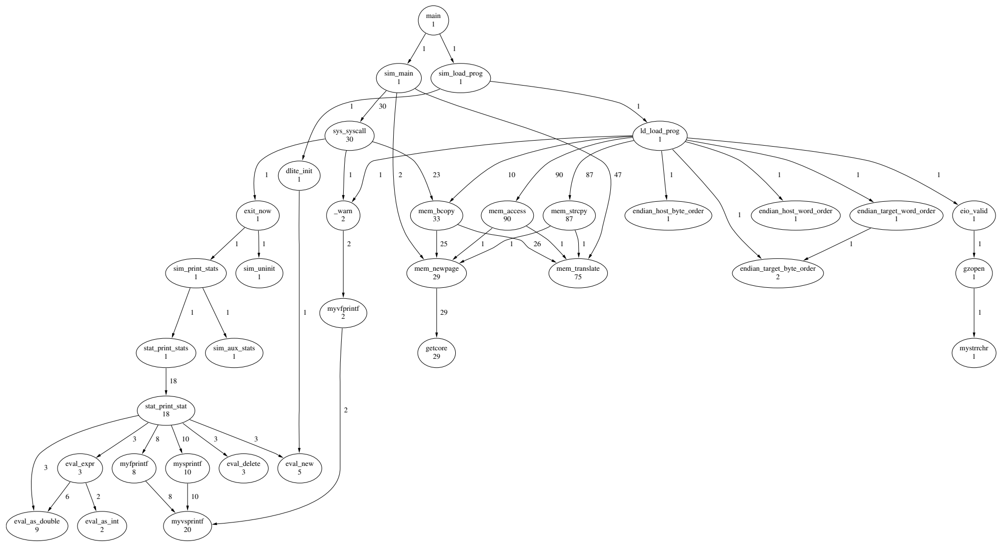

Figure 2. Sim-safe: fast functional simulation.

out much of the detailed simulation to called functions.

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. These were easily generated (actually using `diff` on the *dot* files), and appear below.

Figure 4 is a view of the parts of `sim-outorder` that are not in `sim-safe`, and Figure 3 is a view of the parts of `sim-safe` that are not in `sim-outorder`. Since `sim-outorder` is the more complex simulator, we would expect that it contains much behavior not found in `sim-safe`, and indeed the graphs show this. The content of Figure 3 is actually there only because the number of calls differ between the two simulations—the functions and their relations exist in both simulators.

These graphs show some interesting behavior and point to some possible ideas for modifying that behavior. We should note that initially we started comparing the call graphs to output taken from a run of `sim-outorder` compiled statically (from an unmodified distribution). When we did this a few anomalies, such as the number of pages allocated being 1 off of the number of calls to `mem_newpage`, the number of cycles simulated being several tens off of the number of calls we thought would match it, and the number of memory access being slightly different than the number of calls to `mem_access`. We discovered that simply compiling `sim-outorder` differently actually affected the results!

Investigating this further, we found that this was known in the SimpleScalar community, and we believe it comes from the fact that it use memory addresses transparently from the simulator space into the workload space, and this affects caching and subsequent instruction execution. While the size of the variation is probably miniscule in proportion to the magnitude of the results that are of interest to architecture researchers, this variablity was disconcerting to us, and seems incongruous with the desired qualites of computer simulations.

Nevertheless, in looking at the output produced by the run from which we produced the dynamic call graph, the numbers matched very well. The `sim-outorder` simulation calls its main subfunctions each 52,382 times, which is one more than the number of cycles simulated (more than the number of instructions committed but less than the number of instructions executed, since speculative execution is supported). We also see that in `sim-outorder`, `ruu_dispatch()` calls `mem_access()` 13,656 times, also one more than what the simulator reports. **Check this – This slight difference would be worth looking in to in the source code, or may be an artifact of our data collection.**

As far as potentially using DDL on `sim-safe`, as we mentioned earlier, it does not call functions to perform instruction simulation. From our perspective, this means we would not have much access to its behavior, since DDL operates at the function-call interface.

For `sim-outorder`, however, it has extremely convenient behavior from our point of view. As can clearly be seen, the `ruu_*` and `lsq_*` calls, which are called

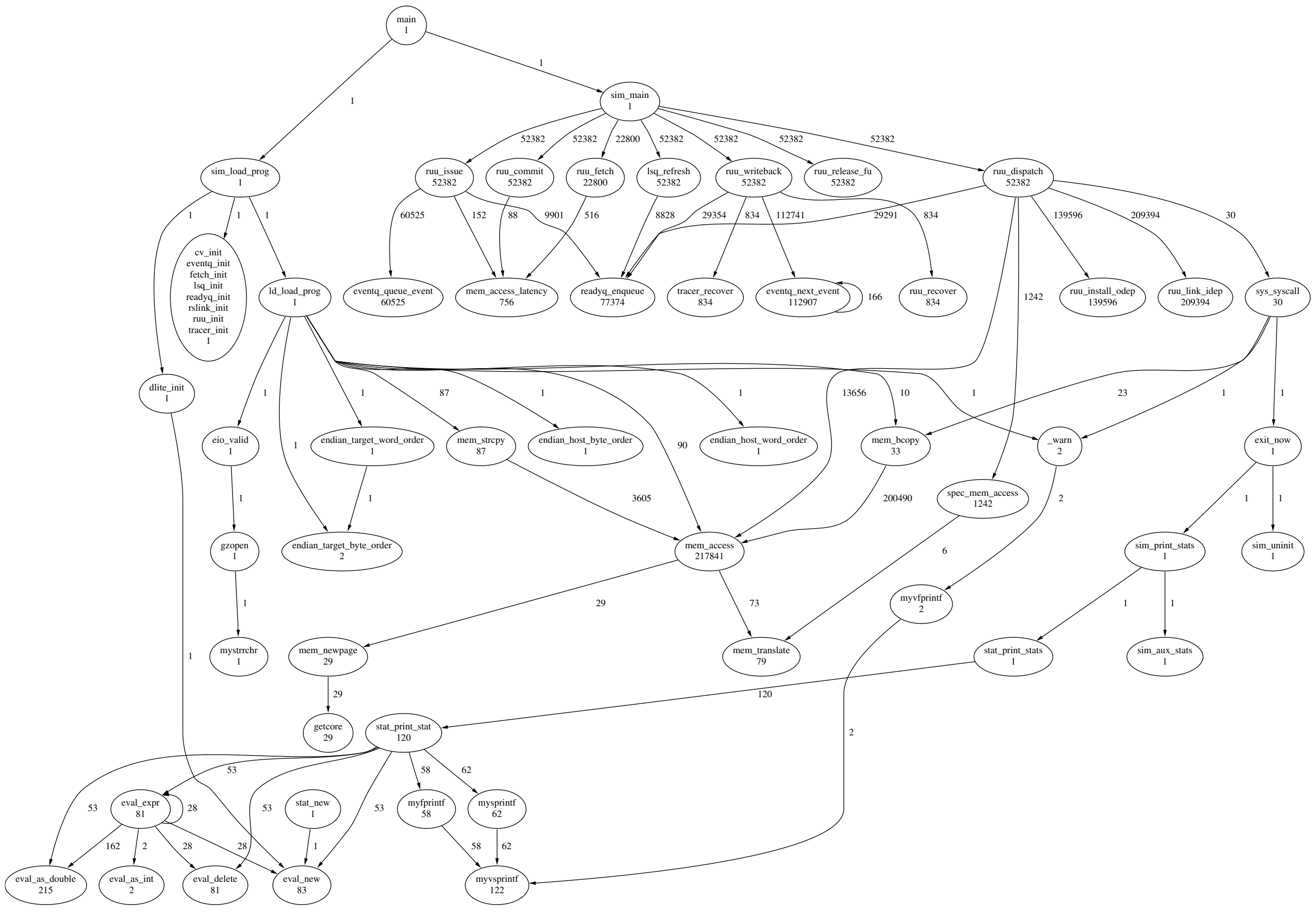

Figure 3. Sim-outorder: full architectural simulation.

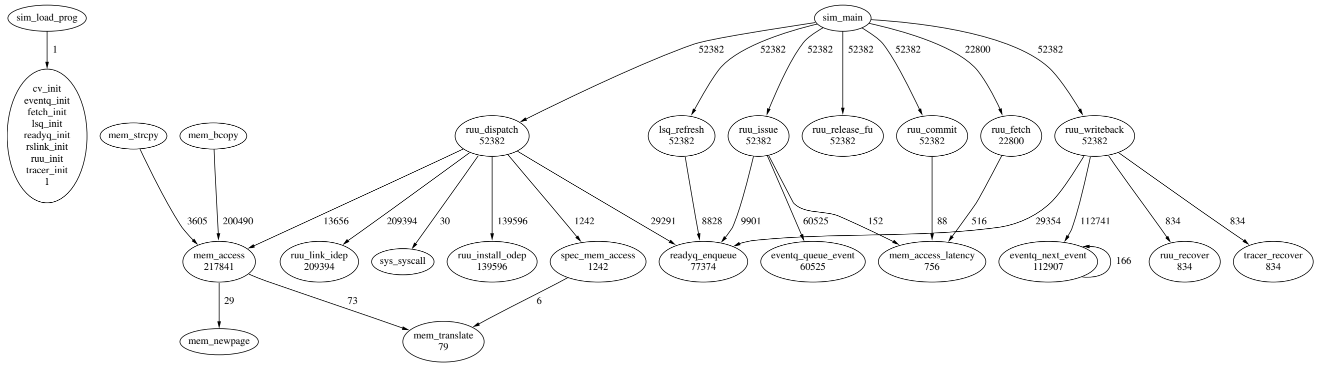

Figure 4. Parts of `sim-outorder` not in `sim-safe`.

once for each simulated cycle, offer a clean slicing point at which we could expect to modify the behavior with DDL. Since we had the source code in this small study, looking at the source code shows the main simulation loop in `sim_main` to be almost solely the repetitive calls of these functions (along with some counter updates).

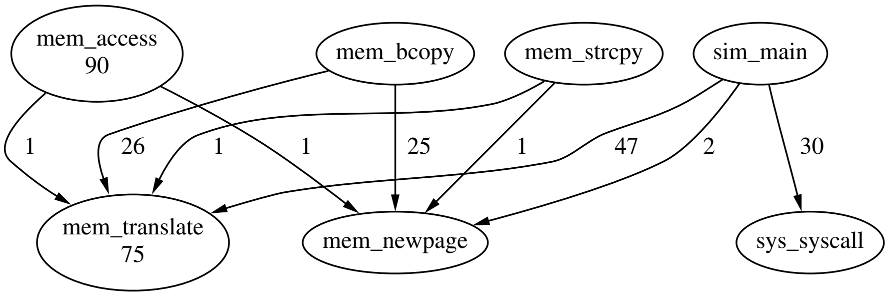

Figure 5. Parts of `sim-safe` not in `sim-outorder`.

If one is familiar with modern CPU design and the accepted names of pipeline stages, it can quickly be seen that the functions are simulating the pipeline stages. Furthermore, the stages are called in reverse order—e.g., the main simulation loop first calls `ruu_commit`, the last stage in a real pipeline, and finally calls `ruu_fetch` at the end of the loop, which is the first stage. This makes sense because while in a real simulator the stages operate concurrently, in lock-step mode with the system clock, the simulator simulates them sequentially, and so simulating the last stage first makes available the buffer between it and the previous stage, and this wripples all the way back up the pipeline.

4. Modifying SimpleScalar

Through understanding `sim-outorder` we saw a very promising point at which to use DDL to modify its behavior, namely that of the set of functions that are called once per cycle. Our basic idea is shown in Figure 6. 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.

Note that this study and this paper are not about *deciding* when to switch modes—-that is a complicated decision and is still being researched in the simulation community [**?**]. We are just interested in seeing *if* DDL could be used to support a change known to be of interest to users of the simulator. As such, we had to rig a dummy mechanism to switch back and forth between the detailed and functional simulation modes. We did this simply based on counters such that we could control the percentage of time spent in either mode and the frequency of switching modes, both of which are important in evaluating the performance of our approach.

Our initial attemps at naively switching were not successful. Indeed we suspected this might be the case, but we wanted to approach the problem one piece at a time. When we saw the simulator failing, we started investigating the second part of the problem, and that is how to manage consistency between the two simulation modes.

We rightly suspected that we could not arbitrarily switch out of detailed simulation mode, because the simulator would leave partially executed instructions in the simulated pipeline. Not only does this mean that some small number of instructions were not finished that should be finished before the functionally simulated instructions were executed, it also meant that those instructions *would* finish once we switched back to detailed simulation. Thus, switching modes corrupted the state of the simulated workload because some instructions were left incomplete. We suspected this would happen, and indeed it did.

To overcome this situation, we needed to learn more about the detail of the `sim-outorder` implementation. Rather than start from scratch on this problem, we sought some expert help from researchers using the simulator for their own research and who had experience in modifying it (two of the co-authors of this paper). Essentially, before switching to functional simulation, we needed to simulate all of the pipeline *except* the fetch mode (which fetches new instructions) until the pipeline was empty.

Fortunately, the fetch mode was embodied in a function call. 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.

Figure 7 shows our second approach, embodying the three modes that we now knew we needed. Intermediate to switching to the functional mode, we use DDL to point the `ruu_fetch` call to a function that does not fetch any new instructions but does check the sta-

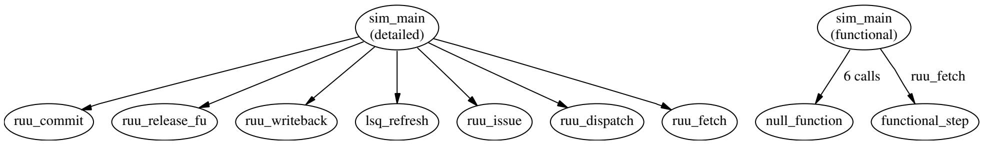

Figure 6. Initial DDL-based `sim-outorder` dynamic modification approach.

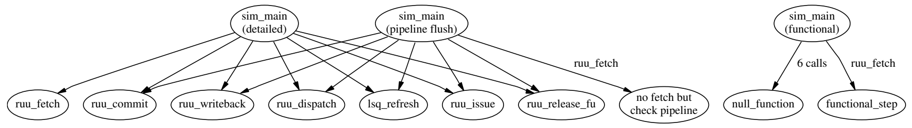

Figure 7. Correct DDL-based `sim-outorder` dynamic modification approach.

tus of the pipeline simulation. Our `sim-outorder` experts had found a simple condition over global variables that indicated when the pipeline was completely empty, and when this condition was reached, we then completed the switch to functional mode by using DDL to now switch all of the necessary function calls over to the functional mode.

Modulo a few minor programming mistakes, this technique worked almost instantly. 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 DDL framework to modify the system to exhibit new, valuable behavior.

Simulation	Seconds
statically linked	168
dynamically linked	185
10% functional	173
25% functional	150
50% functional	112
75% functional	75
90% functional	52
sim-safe (100% func)	11

Table 3. Performance over varying amounts of functional simulation. (switching frequency is nominally every 100,000 instructions

5. Evaluation

Of course, the important aspect of this case study was not only to see if DDL could be used on a practical problem, but also to actually speed up the simulation of a workload by `sim-outorder`. This means that we needed to evaluate the performance of our modified simulator.

While it might seem obvious that the performance will improve, we do add some overhead. Firstly, the original `sim-outorder` is linked statically, while our changed version is linked dynamically. Secondly, the switching adds overhead in both the basic DDL code that does the switching, and perhaps more importantly, the overhead to flush the pipeline each time switching is done in the detailed-to-functional direction.

We considered that the two main dimensions to evaluate performance on were the amount of the workload that was simulated in each mode, and the frequency at which the simulator switched modes. The first captures a relative measure of the time it takes to simulate an instruction in detailed mode versus the time it takes in functional mode, and the second provides a measure of the overhead costs of doing the switching. We also provide measurements of SimpleScalar's own pure functional simulator `sim-safe`, the performance of the original statically linked detailed simulator `sim-outorder`, and a baseline dynamically linked `sim-outorder` that is not running under our framework. Both of the `sim-outorder`'s are of course doing 100% detailed simulation.

We used an integer SPEC benchmark workload, MCF, with a very small input that we created by hand. Since we are not particularly interested at this point in *what* the simulator is simulating, rather just in exercising it reasonably, we created a small input, and ended up with a simulation workload of 84,448,851 instructions, which ran in about 11 seconds for a functional simulation and in 168 seconds on the original SimpleScalar detailed simulation.

Table 3 and Figure 8 shows the performance of the base simulators and our modified simulator over a vary-

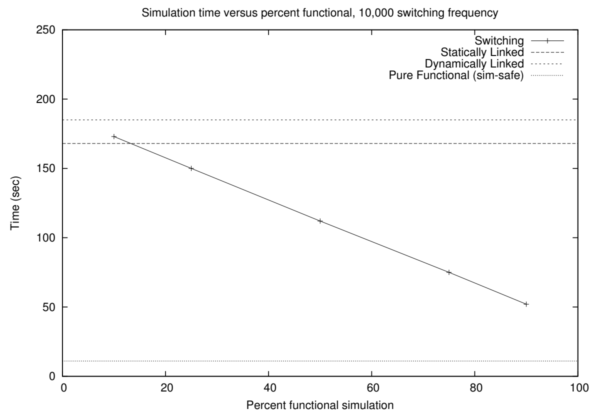

Figure 8. Functional percent variation plot.

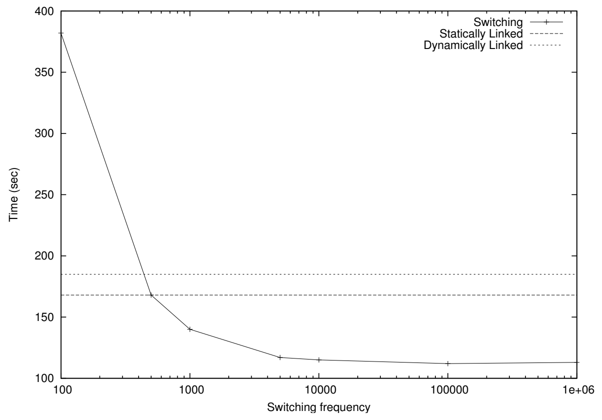

Figure 9. Switching frequency variation plot.

Simulation	Seconds
statically linked	168
dynamically linked	185
100 instructions	382
1,000 instructions	140
10,000 instructions	115
100,000 instructions	112
1,000,000 instructions	112

Table 4. Performance over varying switching frequencies, with a constant 50% functional simulation.

ing amount of functional simulation, with the switching frequency held constant at 100,000 instructions. As can be seen, we achieve a very nice linear performance improvement as the amount of detailed simulation goes down.

Table 4 and Figure 9 shows the performance of the base simulators and our modified simulator over a varying rate of switching, and the percentage of functional simulation held constant at 50%. Note that on Figure 9 a log scale is used on the horizontal axis. Since there is overhead in switching, not only just the mechanics of switching using DDL but also the flushing of instructions already in the pipeline when switching from detailed to functional, we would expect that a point can be reached where too much switching actually slows down the simulation, and this is clearly the case. At the far end of the frequency, we reach an asymptote where the switching overhead is no longer a measureable part of the performance, and the ratio of detailed to functional simulation is the only dominant factor in performance.

6. Related Work

The DITools project [8] is the closest related work to our DDL project. They used a similar approach to link interception and modification, and supported redirecting a link to a wrapper and also an event notification mechanism where each monitored call was not wrapped but did generate an event to a fixed-interface callback. It does not appear that they addressed the issues surrounding C++, nor did they do non-function symbol resolution nor runtime link modification.

Ho and Olsson [5] describe *dld*, a tool for "genuine" dynamic linking. Their tool provides the capability to load and unload shared libraries, breaking links when a library is unloaded and relinking them to new code when new libraries are loaded. However, it does not appear that they ever supported redirection of links to different symbol names.

Hicks et. al [4] work on binary software updating from a formal perspective. Their methods use typed, proof-carrying assembly code from which they can verify that an update will be safe. Their infrastructure includes special languages and compilers to generate the annotated assembly code, and a runtime framework that uses it.

Additional systems that provide instrumentation capabilities on executable binaries exist. Dyninst [1] can patch custom code into pre-existing executable code, and has provided a platform for several research tools. Valgrind [9] provides a complete simulated CPU and execution space to the program under inspection, and is extensible, thus allowing new dynamic analyses to use it as a foundation.

There is much work in dynamic introspection and modification of Java programs, but since this work is in a very different environment than ours, we do not explain it in detail here. Some representative references are [2, 3, 6, 7].

7. Conclusion

This paper explored the use of DDL to understand and then evolve a medium-sized application. DDL is an extensible platform on which tool builders can inspect and control the dynamic linking process, for uses from simple runtime monitoring all the way to runtime software evolution. In this paper, the study evolved the SimpleScalar architectural simulator without changing any existing functional code. This was done by using DDL to dynamically relink existing function calls.

In the future, we plan to build upon DDL a full reliable runtime evolution environment that will give engineers full support for a process of software deployment that is tolerant of some common errors.

References

[1] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000. www.dyninst.org.

[2] M. Dahm. Byte Code Engineering Library. 2002. http://jakarta.apache.org/bcel/.

[3] S. Eisenbach and C. Sadler. Changing Java Programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 479–487, Nov. 2001.

[4] M. Hicks, J. Moore, and S. Nettles. Dynamic Software Updating. In *Proc. 2001 ACM Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[5] W. Ho and R. Olsson. An Approach to Genuine Dynamic Linking. *Software Practice and Experience*, 21(4):375–390, 1991.

[6] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proc. European Conference on Object-Oriented Programming*, pages 337–361, 2000.

[7] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. In *Proc. 2002 International Conference on Software Maintenance*, pages 649–658, Oct. 2002.

[8] 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.

[9] J. Seward. Valgrind, an Open-Source Memory Debugger for x86-Gnu/Linux. Technical report. valgrind.kde.org.