

# Improving Troubleshooting via Dynamic Dependency Analysis

Marc Chiarini, Josh Danziger, Dr. Alva Couch  
Tufts University

April 24, 2006

## Introduction

The task of identifying the “root cause” of a problem becomes increasingly difficult as systems become more complex, yet existing tools for root cause analysis remain fairly primitive. We examine the potential for aiding root cause analysis via auditing of dynamic dependencies at the kernel level. There are two main challenges to this kind of analysis: gathering data without unacceptably degrading system performance, and representing that data at an appropriate level of abstraction.

We base our work on several prior efforts at dependency analysis that apply differing levels of abstraction. *Sowhat* [3] performs a static, global analysis of dynamic library dependencies in order to predict the effects of a system change. *Strider* [4] performs a similar form of analysis on the Windows registry, using “before and after” snapshots of registry contents to infer potential causes of problems. Our approach looks instead to information that is naturally generated by auditing as a vehicle for dependency analysis. Unlike these prior attempts, our work concentrates on *dynamic dependencies* that cannot be inferred from static analysis but must instead be observed at run-time.

While library dependencies can be identified effectively using *sowhat*’s static analysis, libraries themselves are infrequently modified entities. Program behavior is determined in large part by configuration files and environment; these dependencies change more frequently, yet few tools automatically identify them. *Strider* uses some dynamic analysis, but it only does so on individual programs, essentially missing transitive and asynchronous dependencies. Furthermore, *Strider*’s approach assumes that undesired program behavior is reproducible, which is not always the case.

## Dependency Analysis

Auditing at the kernel level is an effective strategy to overcome the inherent shortcomings of the tools we

have mentioned. By monitoring all program I/O over an adequate time period, it is apparent that a complete dependency graph can be built. As a first step, our research explores dependencies found in the Linux file-system. We make the following observations:

- If a program opens a file for reading, then that file is a potential dependency of the program.
- If a program opens a file for writing, then that file is dependent on the program’s behavior.
- If program *P* writes to file *F*, and program *Q* reads from *F*, then *Q* potentially depends on the behavior of *P*. This relationship may be extended to an arbitrary degree.

A list of potential dependencies may not be particularly useful. For example, on one test run, *zcat* opened 5,617 files across 11,207 invocations, yet the vast majority of these are not dependencies. In fact, *zcat* is statically linked and therefore only depends on the loader and *libc*. Clearly, a metric is necessary to determine the relative strength of a particular dependency. By itself, this metric should significantly reduce the number of files that must be examined during troubleshooting. Complementary strategies will further reduce the search space. For example, an administrator might only examine files that have changed in the last 24 hours.

## Methodology

As proof of concept, we have begun to collect data on the dependency of programs on files. The Linux 2.6 kernel introduced a lightweight auditing subsystem capable of reporting kernel events to user space in soft real-time. The kernel logs events into a ring buffer that are then culled by *auditd*, a user space daemon similar in structure to *syslogd*.

One of the features provided by the audit subsystem is the ability to conditionally log system calls. Auditing the *open()* call generates a list of all file accesses.

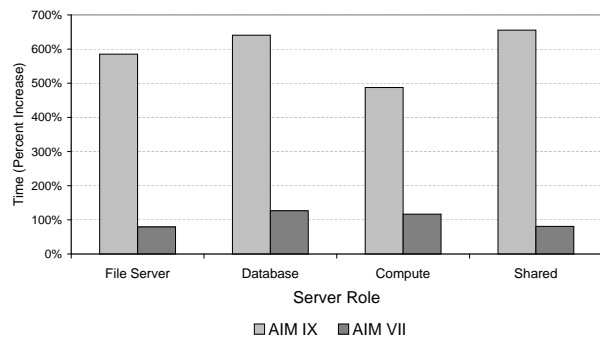


Figure 1: Auditing performance measured by AIM.

Specifically, each log entry provides the following information:

- Filename, inode, and attributes
- Calling program and PID
- Mode (read, write, append, etc.)
- Time
- Success

Using these data, generating dependency graphs is simple. Each entry in the log represents a graph edge between a program and a file. The usefulness of the graph is proportional to the length of time and frequency with which audits are performed. Many graph-theoretic algorithms can also be used to expose interesting substructures.

## Technical Challenges

As with any dynamic auditing mechanism, the Linux audit subsystem is intrusive; it stops execution of the running program in order to log events of interest. Auditing is therefore a double-edged sword; on one side, using native facilities avoids the need to modify an existing kernel; on the other side, the performance impact may be unacceptable in certain environments.

In tests performed with the AIM VII (multi-user) benchmark [1], the time spent in the `open` call roughly doubled with the introduction of auditing. However, the AIM IX (single-user) benchmark showed a dramatic increase in the time spent per call. Further research must be conducted to ascertain the reason for this disparity. Nonetheless, the results of AIM VII are encouraging (Figure 1). We plan to make auditing run even faster by reworking the kernel code and filtering unnecessary data at the source.

Another significant challenge for our research is representing collected data at an appropriate level of abstraction. Because our approach constructs graphs, it is amenable to many useful kinds of visualization. We have already begun to build a graphical query facility using the Prefuse Visualization Toolkit [2]. This immediately allows a system administrator to see important patterns and clusters of strongly connected components. Statistical analysis and graph-theoretic algorithms can also be applied to guide the SA toward root-cause dependencies.

Managing the tremendous volume of data generated by auditing presents a third challenge. In the span of an hour, the AIM benchmarks generated 400MB of raw ASCII data. One way of clearing this hurdle involves simple data compression: by eliminating redundant entries from the audit log, we have successfully reduced 250MB of raw data to a 2MB SQLite database.

## Future Work

Currently, our tools can only perform dependency analysis on a single, non-networked host. If our analysis of file dependencies continues to bear fruit, we plan to implement a similar analysis of networked host populations at the earliest opportunity.

## Conclusions

Root cause identification is significantly enhanced by the comprehensive analysis of dynamic dependencies. We have built new tools to identify and correlate these types of dependencies as found in the Linux filesystem. We will continue to actively tackle the challenges of collecting, managing, and presenting necessary data, while keeping an ever-vigilant eye on performance.

## References

- [1] Aim benchmark. <http://aim.sourceforge.net>.
- [2] Prefuse: Interactive information visualization. <http://prefuse.sourceforge.net>.
- [3] Yizhan Sun and Dr. Alva L. Couch. Global impact analysis of dynamic library dependencies. Proc. LISA, 2001.
- [4] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. Proc. LISA, 2003.