DEPMAP: DEPENDENCY MAPPING OF APPLICATIONS
USING OPERATING SYSTEM EVENTS

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Ryan Matteson
December 2010
<table>
<thead>
<tr>
<th>COMMITTEE MEMBERSHIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE:</td>
</tr>
<tr>
<td>AUTHOR:</td>
</tr>
<tr>
<td>DATE SUBMITTED:</td>
</tr>
<tr>
<td>COMMITTEE CHAIR:</td>
</tr>
<tr>
<td>COMMITTEE MEMBER:</td>
</tr>
<tr>
<td>COMMITTEE MEMBER:</td>
</tr>
</tbody>
</table>
Abstract

DepMap: Dependency Mapping of Applications Using Operating System Metrics

Ryan Matteson

Applications are increasingly hosted on shared, heterogeneous distributed systems. Web applications and “cloud” computing are broad examples of this, but even relatively isolated servers often depend on shared, network-accessible resources. This standardization around service layers abstracts aspects of service delivery, but often entails a large collection of component systems. The complexity that is abstracted from a functional perspective can give rise to runtime issues. Critical issues arising out of this model include resource utilization (sizing, capacity testing, resource monitoring) and dependency management (identifying and monitoring dependencies between components and potential for impact to an application).

This thesis presents an automated solution called DepMap for identifying and monitoring file, network and other communication dependencies in applications by analysis of low-level operating system activity. This work is applicable to resource utilization, diagnosis of performance issues, characterization of workloads, and systems management. This thesis discusses DepMap’s requirements, design and implementation, and evaluates the effectiveness and performance on simulated and actual applications. DepMap can be used to create models of system behavior, based both on observations over time and on injection of delays into selected system operations to understand impact to the larger system.

When applied to two commercial systems DepMap was able to identify changing dependencies, and to characterize the behavior of network and storage dependencies. On a cluster of systems hosting a web application, DepMap showed unexpected variation in network transmission time between peers in the cluster, and helped to uncover large jumps in system clock times arising from unreliable Network Time Protocol (NTP) services. On a database server that processes hundreds of gigabytes of data each day, DepMap was able to characterize I/O workload to storage connected by Fiber Channel and iSCSI. This work showed that serious performance limitations existed in the storage server due to
fragmentation, design assumptions poorly suited to a data warehouse workload, and competition from other storage consumers. This information was used to design and validate a new server/storage platform specifically for this workload. The insight gained by using DepMap in this case has provided dramatically improved performance (10-15x faster throughput for some relevant workloads) and large cost savings relative to other options.
Acknowledgements

Thanks to Dr. Haungs, for his guidance throughout this work and enthusiasm for operating systems and the research process. Thanks also to Dr. Nico and Dr. Dekhtyar for their generous time spent in reviewing this work and providing additional guidance. Thanks to Dr. Elmo Keller for sharing his contagious interest in the behavior of systems. Thanks to the faculty of the Computer Science Department, past and present, for their combined support of hands-on learning. Many have benefitted from your efforts.

Additional thanks to friends in the Physics and Philosophy programs for their thoughts from other perspectives. The most important ideas span the disciplines.

Thanks to the Cal Poly Information Technology Services department, and to other employers and clients who have provided me with valuable experience in the challenges of designing and operating large distributed systems.

Deepest thanks to Heather for her conversations about all types of engineering, operations, and failure modes, and for caring for Zoe and me throughout this process. And thanks to Zoe for showing that it is all very simple.
Contents

List of Tables..................................................................................................................ix

List of Figures..................................................................................................................x

1. Introduction .................................................................................................................. 1
   1.1. Contributions of This Work.................................................................................... 2
   1.2. Example of DepMap Usage .................................................................................. 3
   1.3. Outline ................................................................................................................... 5

2. Background .................................................................................................................... 7
   2.1. Terms ..................................................................................................................... 7
   2.2. Software Engineering: Reuse, Coupling and Cohesion ........................................... 8
   2.3. Distributed Systems and Cloud Computing .......................................................... 11
   2.4. Systems Thinking, Design and Engineering .......................................................... 15
   2.5. Requirements Broadly Defined .............................................................................. 19
   2.6. Related Work ......................................................................................................... 21

3. Design ........................................................................................................................... 24
   3.1. Methods of Dependency Detection ....................................................................... 24
   3.2. Methods of Operating System Event Monitoring ............................................... 27
   3.3. Requirements and Use Cases ............................................................................... 30
       3.3.1. Establish a detailed inventory of actors and the connections between
              them ..................................................................................................................... 30
       3.3.2. Link together actors based on these connections to build a high level
              map ....................................................................................................................... 32
       3.3.3. Describe both connections and actors in detail ............................................. 32
       3.3.4. Provide data analysis and visualization methods .......................................... 33
       3.3.5. Moderate and tunable performance impact .................................................. 34
       3.3.6. Supported platforms ...................................................................................... 35
   3.4. Use Cases for Evaluation ...................................................................................... 36
   3.5. Challenges ............................................................................................................. 38
   3.6. Architecture and Detailed Design ........................................................................ 42

4. Implementation ............................................................................................................. 49
4.1. Data Collection ............................................................................................................. 49
4.2. Sysstat Collection ....................................................................................................... 49
4.3. DTrace Setup and Collection ........................................................................................ 50
4.4. Collector ....................................................................................................................... 58
4.5. Analyzer ....................................................................................................................... 58
4.6. Reporter ....................................................................................................................... 60

5. Results and Analysis ...................................................................................................... 64
5.1. Limitations on Operational Details .............................................................................. 64
5.2. Test Platforms ............................................................................................................. 64
5.3. Synthetic Testing for Basic Function and Overhead ..................................................... 65
5.4. Real-World Applications .............................................................................................. 69
5.5. Conclusion ................................................................................................................... 80

6. Future Work .................................................................................................................... 81

7. Bibliography ................................................................................................................... 85

8. Appendix A: Full Page Diagrams and Charts with Explanation ...................................... 90
List of Tables

Table 1: DTrace Probes by Platform................................................................. 51
List of Figures

Figure 1: Example Network Dependency Diagram ................................................................. 4
Figure 2: Cloud Application, Logical .................................................................................... 12
Figure 3: Cloud Application, Runtime ............................................................................... 12
Figure 4: Two Simple Systems ......................................................................................... 15
Figure 5: Web Server Dependencies - Detail ................................................................. 25
Figure 6: Web Server Dependencies - Aggregate ............................................................. 25
Figure 7: Unix Process Model ............................................................................................ 31
Figure 8: High Level Design of DepMap ......................................................................... 42
Figure 9: Map of Process, Thread, and TCP Socket Dependencies for Tomcat Startup .... 45
Figure 10: Line Chart of Cumulative I/O To Storage Device Set by Time (Read, Write, and Total) ........................................................................................................... 45
Figure 11: Scatter Plot of I/O to Storage Device Set by Time (Read, Write, and Total) .... 45
Figure 12: Architecture and Components for DTrace Usage ........................................ 47
Figure 13: Architecture and Components for iostat Usage ............................................ 48
Table 1: DTrace Probes by Platform .................................................................................. 51
Figure 14: Hierarchical Graph of Checking Email ............................................................ 61
Figure 15: Median of Sequential Read Throughput Across 5 Runs ................................ 62
Figure 16: CPU IOWait Status Versus Usr, System Time ............................................... 62
Figure 17: IO Wait Time Relative to Processing, Two Week Period .................................. 63
Figure 18: Histogram of Read Timing ............................................................................... 63
Figure 19: Scatter Plot of 1 Second Event Data .............................................................. 63
Figure 20: Visualization of Simple Read Test Case .......................................................... 66
Figure 21: Performance Impact from DepMap Monitoring, I/O Test ............................. 67
Figure 22: Network Transmission Time, Previous Environment .................................... 70
Figure 23: Network Transmission Time, New Environment

Figure 24: Proportion of System Time Spent IOWait - 1 Minute Samples

Figure 25: Proportion of Processor Time Spent in an IO Wait State - 1 Second Samples

Figure 26: Impact of Rescheduling Backups

Figure 27: Sequential Reads Against Device - Raw Timing

Figure 28: Sequential Reads on Device - Median Filtered

Figure 29: Sequential Reads on Device - Delta(Median Filter)

Figure 30: Sequential Read Throughput from Various Devices

Figure 31: Sequential Writes to Fill Device
1. Introduction

Software systems are built as a collection of abstracted layers. These layers shield developers and administrators from much of the complexity in the system, but at times this complexity emerges in the form of unanticipated behaviors such as slow performance or total system failure. The abstraction becomes a hindrance to understanding the runtime dependencies present in a system, and to diagnosing and resolving performance issues. As applications are increasingly deployed in distributed models, such as multi-tier web applications and cloud computing environments, runtime resource dependencies emerge as a frequent cause of system failures. These dependencies may be local, such as code libraries and configuration files, or remote, such as network services.

This work focuses on the issues of identifying runtime dependencies in existing applications, characterizing the behavior of these, and, where possible, making determinations about the type of each dependency and its impact to the application. The result of this work is a tool, DepMap, as well as detailed case studies of the effectiveness of this approach in diagnosing real-world performance issues in two quite different distributed systems.

A system administrator or developer may leave DepMap running on OS X, Solaris, and Linux systems to record information about access to files, network connections, and other inter-process communication mechanisms. Reporting and visualization tools included with DepMap may then be run to generate graphical representations of these dependencies and charts of relevant characteristics (e.g. volume of data per unit time and frequency of use). This information provides a basis for understanding how applications function and can be used to identify and resolve issues with performance and availability.

While many Unix distributions include tools to observe resource utilization, these provide narrow views and little possibility of integrating data to provide a high-level map of system behavior. For example, \textit{vmstat, netstat, ps,} and \textit{fuser} each provide point-in-time information about memory, network, process, and file usage respectively but with insufficient context to build a map of relations among processes. Further, because these tools produce data that is point-in-time or sampled at large intervals (seconds to minutes), relevant trends in may be missed. For example, extreme cycles below the sampling rate of
io\texttt{stat} are averaged away, while tools like \textit{netstat} are not practical for detecting network connections that occur intermittently.

Another class of tool, including \textit{strace}, \textit{truss}, \textit{auditd}, and \textit{SELinux} can be used to monitor and record event-based information. While these address many of the weaknesses of the sampling tools mentioned previously, because these tools are capable of reporting so many events (\textit{e.g.} system calls) their use must be carefully targeted and even so a large volume of data may be produced. The targeting and analysis effort involved can be substantial on a large production system.

\textit{DepMap} bridges these two types of tools, using the \textit{DTrace} toolset to collect event-based data that is well-suited to building a weighted map of system dependencies, with relevance up to the application layer. Where necessary, this data can be converted to time-series form at a finer resolution and narrower target than is typically possible with standard Unix tools. Thus, \textit{DepMap} provides the system administrator with both a holistic description of system behavior in terms of dependencies between processes and resources, and methods of drilling down to relevant details such as I/O operations performed per thread or network connections that occur on an infrequent basis. \textit{DepMap} shares a weakness of all of the tools previously mentioned in that it does not provide any mechanism for aggregating information from multiple hosts into a comprehensive view of a distributed system, however it does provide a basis for improvements in this area.

1.1. Contributions of This Work

The specific contributions of this work:

- A toolset for monitoring running processes and recording information about run-time dependencies between them.
- Visualization functionality to provide a graphical representation of process and resource connections within a system.
- Analysis functionality to weigh dependencies based on multiple factors relevant to that dependency type, for example volume of data and frequency of access for a file access dependency
- Analysis and reporting functionality to characterize these connections at a detailed level.
• Application of the toolset to actual systems, and insight gained into the workload of a web application and a data warehouse, performance of shared storage systems, and response characteristics of common network services including DNS and NTP.

1.2. Example of DepMap Usage

To illustrate the capabilities and typical usage of DepMap before I consider details, let us briefly consider a scenario and review the usage and outputs of DepMap. In this scenario, a system administrator has been tasked with reconfiguring a web application server so that it is listening only on port 80 (the standard HTTP port), and to disable listening on the common development environment port 8080, as well as any other ports that may be in use. No documentation is available for the existing configuration, so the administrator must begin by examining the system. For our example I will use the Tomcat application server, which is in common use and well documented. To understand the benefits of DepMap, consider that similar situations often arise with more obscure, undocumented software.

As a starting point, the administrator knows that the application server is initiated from an init script. Investigation shows the following (portions of paths and outputs are removed for readability, with typed commands shown in italics, "$" signifying a shell prompt, and "#" signifying a commentary on the step):

```
$ wc -l apache-tomcat/bin/startup.sh
  64 startup.sh
$ tail -l apache-tomcat/bin/startup.sh
exec "$PRGDIR"/"$EXECUTABLE" start "$@"
```

The script contains no obvious port numbers, and references another script. At this point, common administrator practice would be to run `grep` on the approximately 600 files in the `apache-tomcat` directory to find instances of all existing port numbers, so that they can be changed or disabled. Including matching log entries, the number of results would number in the hundreds even on a lightly used system with small logs. This would also require determining all of the relevant port numbers, for example by running Tomcat and then “netstat –a” to show listening ports. With DepMap, the process would instead be:

```
# generate the appropriate collection script for this host, accepting defaults
# this step is only necessary on the first run
$ dmap_gen
DMAP Script Generator 20100531
  Host is type “Linux”
```
Verified DTrace present as kernel module “dtracedrv”
Probes supported: 186331
No arguments, creating default script
Collector created as “dmap_watch.d” – complete

# run script to monitor all activity on system, then start Tomcat
$ sudo dmap_watch &
DMap: dmap_watch.d is now logging to dmap_trace_20100627.dmap
$ apache-tomcat/bin/startup.sh
   - Using CLASSPATH: apache-tomcat/bin/bootstrap.jar
$ sudo dmap_stop
DMap: logging to dmap_trace_20100627.dmap complete, 889 records generated

# check .dmap records for presence of port number, then
# create map of activity based on .dmap records
$ grep 8080 dmap_trace_20100627.dmap
dep 11567.255854 ::0:8080    1000
# process 11567 was listening on port 8080, with no IP mask applied
# DepMap assigned a relative importance of “1000” to this dependency (a high value)
$ dmap_parse --n dmap_trace_20100627_net.dmap | dmap_diagram
DMap: creating diagram dmap_trace_20100627.png, network only

Figure 1: Example Network Dependency Diagram

# show the top ten (highest rated) file dependencies
$ dmap_parse --f dmap_trace_20100627.dmap | head -10
dep 11567.255854 apache-tomcat/conf/web.xml 123
dep 11567.255854 apache-tomcat/conf/context.xml 67
dep 11567.255854 apache-tomcat/Catalina/localhost/manager.xml 33
dep 11567.255854 apache-tomcat/Catalina/localhost/host-manager.xml 33
dep 11567.255854 apache-tomcat-users.xml 16
dep 11567.255854 apache-tomcat/server.xml 15
dep 11567.255854 apache-tomcat/webapps/manager/WEB-INF/web.xml 12
dep 11567.255854 apache-tomcat/webapps/host-manager/WEB-INF/web.xml 12
dep 11567.255854 apache-tomcat/webapps/docs/WEB-INF/web.xml 12
At a glance (see Figure 1), the administrator can see that from a network perspective the startup.sh script is initiating a java daemon process (which is orphaned, and thus owned by init with a parent process ID of 1). This process has three threads, one of which listens on three ports including our port of interest, 8080. The ten highest rated file ("-f") dependencies are primarily .xml configuration files (the process for determining ratings is described in detail later). In this case, server.xml is the sixth highest-rated file and contains the relevant configuration parameter. Here, in a relatively simple use of DepMap, I was able to easily identify the relevant network sockets and configuration files. More advanced application of DepMap would allow the administrator to characterize traffic patterns of these ports, and access patterns associated with these files.

Additional synthetic and real-world case studies are provided later in Sections 3, 4, and 5 (Design, Implementation, and Results and Analysis).

1.3. Outline

This work is presented in this introduction and five primary sections, supplemented by a bibliography and appendices which provided additional detail. Section 2, Background, considers the current state of software systems and their structure from three perspectives: software design for reuse, distributed systems, and systems engineering. Together these perspectives build an understanding of the types of runtime dependencies present in software, the benefits and challenges associated with these dependencies, and the approaches which can be used to mitigate the complexity and risks they create in large software systems. This section includes a list of the general requirements for a tool to map dependencies, and an overview of possible approaches and related work in this area.

In Section 3, Design, I describe detailed requirements. Use cases are presented to explain the two primary functions of DepMap: identifying the existence of runtime dependencies, and characterizing dependencies of interest. I then describe some of the design and implementation challenges which come from these requirements. I explain the major components of DepMap including those that perform data collection, data analysis/transformation, and reporting. I describe design tradeoffs motivated by the requirements and challenges identified in the previous section.
Section 4, Implementation, summarizes the work I performed to implement DepMap functionality across a range of operating system platforms. It also describes some approaches I used to automate the process of porting to three different Unix-like operating systems, and to ensuring appropriate and equivalent data collection coverage was implemented on each.

Section 5, Results and Analysis, demonstrates the functionality of DepMap on synthetic and real-world scenarios. Synthetic testing is used to establish correctness and coverage of DepMap and to measure the performance overhead associated with different modes of usage. I then provide extensive description of how I used DepMap on two very different real-world scenarios: mapping the dependencies in a complex web application, and characterizing the workload of a large database server to solve major performance problems. The results of these real-world scenarios demonstrate the utility of the DepMap approach on a wide variety of system performance issues.

I close by considering future work for DepMap, including additional data collection, analysis, and action capabilities, as well as application to other problem spaces not yet tested.

The appendices provide detailed example output from my testing of DepMap in real-world case studies. I refer to these from Section 5, Results and Analysis, as many of the diagrams and charts require full pages for readability and thus do not fit within the flow of the main text.
2. Background

Computer systems are complex, and that complexity can be abstracted but not removed. I will review concepts from software engineering, distributed systems, and systems thinking to build a broad understanding of dependencies. I will explore how and why dependencies are introduced, some of the general approaches that may be used to understand and manage dependencies in distributed systems, the broad requirements for DepMap to identify and measure dependencies, and existing work relative to DepMap.

2.1. Terms

I define some terms as they are introduced. Terms that are used with great frequency or are central to this work are defined below:

**System**: Broadly, a set of certain things together with the relations between them (Klir, 1985). Also, a “way of looking at the world” (Weinberg & Weinberg, 1988). More specifically for this work, a collection of software and hardware components, the connections between them, and the behaviors perceived by those who work to manage the system.

**Module, element, actor, component**: a piece or thing within a system, in the context of this work computer hardware or software in general and running processes in particular.

**Connection, relation**: an interaction or communication between elements, or the explicit potential for communication. Klir offers the additional terms: constraint, structure, interaction, coupling, linkage, and correlation (Klir, 1985). In the context of this work, this includes files, network connections, pipes, shared memory, and other communication mechanisms.

**Dependency**: A connection, and the expected behavior of that connection as required in the operation of a system.

**Production system**: a system that directly supports delivering service to users, for which expectations on levels of performance are strict.

On storage units: I use power of ten units when referring to storage on disk (e.g. 1MB is \(10^6\) bytes), and the power of two units for space in RAM and I/O block sizes (e.g. here 1MB is \(2^{20}\) bytes, or more precisely 1 Mebibyte or 1 MiB). This follows common conventions.
2.2. Software Engineering: Reuse, Coupling and Cohesion

“Efficiency stems more from good design than from good coding” (Glass, 2003).

In order to understand how runtime dependencies are introduced into software, let us begin by considering how software is designed, and especially how software is structured for abstraction and reuse.

The roots of software engineering as a discipline can be traced back to the development of the first general purpose computing systems and high-level languages (Brooks, 1995). The goal of a structured approach to software design and development is to provide specified functionality (meet the requirements) using an acceptable amount of resources (both human, for development and maintenance, and hardware for computation, storage, etc.)

Early computing systems provided very limited computational, memory and I/O resources with respect to the problems that were being solved. This caused emphasis to be placed on efficient code: machine or assembly code, optimized for the specific task at hand. As hardware capacity increased, and similar processing was required in a range of applications, reuse of code and ease of maintenance became more important considerations. Software complexity also became an issue, and the abstraction that comes with reusable components masks much of this complexity. The expected benefits of code reuse and abstraction are many and well recognized, and include: reduction in developer effort, increased code quality and improved readability. Conceptually, both goals are accomplished by breaking code into well-defined pieces. In specific contexts, these may be referred to as modules, components, procedures, functions, or services. As Glass notes, “Reuse-in-the-small... is a well-solved problem” (Glass, 2003).

Many paradigms have been developed that encourage software design for reuse. Design patterns (Gamma, Helm, Johnson, & Vlissides, 1994) offer standardized approaches for achieving reusability and flexibility, and describe abstract models with wide applicability for how software components interact. Web services (and related concepts of “software as a service” and “cloud computing”) emphasize standardized application programming interfaces (APIs) implemented as a distributed system. In this approach, the API hides implementation complexity, while the deployment promises decreased systems
management complexity (more on this later). As Bloch has described, “all programmers are API designers. Good programs are modular, and inter-modular boundaries define APIs.” This is not without some risk, however: “APIs can be among your greatest assets or liabilities” (Bloch, 2006).

All of these approaches build on a common, well-established conceptual framework. Yourdon and Constantine describe many of the considerations for obtaining the most value from reuse, and provide cautions as to the risks. Critical concepts include coupling and cohesion (Yourdon & Constantine, 1979). Coupling is defined as “the degree of interdependence between two modules”, and the objective is to “minimize coupling ... to make modules as independent as possible”. A well-partitioned system is desirable, and can be obtained by eliminating unnecessary relationships, reducing necessary relationships, and easing the “tightness” of necessary relationships. The benefits of this include reducing the chance of ripple effects (in which a defect in one module appears as a symptom in another), ability to change one module while having minimum impact on others, and providing useful functionality while abstracting (hiding) implementation complexity (Page-Jones, 1988).

Page-Jones outlines five principles to reduce coupling, based on Yourdon and Constantine. Each principle describes a desirable attribute for connections (relationships) in a system: narrow, direct, local, obvious, and flexible. The last four of these are particularly relevant to my work. Developers more easily understand direct connections, while indirect (broad) connections hide information that is relevant to understanding a system. Local connections are concentrated in space (e.g. lines of code, data items) and time (e.g. related values and processing occurring in sequence), while remote connections again create complexity by hiding information. The often-cited issues with global variables illustrate the problem of remote connections.

Obvious connections, by definition, are easily understood. This principle can be illustrated through examples of obscure connections: in the realm of database design, it is good practice for primary keys to be opaque, that is, without special significance other than their association with application values (Silberschatz & Korth, 2002). A key that includes a user’s last name encourages obscure connections, as application code may begin extracting the name from the key, rather than explicitly requesting it from the database (and also, names change!)
Flexible connections increase the likelihood that new use cases can be implemented without significant modification to modules. Rigid connections, in contrast, often involve unnecessary limitations to a module's interface. For example, a module that accepts last names as arrays of eight characters, as in ['S', 'm', 'i', 't', 'h', null, null, null] is less flexible than one that accepts last names as a String type without explicit bounds. This allows for longer names, with alternate character sets, as in "Rāmānujan". Note that this does not preclude input validation, but rather suggests that limitations should be explicitly handled in code rather than implicitly “baked in” to the connections.

In practical design, these connection properties are expressed in three broad classes of coupling: normal, common, and content. Within these broad classes are many variants: normal, data, stamp, control, common (alias global), content (alias pathological), message, external, and others (Page-Jones, 1988). It is out of the scope of this work to review the details of each type. For present purposes, please understand that the type of coupling between two modules is determined by the “worst” (most tight) coupling they exhibit, and that all of these types are determined by the degree to which they demonstrate the desirable properties for connections. Of particular relevance for this work are message coupling and, related, external coupling. Message coupling is one of the loosest (“best”) types, where a well-defined public interface is used to exchange messages. External coupling consists of modules sharing externally imposed data formats or communication protocols.

Another way to measure the modularity of a system is cohesion. It is defined as “the measure of the strength of functional relatedness of elements within a module” (Yourdon & Constantine, 1979), or simply as “module strength.” Strong (independently useful) modules by their nature reduce coupling by minimizing both necessary and unnecessary connections.

Implicit in all of this is abstraction. Complexity is repeatedly encapsulated and abstracted as we move up the software stack from operating system through libraries, frameworks, and application code. The complexity is not removed, however, and these levels of encapsulation can make it difficult to diagnose functionality or performance flaws in software. Developers familiar with large software frameworks will recognize the pattern of following function calls through layer upon layer of code, increasingly distant from the proximal use of the framework by the application. “The ... problem here is software
observability, or more accurately, the pronounced lack of it. We have built mind-bogglingly complicated systems that we cannot see, allowing glaring performance problems to hide in broad daylight in our systems” (Cantrill, 2006).

2.3. Distributed Systems and Cloud Computing

“Not all is delight, however, and knowing the inherent woes makes it easier to bear them when they appear.” (Brooks, 1995)

Coulouris et al define a distributed system as “a collection of autonomous computers linked by a network, with software designed to produce an integrated computing facility” (Coulouris, Dolimore, & Kindberg, 1994). When this definition was written, the world wide web was in its infancy, and the terms “software as a service” and “cloud computing” had not yet been coined. Since that time the Internet has become a dominant deployment platform for applications. These applications are based on standard protocols, data formats and development models including hypertext transfer protocol (HTTP), extensible markup language (XML) and asynchronous JavaScript and XML (AJAX). There is little doubt that these technologies will continue to evolve and be replaced but the fundamental issues of distributed systems will remain. Distributed systems will be ever more common.

Distributed systems can be considered as a subset of software systems. The concepts of coupling and cohesion apply for the local (node) level, as well for the entire system. Additional complexity arises at the system level. Issues that are challenging to manage on a single node become even more significant in a highly distributed system. Concurrency, latency, and fault tolerance (resiliency) become primary concerns, even in relatively simple systems.

The term “cloud computing” is currently in use to describe systems in which reusable runtime components are not only distributed, but are operated by external organizations. These range in complexity from simple key-value databases to entire (virtualized) hosts running multiple services. As in the classical structured system design model each component exhibits coupling and cohesion in code. In addition to this there are now runtime connections between components in the system. An application implemented using cloud computing services will likely exhibit two of the “best” types of coupling from the classical design model: message and external. This means well-defined public interfaces for functionality. As such, the connections will tend to be narrow, direct, obvious, and
**flexible** from a *design perspective*. This is one of the reasons that this model has become popular: it encourages and benefits from clear APIs.

Two broader reasons are operational flexibility and service specialization. Operational flexibility means that the run-time deployments of components can be readily substituted. For example, an application that uses a key-value storage service may be readily moved from one implementation to another. Service specialization means that a company can focus on providing a fast, reliable, cost-effective implementation to a large number of customers. In practice, the early examples of this have been from companies such as Google and Amazon that use these services internally. From an economic perspective, there is potential for application owners (consumers) to benefit from low entry costs and predictable, incremental costs to scale larger user populations. Service providers are able to build larger, more cost effective data centers and infrastructures by sharing these resources with their paying customers.

**Figure 2: Cloud Application, Logical**

**Figure 3: Cloud Application, Runtime**

Figure 2 shows a conceptual model of a simple distributed system based on four cloud services: application server, federated authentication, binary file distribution, and a key/value database. This standardization and abstraction simplifies some aspects of service delivery: the providers manage operational concerns like hardware failover, backup, and recovery. Resources can be added on demand, in units of computation or space, to handle peak loads.
At the same time, this approach introduces new complexities: trust in the providers to actively manage these issues, potential for competing consumers to impact shared resources, and a much larger number of actual implementation components that are hidden by the abstract model. Figure 3 shows a possible runtime implementation of the conceptual model, and some types of inherent complexity that are hidden from the consumer: internal single points of failure, hidden dependencies between services, and impact from other consumers. These are in addition to hidden dependencies within the application code itself, an increasingly common phenomenon as third-party software packages (open source and commercial) increasingly assume “always on” network connectivity to function, without making these requirements explicit.

Let us consider three scenarios for the above diagrams. In the first, Application “A” relies on the key/value service to map a user’s identity to content intended for that user. Internally, the key/value service provider may be dividing keys across a set of servers, such that when one server is unavailable most users see uninterrupted service, while a small subset of users see a total application outage. In the second scenario, Application “A” stores the user’s content in the file store service. Internally, the provider mirrors this content for performance and availability. Application “B”, unknown to “A”, shares many of the same servers, and sees a large peak in demand from its users leading to sudden, greedy consumption. Though the “A” content is mirrored, all of the relevant mirrors are affected by the peak demand from “B”, and service to “A” becomes consistently slow. In a third scenario, the key/value service itself depends on the storage service. Issues with any one service not only cause direct impact to “A”, but additionally an amplified indirect impact due to their interdependency.

In each of these scenarios, transient conditions at the service provider leads to impact that ranges from subtle to significant. Service level agreements often specify average and worst case performance, and are used to set expectations between service providers and consumers, but these will rarely address all possible impacts to the application. In any case, the consumer will likely wish to monitor not only average performance, but to understand specific dependencies and measure the varied performance seen from them under real-world scenarios.

As distributed computing began to grow quickly in the early 1990’s, Peter Deutsch identified seven common Fallacies of Distributed Computing while working at Sun. In 1996
James Gosling (the creator of Java and also at Sun) added an eighth. These are explained by (Rotem-Gal-Oz, 2006). The fallacies, expressed as assumptions about the computing environment, are:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Some of these capture technical issues, some involve organizational and support issues, and while all apply to cloud computing I suggest that numbers five, six and seven are especially relevant. From the perspective of software design and economics the distributed and cloud models often make good sense, but each of the fallacies must be understood and mitigated through design and deployment. Issues of concurrency, resource management, latency, and fault tolerance must be understood and addressed. This thesis focuses on programmatical identifying the connections present in such a distributed system, determining how they relate to application functionality, classifying them based on importance to the system as a whole, and building a model for how an application may perform when the fallacies are realized (e.g. when latency goes high, bandwidth drops, the network topology changes without warning, or changes and complexity are hidden as they are handled by other administrators).

Real-time systems are a subclass of systems that directly address these same issues, typically by providing less flexibility and ability to scale in order to reduce complexity. This is accomplished via e.g. hard constraints on shared resources and limitations to concurrency. The goal for most distributed systems is less strict: to provide for predictably good performance, and not to prevent most types of runtime problems but rather to detect and mitigate problems quickly. The economic basis for distributed systems is to accomplish more with shared resources.
2.4. Systems Thinking, Design and Engineering

“The only difference between systems that can fail and systems that cannot possibly fail is that, when the latter actually fail, they fail in a totally devastating and unforeseen manner that is usually also impossible to repair.” Murphy on software as recounted in (Weinberg G. M., Quality Software Management: Systems Thinking, 1992)

Extending beyond software and disciplines in engineering and science, systems science, thinking and engineering approaches are applicable to understanding a variety of phenomena. A full consideration of systems science is outside of the scope of this thesis, and is covered well by many of the sources referenced, particularly (Klir, 1985) in a detailed manner, (Meadows, 2008) in an accessible manner, and the works of Weinberg as they relate directly to the design and support of computer systems. I will note a few concepts that are relevant to what is described in the previous section.

Meadows defines and describes systems: “A system is an interconnected set of elements that is coherently organized in a way that achieves something. ... A system must consist of three kinds of things: elements, interconnections, and a function or purpose. ... [it] may exhibit adaptive, dynamic, goal-seeking, self-preserving, and sometimes evolutionary behavior (Meadows, 2008). The parallels between the general (elements, interconnections, function) and the specifics of distributed computer systems (nodes/components, APIs/networks/dependencies, and requirements) are clear.

Meadows defines and describes systems: “A system is an interconnected set of elements that is coherently organized in a way that achieves something. ... A system must consist of three kinds of things: elements, interconnections, and a function or purpose. ... [it] may exhibit adaptive, dynamic, goal-seeking, self-preserving, and sometimes evolutionary behavior (Meadows, 2008). The parallels between the general (elements, interconnections, function) and the specifics of distributed computer systems (nodes/components, APIs/networks/dependencies, and requirements) are clear.

Figure 4: Two Simple Systems

Figure 4 depicts two simple systems. Each has four nodes, or actors, but the systems vary in the relations (dependencies) between the nodes. The one on the left is serial in nature, with each node depending directly on at most two other nodes. Interaction to actors outside of the system is via node A. The system on the right is hierarchical in nature, with node A depending directly on all other nodes. Four concepts can be illustrated with these systems: first, there is inherently a concept of system boundary, that which is within the
system. If some actor Z communicates only rarely with A and has little impact on the system, then the above diagram holds. If Z communicates regularly with A, and with high impact (or dependence) on the system, then Z might be more appropriately considered to be part of the system. Taking this to the logical extreme, Hans Bremermann described what is now known as Bremermann’s Limit: “No data processing system, whether artificial or living, can process more than $2 \times 10^{47}$ bits per second per gram of its mass.” The size of the system is constrained by the computational capacity of the mass and energy of Earth, or of the known universe.

Second, some level of abstraction is inherent in defining actors within a system. D, for example, may be internally composed of $D_0..D_n$ in an unknowable subsystem configuration. Similarly, these systems could be considered as $S_0$ and $S_1$, each encapsulating the shown actors.

Third, the use of each connection is as important as its existence. If, in the system shown in the right of Figure 4, node A makes a request of node B, then hands the result to node C, and hands that result to node D then both systems behave in a serial manner. Thus, both the existence and collective behavior of connections is important in modeling a system.

Fourth, each system has measurable complexity. Klir provides two principles for this: (1) “the complexity of a system should be proportional to the amount of information required to describe the system”, and (2) “the complexity should be proportional to the amount of information needed to resolve any uncertainty with the system involved” (e.g. predictive) (Klir, 1985). This also goes to the question raised by the second point, above, of what level of abstraction is acceptable in modeling a system. Pragmatically, “Model utility depends ... on whether [the model] responds with a realistic pattern of behavior” (Meadows, 2008).

When attempting to understand the elements of an existing system, relevant concepts include cumulative failures, latency, redundancy, feedback loops, buffering, averages versus extremes of behavior, and limitations of modeling. Cumulative failures are variations from expected behavior that increase with the number of observations or operations. In the context of a distributed system, this can be thought of in terms of a chain of nodes each consuming requests from the previous and supplying to the next. When one of the nodes fails to behave as expected, all subsequent consumers will be affected, and the affect may increase with each consumer. In situations where the failure is obvious and explicitly
handled, its impact may be reduced by a consumer, but more typically the error will either result in a failure obvious to the outside user (fail fast), or one that will linger and become amplified over time (silent or masked failure). It is generally not practical or expected to remove all errors from a system: “Residual errors will always persist. The goal should be to minimize or eliminate severe errors” (Glass, 2003).

An example of this is latency: a node may occasionally take 100 times as long as normal to produce a response. Depending on design criteria, this might be considered a failure. If a consumer times out after a short predefined time, we have failed fast. If the consumer waits, then at best all subsequent processing will be delayed. Validating the performance of a system is difficult with regards to latency and other factors that are at the mercy of components not directly within our control.

Redundancy is often used to improve the resiliency of a system, by mitigating node failures. By adding additional nodes, dependence on one node may be reduced. This technique also introduces additional complexity, however, as there are now more nodes with the potential to fail. Redundancy can also increase the presence of feedback loops, as if the redundant nodes must sometimes be interconnected, and this series of connections can lead to an actor indirectly depending on itself. This can be beneficial, as in cases where it moderates system activity and leads towards steady states (sustainable operation), or detrimental as in cases where it magnifies activity and leads to an unsustainable state (e.g. the “death spiral” that can occur with race conditions or the increase in duplicate requests that can come from users losing patience with a heavily loaded system).

Buffering also improves the resiliency of systems against some types of failures, and also increases complexity. Buffering provides an increase in the time over which an actor can act without being affected (or in some cases aware of) failures in a dependency. There is potential for failures to be masked, however, past the point where a system can respond appropriately.

When modeling a system it is important to understand not only the average or typical performance of nodes and connections, but also the extremes of behavior. Aggregation is a powerful modeling tool, but over-aggregation can cause a model to be useless, or worse, misleading. This is related to the Shannon-Nyquist sampling theorem, which has the result that a signal (or behavior of a dependency in a system) cannot be described by a series of discrete measures sampled at less than the frequency of changes in the signal. In a sense,
this is what systems science is about: understanding, with a necessarily limited model, the operation of a system under all possible conditions, including those where there are frequent fluctuations.

Returning to the first concept, of system boundaries, in any computer system it is important to consider the users and managers of the system, even if they are not explicitly modeled. Humans are resilient and adaptive, but often unpredictable (Weinberg G. M., Quality Software Management: Systems Thinking, 1992).
2.5. Requirements Broadly Defined

"Understanding the existing product is the most difficult task of maintenance" (Glass, 2003).

The requirements for DepMap are based on the issues identified in the previous three subsections. Given an existing heterogeneous distributed system composed of nodes running Unix-like operating systems, DepMap should provide application developers and system administrators with an inventory of the runtime dependencies (file and network resources, and other common mechanisms supporting inter-process communication) for a given host, and measures of the behavior of these dependencies. This is accomplished by observing activity on one or more nodes under synthetic benchmarks or real world workloads. For the purposes of this work I focus on Linux, Solaris, and Mac OS X (speaking generally, BSD services on top of a Mach kernel) but with the desire that much of the work be extendable to other operating systems that can provide a consistent view of low-level resource access events.

The heterogeneous nature of distributed systems means DepMap must make few assumptions about the application development technologies in use on each node, and that the raw data collected to build a dependency map equivalent must be available on all node types. Where the forms of equivalent data or resource invocation differ by node type, DepMap must translate into a common format. The availability of source code for the operating system, applications, and supporting infrastructure cannot be assumed.

In systems thinking terms, DepMap must:

1. Establish a detailed inventory of actors and the connections between them.
2. Link together actors based on these connections to build a high level map.
3. Describe both connections and actors in detail, and, for each, estimate measures of importance to the correct operation of the system (in other words, the likely impact to the system if a given actor or connection should fail). This is accomplished by the following features, which together require that DepMap be able to dynamically adjust what is monitored without a system restart or any gap in information collection:
   a. Always noting the existence of connections to relevant actors to build a complete map.
b. Sometimes monitoring the content of a connection, or metadata associated with it, when used in a targeted assessment scenario.

c. Sometimes injecting unusual behavior into connection (e.g. adding a delay, or causing an error to be returned instead of data), when used in a repeatable test scenario.

4. Provide data analysis and visualization methods to support interpretation of results by developers and system administrators.

5. All of the above must be accomplished with at most a moderate impact to performance of the system, and excepting the case of intentional failure injection, with extremely limited potential to introduce system failures.

I will refer back to these broad requirements when considering methods of detecting runtime dependencies, and methods of operating system event monitoring.

Result data should be stored in a primary format. Tools should be included to translate this data into forms that support human understanding, including diagrams of dependencies and charts of dependency performance (on time as well as frequency bases), as well as system configuration (e.g. firewall rules). These tools should be extensible, so that additional output formats can be added later, perhaps including a hyper-linked and correlated repository of dependency information from nodes throughout a system.

The emphasis of this effort is on analysis of system performance based on a set of collected data. The design should allow for future integration into an event detection and response triggering system, such that systems can be reconfigured to maintain acceptable performance and availability. It should be safe to run DepMap on a production application with a significant portion of it's functionality enabled, that is, with minimal risk of disabling the application or severely impacting server resources.

While DepMap is initially intended to support efforts to improve reliability and performance, understanding dependencies is also critical when evaluating and increasing the security of a system. As noted by McGraw: "...possible security issues [may be introduced] due to unintentional or inappropriate control coupling." More broadly, "In the coming days of Service Oriented Architectures (SOAs), understanding which services your code is counting on and exactly what your code expects those services to deliver is critical. Common components make particularly attractive targets for attack. Common mode failure goes global."
2.6. Related Work

Many anti-virus and anti-malware products intercept system calls. This is done to allow checking of I/O results against data patterns for known threats (signature-based detection), as well as to establish baselines for normal process activity and detect abnormal activity (behavior-based detection). The focus in this approach is more on the data being transferred and the processes performing that access than on the shared resources and implied connections between processes.

Some commercial products, such as VMware Beehive, monitor network traffic latency for hosts. This is used to determine when a service crosses a performance threshold so that additional resources can be allocated to that service. In the case of Beehive, additional virtual servers are initialized on demand to share load. This is similar to DepMap, in that both monitor the behavior of services using black block techniques, though Beehive is server/service focused and uses network activity as an indicator while DepMap’s goal is to identify both endpoints and it uses system call activity.

There are many toolsets for sampling and charting time-series operating system metrics. RRDTool focuses on storage and graphing of time series data (Oetiker, 2009) while Cacti builds on RRD with improved polling, graph templates, data acquisition, and management features (The Cacti Group, 2010). The focus of both is reporting on fairly static sets of resources, e.g. reporting metrics for 40 servers over a period of months. The initial emphasis for DepMap is batch analysis and reporting of dependencies, which may vary greatly on each run. Time series outputs from DepMap are associated with a mapped dependency, which may exist for months, or for less than a second. The results of mapping with DepMap could inform creation of additional monitoring targets within a tool like Cacti.

The sysstat package for Linux provides rich implementations of the standard Unix monitoring tools such as iostat (Godard, 2010). I use some of these for the Linux port of DepMap, and note that for pure time series analysis some of the functionality I implement is available in recent versions of the sysstat tools. The sysusage package, also for Linux, provides summary reports of activity based on output from sysstat tools, and uses RRDtool for reporting (Darold, 2010).

Application metric tools generally focus on either development (profiling, debugging) or deployment (log event reporting, application level resource utilization) but do not generally
provide for real-time analysis relative to low-level operating system metrics. For example, the profiling tools in the Eclipse IDE allow for counting and timing the invocation of a Java method that is already known to perform I/O tasks, but does not measure the I/O itself or identify remote participants (Popescu, 2006). An exception to this is the Instruments application included by Apple with its Xcode developer toolset. Instruments is intended to be used by developers to monitor system and application performance, and provides visibility into both application level and operating system events (Apple, 2010). Like DepMap, Instruments uses DTrace to perform data collection.

Dependency mapping occurs in a narrow way in many areas of software development and system management. With shared code libraries common on many operating systems, a library dependency list (map) is built into executables and used by the operating system to satisfy calls to shared libraries (Bovet & Cesati, 2005). Many dependency mapping tools focus on either ensuring that local runtime dependencies are met (e.g. RedHat RPM for package and deployment management), or documenting dependencies (via manual update, or scanning of configuration files). Bacon Map is a tool for documenting and visually displaying known dependencies between hosts, routers, storage, and other infrastructure devices (Zenisek, Carr, & Sanchez, 2010). As such it shares much of the intent of DepMap, though it focuses more on storing and reporting dependency data, where DepMap focuses on detecting dependencies and characterizing them. I discuss possible combined use of these tools further in Section 6, Future Work.

ZZUF is a “multi-purpose fuzzer,” which corrupts user-contributed data. It intercepts file and network operations, and by configuration it deterministically targets data for bit changes. The intent of this is to discover security flaws in the form of insufficient input validation. It operates using the LD_PRELOAD mechanism to intercept calls to library functions including variants of open and read. This approach is similar to DepMap, and for targeted mapping of specific processes suggests an alternate method to system call interception (Hocevar, 2010).

Members of the original DTrace team at Sun Microsystems (now Oracle Corporation) have created highly instrumented storage servers (Oracle Corporation, 2010). The published results of their work show many similarities to the work I have done to measure I/O performance in dependencies on storage devices (Gregg, 2010). Where their work emphasizes monitoring at the storage layer, and visualizations in particular, my work takes
place at the server layer and thus has more contextual information on application dependencies, and extends to dependencies beyond storage.
3. Design

In this section I explain the options I considered and decisions I made for the design of DepMap. I describe some simple systems containing dependencies in order to illustrate these decisions. I then explain the full requirements for DepMap and its intended use cases. I outline some of the challenges in meeting these requirements, and conclude with the design I selected for collection, analysis, and presentment of data.

3.1. Methods of Dependency Detection

“The basic idea here is to understand what kind of assumptions you are making about outside software, and what will happen when those assumptions fail (or are coerced into failing). When assumptions fail, weaknesses are often revealed in stark relief” (McGraw, Chess, & Migues, 2009).

I considered three approaches to mapping runtime dependencies. In order to describe these approaches, I will use the simple example of a web server with locally stored content. This server obtains its configuration information from a file, accepts connections on TCP port 80 and spawns worker threads to handle requests. Workers parse requests, and based on these read files from local disk and send content to the requestor over a TCP socket. Summary information for each request is written to a log file. Conceptual maps of the dependencies for this system are shown below. In Figure 5 we see a more detailed view with worker threads shown. In Figure 6 we see this simplified to one abstract “web server” and all of its connections in aggregate (across all processes and threads).
In a semantic approach a tool might scan the source code for the web server to find all statements that are likely to introduce runtime dependencies. This would require access to the source code for the web server, the configuration files, and potentially all supporting libraries and the operating system. This approach would provide substantial detail about the behavior of the software, but is not practical for a heterogeneous system involving numerous languages, and where source code may not be available. Developers often use this approach when they already have clues to a specific problem, and have time to manually review source code looking for relevant statements.

In a signature-based approach a catalog of known software packages would include information about common configuration file elements for this type of web server. Signatures (patterns) would identify the configuration line specifying the location of the log file, that client connections will be accepted on port 80, and that content may be accessed from the directories where f1, f2, and f3 are stored. The signature approach would not be able to provide information about which clients actually accessed the web server during a given period of time, or that f3 was the most requested file but that these requests failed due to restrictive file systems permissions. This approach is efficient for providing high-level information about configurations involving well-known software, but provides little information about actual run time use. It may miss those dependencies that are least likely to be currently understood by administrators who have already examined configuration files, and is ineffective on new or uncommon software for which no signatures exist. This appears to be the primary approach used by most commercial application dependency mapping software, though it is sometimes supplemented with traffic monitoring.
An event-based approach observes software as it is running with a synthetic or actual workload, and records the behavior of the software at an application or system layer. At the application layer, debuggers such as *gdb* use this approach to provide visibility into the execution of a program. Call parameters and return values can be observed. Modification of named variable and line-by-line execution can also be observed if debugging information is compiled in to the program. As with the semantic approach, understanding runtime application dependencies would require tracing calls through to libraries and the operating system to understand which actually result in a connection to another actor. As with the signature approach, monitoring at the application layer would potentially require specific implementation work for each application technology (e.g. Java, Perl, PHP, Python). In practice many of these depend eventually on the standard C library and efforts could be focused there at the cost of giving up access to some application-specific information.

I elected to use the event-based approach at the operating system layer, by monitoring system calls. While this forgoes some of the specific knowledge of application intent associated with the other approaches previously described, it ensures that all connections are identified while observing a minimum of unrelated events. It also has the advantage of leveraging the relative standardization of system calls via POSIX for most Unix-like operating systems. The operating system mediates connections between running processes (e.g. pipes, shared memory, local sockets, signals), and between processes and resources (e.g. files, remote sockets). For example, while there may be many variations of the *open()* call on Linux and Solaris (e.g. to support 64-bit file systems), they sufficiently similar be trivially mapped to a common meaning: an observed process (actor) just asked the operating system to create a file descriptor based on a path name or a connection request from another actor. The number of system calls on most POSIX systems numbers in the hundreds, with a subset being sufficient to identify connections. An approximation of this subset is easily found by identifying all system calls that manipulate file or socket descriptors.
3.2. Methods of Operating System Event Monitoring

As described above, this work focuses on monitoring of application behavior from the operating system level. Most operating systems include at least rudimentary debugging tools, and most Unix-like operating systems allow for additional monitoring either via changes to source code or addition of modules or drivers that intercept and report activity.

Kernel-level debugging is available for many Unix-like operating systems, but typically requires a custom kernel with debugging enabled, and this is often advised against for production use. Availability of tools is restricted for some commercial operating systems, and licensing terms may preclude full examination of system operation (e.g. digital rights management). Kernel debugging tools tend to focus on very low-level events, such as modification of memory and registers, in order to understand the internal operation of a system call or kernel event handler. This level of detail is not well suited to monitoring application behavior. Kernel probes provide higher-level information, but are not consistently implemented between different operating systems. Unix-like operating systems vary significantly in the implementation of their kernels (e.g. Linux's relatively monolithic kernel versus the less monolithic Mach microkernel in OS X), requiring more platform-specific code for an application-level monitoring tool. For these reasons, I did not consider kernel-level debugging or probes to be viable data sources for DepMap.

SELinux is an extension to the Linux kernel that allows for monitoring and control of the flow of information. It implements the mandatory access control model, in which the operating system constrains the operations that may be performed by a process. It can also be run in a non-enforcing mode, which is commonly used to establish correct operation before enabling full enforcement (Smalley, Vance, & Salamon, 2006). In this monitoring mode, SELinux could be used to monitor access attempted by a process. Given the platform requirements for DepMap I did not consider this option further, but noted that the results from DepMap could also be of use in generating enforcement rules for use by SELinux.

As described previously, I have focused on system calls as a standardized tool for monitoring connections between actors. System administrators on Solaris are familiar with the truss tool, and on Linux with the equivalent strace tools. Both tools allow for monitoring of system calls and signals at the level of an individual process and all of its children and threads. These provide sufficient information to support requirements 3a and 3b in Section 2.5, including providing the full contents of buffers used in calls to read() and write(). They
are not suited to injecting flaws or changing behaviors of system calls. These tools are designed for use against a single process and its children; they are not well suited to monitoring an entire server. This could be accomplished by running a separate trace against each process, but this would bring significant overhead to a server running hundreds of processes. Further, neither is available for OS X. Given these limitations, truss and strace are suited to a subset of the functionality required for DepMap, and may be appropriate for use in some modes but are not alone sufficient.

OProfile (Levon, 2009) is a system-wide profiler for Linux systems, designed to have low overhead and allow monitoring of resource consumption as well as call-graph level activity for processes. No kernel-level patches or recompilation are required. Debugging information is not required. It is not currently available for other platforms, and thus was not considered further. I noted that IBM appears to offer a similar tool, ProbeVue, for use on AIX.

SystemTap (Red Hat, IBM, Intel, Hitachi, Oracle, 2010) is a framework for monitoring events on Linux platforms built on top of kernel probes. This framework includes a scripting language that can be used to selectively collect, aggregate, report, and impact system behavior. It appeared to be an ideal candidate for use by DepMap, but with two limitations: it is not currently considered safe for use on production systems (perhaps due to fundamental design decisions), and it is available only on Linux systems. As with strace/truss, SystemTap appears to be a partial solution to meeting the data needs of DepMap, or a nearly full solution in a Linux-only environment that can tolerate some risk.

DTrace (Sun Microsystems, 2008) is conceptually similar to SystemTap, in that it is a framework and language for monitoring system behavior. Developed at Sun in 2001 and now an open source project also supported by Apple, it is well documented. SystemTap is considered by some to be based, at least in function, on DTrace. While DTrace does not provide as many facilities for monitoring all possible events in a system, it is capable of monitoring and impacting system calls and all other relevant events. On Mac OS and Solaris it effectively replaces the strace/truss types of tools, and the framework is used internally to implement many of the standard Unix system monitoring tools.

Similar to the Java Virtual Machine, DTrace interprets scripts written in the "D" language, performs safety validations, and handles any run-time errors. DTrace includes a concept of destructive versus nondestructive actions. With destructive actions disabled
(the default behavior), it is considered to be safe for use on a production system. With destructive actions enabled, D scripts may perform such actions as: invoking executables, modifying memory locations, setting breakpoints, triggering a kernel panic and delaying action for a specified time period. On the Macintosh platform, standard dtrace tools are available as of operating system version 10.5 and are the basis for the Instruments graphical profiling tool (Apple, 2010). On the Linux platform, Paul Fox has ported dtrace as a dynamically loadable kernel module. This work is incomplete, and the software is not considered to be production safe on this platform, but it does provide all necessary functionality to support DepMap requirements and within these limitations functions well on specific versions of the Linux kernel (Fox, 2009). I selected DTrace for use in creating the primary information provider for DepMap, as it best provides for all of the broad requirements.

In cases where additional detail at the application level is desired, DTrace is designed to allow additional providers. For example, an Apache provider provides instrumentation of the web server. This ability to instrument at the application level, in addition to the operating system level, provides a path for expansion of DepMap’s capabilities in monitoring software in specific contexts.

I also use common Unix tools as information sources. Where DTrace provides specific event information associated with a program (e.g. executable “A” invoked the read system call in thread #1234) or system wide (e.g. time spent in calls to sync), tools such as iostat and vmstat provide averaged information for system wide behavior with low resource impact. These are used to establish baselines, trends, and to understand when specific types of dependency monitoring may be called for (e.g. disk I/O operations when iostat reports high %iowait time). This will be discussed further in Section 3, Design.
3.3. Requirements and Use Cases

Software that a typical programmer believes to be thoroughly tested has often had only about 55 to 60% of its logic paths executed. ...it is nearly impossible to test software at the level of 100% of its logic paths (Glass, 2003).

The broad requirements for DepMap were introduced in the previous section. I will now consider each of these in more detail in the context of operating system events and relevance to applications. I then describe the intended use cases for the initial implementation of DepMap. For the purposes of the current work, I focus on mapping dependencies from the perspective of a single host, but with a view to future functionality to aggregate results from many hosts into a larger map.

3.3.1. Establish a detailed inventory of actors and the connections between them

In the context of applications running on an Unix-like operating system, I define actors to be processes and threads. All applications are made up of one or more processes, and it is through the execution of these processes by the operating system that the application performs functions and creates connections. A process can also be considered as a resource consumer; file and socket descriptors as well as memory are allocated by process. Each process may have one or more associated threads, which are sequences of execution that share the resources of the enclosing process. A thread cannot exist without an associated process and cannot itself “own” resources outside of this process. Thus each process is an actor, and we may consider the individual threads within it as responsible for some of its actions. Note that on a distributed system, processes on two separate hosts are considered actors within the same system.

Processes are related through parent-child relationship; each process has a parent, with the “init” process being the ultimate parent of all processes in the typical Unix model. Processes may inherit some resource (and thus some connections) from their parents. For example, a child process inherits a copy of its parent’s file descriptor table (Stevens, 2005).

A simplified diagram of the Unix process model is shown below:
In this diagram, the process of interest has a process ID or PID of 1234. It has a parent process of 1000 (the parent process ID or PPID). It has children with process IDs 2000 and 2001. The parent process ID of a given process may change over time, such as occurs when the original parent exits. Thus it is most helpful to know the original PPID of a process to know how the process came into being, and DepMap should track initial values as well as changes in PPID. This diagram shows two threads with PID #1234, denoted t07 and t08, which have access to the process data and open files.

Open files represents access to many possible resources beyond simple files on disk, including network sockets and pipes, which are indexed by file descriptors. Additional methods of sharing information, such as shared memory and semaphores, have conceptually similar data structures that map an identifier to a resource. Setup and teardown of these resources occurs via system calls, and in some cases access to the resources is also via system call.

The connections between the actors include shared files and any form of inter-process communication, such as sockets and pipes. Note that this also includes so-called “connectionless” or datagram-based communication. While unlike stream-mode sockets there is no concept of a long-term maintenance of a communication path (or reliable delivery), there is clearly some potential for a one actor to affect another. The creation (and in many cases use) of these connections is mediated by the operating system based on requests by processes. These requests occur via system calls.
For example, access to a file on disk is initiated through a call to `open`. The contents are accessed using `read` and `write`. Access ends with a call to `close`. In the case of shared memory, access may be set up with `shmget` and `shmat`, but once this occurs communication happens through modifications of memory in the address space of the involved processes. Given this, DepMap will track all setup and teardown of access to connections, but will have visibility into the use of only a subset of connection types.

Thus DepMap is required to monitor the existence of processes and the parent-child relations between them. It is further required to monitor the subset of system calls that are typically associated with communication between processes, access to files, or communication to other hosts.

3.3.2. *Link together actors based on these connections to build a high level map*

DepMap must note the setup and teardown of connections across all processes, and track identifiers that will have significance across processes and time. For example, knowing that process #1234 opened and read from file descriptor #5 is not useful, but knowing that file descriptor #5 corresponds to a file “/tmp/startup.txt” and that this file was written to by process #2001 is useful. It allows us to not only understand that both depend on the file, but also that there may be a flow of data from #2001 to #1234. This also illustrates it is desirable monitor control information for connection (e.g. read only, write only, or read/write). Because this is from the perspective of the process requesting the action (initiating a system call), features such as chroot and file links create the potential for aliasing of file-based resources, while DNS names and IPv4 versus IPv6 addresses create the potential for aliases of network-based resources. DepMap can mitigate some of these by tracking calls that create potential for aliasing, flagging associated identifiers, and creating equivalency maps. This is imperfect, however, as it cannot emulate every identifier mapping behavior of the system.

3.3.3. *Describe both connections and actors in detail*

Even small programs may have a large number of dependencies, and so it is important to collect sufficient detail on each to allow for meaningful aggregation and presentation. DepMap is intended to be used for two primary use cases: initially mapping a system, and from there drilling in to detail about specific actors and connections. For the first use, we want to see less detail but have options to show only the most meaningful connections in certain contexts (e.g. network connections initiated from a process to a remote host, or
heavily-accessed local files). For the second, we want great detail for a small subset of the connections in a system.

Together, these require that DepMap collect substantial detail for relevant system calls, including time of occurrence (to allow for ordering), duration of call (as a measure of the impact of the call on system progress and to gauge variation in performance of external resources), user ID performing the call, and sizing information where relevant (e.g. number of bytes retrieved per read operation). Monitoring return codes is also necessary, as they provide an indicator of success/failure, as well as sizing and resource identifiers (depending on the system call).

It is also necessary to measure overall host processor status metrics to provide context for specific actors and connections. For example, when a host is spending all of its time on IO wait activity, details about which processes are performing IO and possible causes of contention become more important and should be weighted more heavily.

Collectively, this provides estimated measures of importance of a connection to the correct operation of the system in terms of performance. DepMap must further allow for targeted injection of errors or increased latency. This can be used to isolate and test fixes for known issues (as in assessing the behavior of a service if database response times were doubled). It can also be used to proactively identify issues through automated "gremlin injection" in which test scenarios are repeated with artificial delays or failures caused in various parts of the system. The results of this sort of test provide very high confidence measures of the level of importance of a connection to the overall system.

In some cases it will be necessary to collect full traffic information from a connection. Given the volume of data it is not practical to collect all traffic for all connections, however the capability should exist to manually configure this collection for a subset of connections.

3.3.4. Provide data analysis and visualization methods

DepMap must provide analysis tools that allow for automated analysis of system call event data. Graphical output for this will consist of dependency graphs with actors as vertexes, connections as edges, and labels and visual cues used to describe the weight and likely relevance of each. Further, DepMap must support creation and analysis of time-series observations. Output for this will include scatter charts, line charts, and histograms and the ability to apply aggregation, filtering, and smoothing for selected outputs.
As described above, the two primary modes of operation are generation of diagrams showing actors and connections between them, and detailed views of the performance of connections both individually and in aggregate. The audience for these results is developers and system administrators, and the output should be immediately understandable by someone with the relevant technical background.

A critical feature for analysis is aggregation of information based on contextual values. For example, while standard tools like `iostat` aggregate over time (e.g. ten second intervals), it is often more useful to aggregate over file name ("which files see the heaviest activity when handling this type of request?"), logical block ("which part of this file is most heavily utilized, and perhaps should be moved to faster storage?"), request size ("are we performing a large number of small reads?") and latency ("what is the distribution of the time spent waiting on responses from this network host?"). In common to all of these is the need to aggregate on data relevant to the question at hand, and to avoid averages that may conceal micro-patterns in system behavior, or occasional high impact events.

In addition to user-readable outputs, DepMap must provide both dependency and time series data in formats that can be readily consumed by other analysis, reporting, and alerting tools. This will include tab delimited, XML, and .dot (a graph description format) (AT&T Research and Graphviz.org Team).

3.3.5. Moderate and tunable performance impact

All of the above must be accomplished with at most a moderate impact to performance of the system, excepting the case of intentionally injected delays or failures. There is an expectation that the impact will increase as the number of system calls increases; the user will be able to tune monitoring to an appropriate balance between impact on the running system and collection of detailed data. For example, if monitoring all calls to `read` is causing unacceptable impact, DepMap can be switched to a sampling mode in which approximately one out of N reads is monitored, including disabling monitoring of some calls entirely. Generally, DepMap will allow for detailed dependency identification (high overhead), sampling identification/monitoring (low overhead), and failure injection (intentional impact).
3.3.6. Supported platforms

DepMap will run on Unix-like systems that support DTrace and the standard diagnostic tools iostat and netstat. Initially, this will include MacOS (Darwin), Solaris, and Linux (with restrictions due to level of DTrace support level on this platform).
3.4. Use Cases for Evaluation

You think that’s because you understand “one” you must therefore understand “two” because one and one make two. But you forget that you must also understand “and.” –Sufi teaching story, as related in (Meadows, 2008)

Section 5, Results and Analysis, describes the use of DepMap tools on two complex systems in use at Cal Poly. In order to illustrate how the above requirements are realized from the perspective of a system administrator or developer, I will briefly describe a number of use cases where DepMap should provide actionable information.

Scenario 1, Distributed Open-Source Web Application: A Java web application comprised of tens of thousands of lines of locally developed code, running on a set of open-source frameworks. Third party code is deployed within some of these frameworks. Collectively, this system consists of hundreds of thousands of lines of code hosted across multiple servers. All source code is available, but some network dependencies remain unclear due to dynamically introduced connection requirements. Example problem to solve: all servers become unresponsive occasionally. Does not appear to correlate with time of day, resources available on the local hosts, or a specific usage pattern. The goals in this scenario are to identify external network service dependencies and understand variation in performance of these dependencies.

Scenario 2, Batch Process in Database Server (details in Section 5, Results and Analysis): A data warehouse pulls in hundreds of gigabytes of data on a daily basis. Data can be pulled only after processing has completed in the source databases, and so the start time for this operation is constrained. This data then undergoes transformations to produce a variety of outputs, which must all be completed well before the start of a normal business day, so that consumers of the data can meet their operational needs when they begin providing service to their customers. This occurs in a production environment, as well as development and validation environments. Run times in all three environments are growing worse, to the point that production service targets are sometimes not met (nightly batch processing requiring over twelve hours), and development environment is run sparingly as processing can take over eighteen hours and cause impact to performance runs. Trial and error approach (“easy fixes”) has been exhausted. The goals in this scenario are to characterize workload, quantify performance factors contributing to overall slow performance, and provide vendor-neutral assessment of cause and effect for this issue.
**Scenario 3, Vendor Application Failures:** A client/server application for which no source code is available is failing to complete a critical operation following migration to a new server and operating system release. The displayed error message is not helpful, no further diagnostic information is available, and documentation and vendor support are limited. The goal in this scenario is to identify operations attempted by application leading up to the failure and likely causes of the failure.

In all cases, while various levels of information may be available to the administrator in the form of source code, configuration files, and documentation, DepMap will perform *black box* assessment. No special knowledge of the target under analysis will be required, other than initial hints about which user ID, executable name, or process IDs should be targeted.

Other scenarios may be supported by DepMap through analysis or transformation using other tools, including creation of firewall “accept” rules based on observed traffic for known-good activity, creation of SELinux rules, and creation of automated test plans based on observed actual customer activity.
3.5. Challenges

*I have yet to see any problem, however complicated, which, when looked at in the right way, did not become still more complicated. Poul Anderson, as related in (Meadows, 2008)*

Based on the requirements for DepMap, some challenges become clear prior to design, implementation and testing. I identified these as I began my work in order to ensure they were mitigated as early as possible.

The foremost challenge is the large volume of data that can be collected. Monitoring application behaviors at a low level can easily lead to consumption of more resources in monitoring than in application processing. The method I have chosen is inherently data-centric, and thus it is important to collect only that data which will reasonably be useful, and to filter as much as possible within the context of how the data is being used. In practice, this motivates providing effective sampling to generate a high level map, and then moving to a focused mode that consider specific workloads (by user, process, network address range).

Beyond collection, information can also be overwhelming in presentation to the user (developer or system administrator). As described previously, it is critical to aggregate on meaningful identifiers, and to find the right tradeoff each time DepMap is used. At the other extreme, there is a risk that over aggregation, under sampling, or simplification that would leave DepMap providing no additional detail beyond that available from existing tools while consuming more resources. Balancing between these extremes will require tunable options, with useful settings varying for each usage scenario for DepMap.

A core tradeoff in this area is avoiding examination of connection traffic during normal operation. While DepMap must provide mechanisms for examining traffic when necessary, its focus must be more on identification and characterization rather than detailed analysis (e.g. of specific protocols like HTTP or file formats like .xml).

Establishing the relative importance of a connection within a system is related to the fundamental problem of determining causality versus correlation. In collecting performance data, DepMap will establish a variety of correlation indicators. Consider the case of a web application that is showing high response times during “busy” parts of the day (when more users are active). At a high level, graphs from DepMap could be interpreted to mean that user demand is causing slow response times. Only on further examination of
latency on specific network connections does it become clear that a heat-related issue is causing a network switch to fail during the warmer hours of the day. Generally, my goal is to provide clear information and avoid suggesting causality, except in cases of synthetic workload with controlled variables during automated fault injection.

Key to DepMap providing relevant information about connections between actors is that the connection identifiers used by each actor can be reconciled. For many types of connections, a distinct and consistent identifier is not be guaranteed within the context of the system call API. The identifier that is meaningful to the administrator especially may not be consistent. For example: when a process opens a file, it specifies a path name. Another process may access the same file using an alternate name via a file system hard link. In this case, the filename is a meaningful identifier, but the file’s inode is the consistent and distinct (within that file system) identifier. This becomes especially problematic across hosts, and in cases where translational layers (such as IPv4 to IPv6 mapping) are used. DepMap should use the best possible identifiers for each resource (most likely to be consistent and distinct), and the best possible identifier for display to the user (most likely to be meaningful over time).

Test coverage is a recognized problem in measuring software quality. In order to fully test software, all possible logic flows must be exercised and evaluated for correct performance. “The test(ing) system tends to be more complex than the system being tested; it has to allow both the ‘normal’ and ‘abnormal’ processes,’ or is the system itself should only consist of the ‘normal’ processes” (Armour, 2003). As DepMap builds an understanding of software through observation of execution, it faces the same challenge. If a dependency is never exercised during observation, it cannot be seen. DepMap addresses this challenge in two ways: first, by allowing observation of systems under real-world load. The overhead of running DepMap is intended to be low enough that it can be used on actual production systems. While this does not guarantee full coverage, it provides coverage of the cases that are encountered during normal operation, and can be run for a sustained period of time to detect cases that are intermittent or rare. Second, DepMap allows injection of flaws and by doing so is able to observe the target system under states that are not commonly encountered.

The definition of system calls in Unix-like systems is stable and is defined by POSIX standards. Even systems which are not formally certified as POSIX compliant (including
Linux, OS X/Darwin, and FreeBSD) provide generally consistent data structures and system call signatures. For example, the `send` and `recv` system calls are used to send and receive messages on sockets. Both take as arguments a socket descriptor, a data buffer, a length, and flags (Stevens, 2005). While the developer-facing interface is stable and fairly consistent, the implementation is not. In fact many defined calls are not actually implemented as distinct system calls. For example, `send/recv` may be called on both OS X and Linux but are implemented as user-land wrappers around other underlying system calls rather than as discrete system calls. Given this, DepMap must vary in its treatment of system calls on supported platforms. The `<syscall.h>` header files and list of instrumented calls from the `dtrace syscall` provider will be considered authoritative for each platform.

An alternate approach to using `syscall` for all monitoring would involve using additional DTrace providers, such as `io`, to monitor some events. This would provide for a more simple DTrace script, but would not be as easily portable to each platform and `io` and other available providers are not consistently implemented across all of the operating systems where DTrace can be used.

As DepMap has a requirement of causing minimal impact to a running system, and in turn relies upon DTrace which has a similar design criteria, it is possible that some information could be dropped when a system is under heavy load. This could occur for example due to exhaustion of DTrace buffers in kernel space or heavy CPU load. DTrace monitors its own performance, and will gracefully degrade monitoring when certain resource usage or impact thresholds are crossed. In these cases it reports that this has occurred; DepMap will recognize this and report it as a limitation.

Most server-class systems today have multiple processing cores, which means there is potential for synchronization issues when processes are being monitored on two or more processors. DTrace uses separate buffers for each processor, and does not guarantee that results will be output in chronological sequential order. As processes are regularly moved between cores, this could causes situations where, for example, a `read` against a file descriptor may be reported prior to the reporting that an `open` of the file occurred. DepMap can address this challenge by tracking event times, and reordering events after they are observed to restore ordering.

Monitoring system calls requires root privileges. On some platforms, targeted privileges (least privilege) can be granted to a user to allow only those capabilities that are necessary.
As DepMap will monitor a wide range of system calls, it effectively requires full read access, but not write access. On systems where this level of access is not available, DepMap may be used only in aggregate (time sampling) monitoring modes, in which it uses only standard Unix tools that are typically granted to normal users. For example, on Linux the `iostat` command runs without root privileges and collects information from `/proc/diskstats`, which is normally world-readable.

DepMap provides limited facilities for collecting the content transmitted via connections. With the increasing use of encryption for network traffic, and much of that encryption implemented in user-mode code (e.g. OpenSSL), plaintext network transmissions will not be available at the system call level. In cases where this use is important, DepMap may be extended via application DTrace providers to obtain plaintext content. Obviously this information would need to be protected.

While DepMap is intended to have only moderate performance impact, due to its primary mode of operation there are two situations in which it may have relatively higher impact. First, when monitoring processes that make frequent, short-duration system calls. For example a process that performs a large number of small read operations against a file. In this case the consistent per-call overhead of monitoring will make up a disproportionately large portion of system CPU time than on a system where there are fewer system calls and each takes longer to complete. Second, by running on all of the same processors that run application code, DepMap will have some impact to processor caches; in cases where the application is especially sensitive to processor caching, each system call has the potential to impact non-system call behavior in the form of higher cache misses for the application. I have measured the overhead of DepMap on representative workloads to understand this impact (see Section 5, Results and Analysis).
3.6. Architecture and Detailed Design

*All scientific constructs are models representing certain aspects or perspectives of reality (von Bertalanffy, 1969).*

The high-level design of DepMap is shown below.

![Figure 8: High Level Design of DepMap](image)

The core functionality of DepMap is in the Generator, Collector, Analyzer, and Reporter components. These are typically run as depicted, on a separate host from those under observation to minimize impact to monitored targets. It is possible to run all of these on a target host if required and all software dependencies are met (these are described in more detail below).

The Generator creates the appropriate DTrace and system scripts for the platform(s) being targeted for observation. This includes substituting in necessary predicates, system call variations and interpretations in the .d scripts, and invocation mechanisms for tools like *iostat*. This also includes any workarounds necessary for limitations of that platform (e.g. splitting scripts apart to fit within buffer limitations). The output from Generator is a set of standalone scripts that are copied to the target host(s) and can run there independently from the DepMap host.
After these scripts are invoked, manually or via init or cron trigger, they collect information to local files in a compact, machine-readable format. In cases where collected information is expected to be large (e.g. weeks of observation on a busy system) and extra processor time is available the outputs may be optionally sent through gzip compression to save space on disk. Note that outputs are ideally sent to a local storage device not used by the application being targeted, in order to minimize the impact of DepMap.

The Collector pulls information from monitored hosts, checks for integrity and host identifying information, determines the time period covered, and places it in a data store on the DepMap host. This step can be automated via pulls over ssh, or performed manually as desired by the administrator. Automated triggers may be defined for ending collection (e.g. when an event of interest has occurred).

The Analyzer performs two main types of analysis: reviewing event-based information from the DTrace scripts to identify dependencies, and processing time series data based on events or on the outputs of the system status scripts (e.g. iostat). For dependency identification, the result is a user-readable file describing the identified actors (sets of processes, each potentially composed of threads) and the connections between them (accessed files, network connections, shared memory, etc.) Three types of descriptive information are provided for each connection:

1. Directionality: in the case of a file, if the file is read only than the directionality is from the file to the process, in the case of write only it is from the process to the file. In the case of a network connection, this signifies whether a socket is primarily listening or initiating (for example, :80 in a web server will be primarily listening and thus towards the process), though data flows both directions. In the case of shared memory, this is based on the shmflag argument (read only, or read-write).

2. Descriptions and activity summary relevant to that connection type: (e.g. “TCP 127.0.0.1:80” or “/tmp/log”, and “IN 10/512KB, OUT 10/10KB” meaning ten reads constituting 512 kilobytes of data, and 10 writes for 10KB of data). Options allow some of this information to be suppressed or further aggregated (e.g. show only total data transferred).

3. Calculated weights based on activity observed over that connection: these represent the relative proportion of activity occurring over the connection
compared to other connections of the same type, or to all connections in the system (configurable). For example, if one network connection accounts for 90% of the network traffic observed from a host, its weight will be 90% of the total connection weight for network traffic. This is used by the Reporter to sort or visually emphasize high or low-traffic connections.

The Reporter consumes the data produced by the Analyzer to present the user with maps of runtime dependencies present in the system, and with graphical representations of the time series data grouped by dependency type or performance level. For time series data, Reporter must support:

1. Scatter plots by time, latency, or resource.
2. Line charts, with stacking by time period (e.g. activity each hour or each day) or by separate connections (e.g. level of activity or latency on one connection versus another)
3. Cumulative activity over time for comparing overall levels between connections or systems.
4. Smoothing of data to find common patterns.

These outputs will be discussed in more detail in Sections 4 and 5 (Implementation and Results and Analysis), and full page reproductions are provided in the appendices. Below are examples of a simple dependency map, and two representations of an aggregated (over related storage device) connection from the system:
Each component of DepMap is tunable two ways: via command line arguments, or via configuration files. Command line arguments are used to enable or disable functionality, such as targeted of specific processes or executable names, while configuration files are used to specify mappings (hints) for how DepMap should group or interpret data.
As information collection is one of the central features and challenges for DepMap, let us consider the design of this further. DepMap relies on two types of information source: system call event data generated by DTrace scripts, and time series data generated by standard Unix programs such as iostat. Note that while other executables are used on some platforms to obtain additional information, such as mpstat on Linux for information on the status of individual CPUs, for the remainder of this description I will focus on iostat as a representative example.

For details about the architecture of DTrace, please refer to (Sun Microsystems, 2008) or the many other excellent references available. Briefly, DTrace consists of kernel-mode code that allows “.d” scripts to be compiled into an intermediate form and run in a sandboxed environment within the kernel to collect, analyze, and output information. The syntax of .d scripts is similar to .c, though most types of conditionals and iteration are not supported in order to guarantee safety. DTrace scripts are composed of a series of probe clauses, each of which describes some information collection/analysis/output activity. From (Sun Microsystems, 2008), these probe clauses are of the form:

```
probe descriptions
/ predicate /
{
    action statements
}
```

A probe description specifies the probes on which to trigger, and may use wildcards. The description may include a provider, module, function and name. The predicate allows us to select specific cases in which actions should be taken (for example, only for executables with a certain name). The action statements can include access to arguments made available by providers, storing values in variables, introducing delays or modifying provider variables ("destructive actions"), and output of information. For a specific example, consider the code necessary to monitor all returns from the open and shm_open system call made by executables named "java":

```
syscall::open:return,
syscall::shm_open:return
/execename == "java"/
{
    printf("%s returned %d\n", probefunc, arg0);
}
```
Here we see a probe that utilizes the \textit{syscall} provider. Note that no \textit{module} is specified (syscall::). We execute the actions only if the \textit{execname} (a built-in variable) is equal to “java”. When \textit{open} or \textit{shm\_open} returns from calls by a java process, we output the function name (\textit{probefunc} is another built-in variable in .d), as well as the return value of the system call (\textit{arg0} is defined to be the return value for a system call). Note that as the call is returning, we do not have access to the original arguments to \textit{open} in the \texttt{:return} probe.

Figure 12 shows how the DepMap .d script(s) interact with the DTrace facilities, and the high level architecture of DTrace, as adapted from (Sun Microsystems, 2008):

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig12.png}
\caption{Architecture and Components for DTrace Usage}
\end{figure}

The model for collecting \texttt{iostat} and similar data is according to normal Unix command line arguments and pipes. For example, on Linux highly detailed device I/O information is collected at ten second intervals and saved to a compressed file via:

\begin{verbatim}
iostat –ktx 10 | gzip –c > dmap\_io\_fedorahost.dat.gz
\end{verbatim}

The overall architecture for this model relies on periodic reads from the \texttt{/proc} filesystem, or equivalent APIs on Solaris and MacOS that provide read access to kernel data structures, as shown in Figure 13.
Figure 13: Architecture and Components for iostat Usage
4. Implementation

*I can call spirits from the vasty deep.*

*Why so can I, or so can any man; but will they come when you do call for them?*

*Shakespeare, King Henry IV, Part 1*

Implementation proceeded in six phases, consisting of setup of development platforms, prototyping each of the primary components (Generator, Collector, Analyzer, Reporter), and then integrated development and testing of the components across the platforms.

4.1. Data Collection

The first step was to verify correct operation of DTrace and sysstat tools on each platform. Details on the operating system and hardware platforms utilized are given in Section 5, *Results and Analysis*.

4.2. Sysstat Collection

While sysstat-like tools vary between platforms, they are installed as part of the full default install on each of the three target platforms (Solaris, Linux, and OS X) and on each are well-documented and stable in operation. The effort in this area was to establish the minimum data that would be collected on all platforms, translate it from each into a common format, and provide extensions for handling the additional data available on some platforms.

The lowest common denominator from these implementations is sufficient for the purposes of DepMap, as in most scenarios DTrace-based information can provide the detail not offered by some implementations. On platforms that have a rich version of iostat, for some device-based operations DepMap can rely on iostat alone for information down to the *device at one second interval* level of granularity. *CAMP* (a “Common API for Measuring Performance”) provides access to a limited amount of disk activity information. *(Gabel & Haungs, 2007)* Much of the translation work I performed here could be used to expose more granular I/O information to CAMP, such that it could be used in other efforts. I discuss this further in Section 6, Future Work.
OS X uses an older version of iostat from FreeBSD. This version provides modern space units (MB or KB, in addition to blocks) but is otherwise very simple. Sample invocation on OS X is as follows:

```
$ iostat -n 10 -w 10
```

```
Disk: cpu load average
KB/t tps MB/s us sy id 1m 5m 15m
17.01 2 0.03 5 4 92 0.39 0.26 0.19
542.67 0 0.16 3 4 93 0.41 0.27 0.20
```

Solaris uses a Sun version of iostat that has more features than the BSD version, but less than the Linux sysstat package. Noteworthy is the “-x” option, which outputs service time and queue length data, the “-z” option that suppresses lines of all zeros (reducing the space required to store long term observations from systems with many idle devices). Note that the odd formatting of the cpu numbers is due to the “-z” option:

```
$ iostat -c -x -z
```

```
Device extended device statistics
device r/s w/s kr/s kw/s wait actv svc_t %w %b us sy wt id
0 0 100
sd0 0.0 0.0 0.0 0.0 0.0 0.0 42.8 0 0
sd1 0.0 0.0 0.8 0.1 0.0 0.0 5.8 0 0
```

Most Linux distributions use iostat via the sysstat package maintained by Sebastien Godard (Godard, 2010). This is a feature-rich iostat, and is under active development. Though new outputs continue to be added to this version, common versions on Linux distributions range from 5.x to 9.x. All of these support display of extended information, which is valuable for assessing performance issues. Timestamp and machine identifiers are suppressed, and lines are rewrapped for clarity:

```
$ iostat -ktx sda
```

```
avg-cpu:
%user %nice %system %iowait %steal %idle
0.01 0.00 0.03 0.16 0.00 99.81
Device:
rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgq-sz
sda 0.00 0.10 0.02 0.05 0.26 0.61 24.58 0.00
await svctm %util
51.50 22.37 0.16
```

4.3. DTrace Setup and Collection

DTrace is present in a default installation of both Solaris and OS X (or at least any with basic developer tools included), but is not included in any mainstream Linux distributions.
At present the Linux port of DTrace created by Paul Fox is considered to be incomplete, and as such I compiled the latest released version (dtrace-20091010) from source and used a version of the Linux kernel that seemed well supported by the release. It is made active by loading a kernel module as shown below:

```
# make load
tools/load.pl
15:05:20 Syncing...
15:05:20 Loading: build/driver/dtracedrv.ko
15:05:22 Preparing symbols...
15:05:22 Probes available: 186331
15:05:25 Time: 5s

# lsmod | grep dtrace
dtracedrv             367064  0
```

As shown in Table 1, I validated dtrace on each platform by using it to query the total number of probes instrumented (across all providers, modules, functions, and names), the total number of functions instrumented via the syscall provider, and a basic test of monitoring the open system call as I illustrated in the Section 3, Design. I also verified correct operation of the C Preprocessor, cpp, from within DTrace scripts as I use this for macros.

<table>
<thead>
<tr>
<th>Platform</th>
<th>System Calls</th>
<th>Total Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac OS X / Darwin</td>
<td>430</td>
<td>68622</td>
</tr>
<tr>
<td>Linux</td>
<td>329</td>
<td>33637 (plus 152694 instr probes)</td>
</tr>
<tr>
<td>Solaris</td>
<td>236</td>
<td>60848</td>
</tr>
</tbody>
</table>

*Determined using commands:*

```
dtrace -l -P syscall |egrep -v "NAME$|entry$" | wc -l
```

```
dtrace -l |egrep -v "NAME$" | wc -l
```

**Table 1: DTrace Probes by Platform**

Examining the full outputs, I noted the following platform variations: Mac OS X includes a number of non-standard system calls, as well as nocancel variants of some calls (to prevent interruption mid-call, e.g. fsync_nocancel ), Linux instruments a very large number of functions via the instr (instruction) probe, which is not relevant for DepMap work, and that all platforms have some form of 64 bit specific versions of a subset of system calls (e.g. stat64 ) though in many cases the non-64 bit versions are now wrappers on 64 bit implementations. I also noted that providers other than syscall implement some
functionality relevant to DepMap (e.g. the proc and io providers), but that these are not consistent between platforms. I discuss these further in Section 6, Future Work.

To arrive at the appropriate system calls to monitor on each platform, I used three sources of information:

1. Presence in dtrace output: to generate a list of system call entry and return probes that are instrumented. I also confirmed this list against <syscall.h> to ensure no system calls were missing probes.
2. The prototype present in system include files such as <unistd.h> to determine the argument type signatures of each system call. For example, on Linux `lseek` has the signature (int __fd, __off64_t __offset, int __whence).
3. The system manual pages, to generate comment blocks to aid me in implementation of appropriate dtrace probes for each relevant system call.

I developed a Perl script that iterated over the list of all system calls (source 1), and for each call obtained the function signature (source 2). Using a hash data type, I grouped together all calls with identical signatures, as calls with equivalent signatures can generally be handled by the same probe block, minimizing redundancy in the .d script. For each grouped block of calls, I then added a comment block from the man pages showing the developer-readable explanation of the function arguments as a development aid. I then reviewed the results on each platform, and on each built a "do not include" list of system calls that are not relevant to the current requirements of DepMap, and collapsed some arguments that I intended to ignore.

I determined that for each system call I would have a common filter criteria and at minimum save the current system time and the arguments on entry to the call, and then output relevant time, duration, argument and return value information as the call returned. The result of this work was an auto-generated skeleton .d script on each platform including all system calls monitored via DepMap. An example for open and similar system calls is shown below, including relevant preprocessor macros to always avoid monitoring our own dtrace process, and to monitor "java" executables:

```c
#define FILTER (pid != $pid && execname == "java")
/* on entry into a system call, we must set aside any arguments that we
 * wish to have available on return; at return arg0 and arg1 are set to
 * the return value of the system call and the rest *may* be overwritten
 */
...
#define SAVEARGS_1 self->arg0=arg0; self->ts=timestamp
#define CLEARARGS_1 self->ts=self->arg0==0
```
Note that lines following the /* ADD CODE HERE */ comment are what was necessary to manually code to handle this set of system calls. To explain further how this works: upon entry into the open-like system calls, I check the built-in variable pid (the process ID of the current process) against the PID of the dtrace program. This is to ensure DepMap never triggers on its own monitoring code (which, while theoretically a safe operation, would skew results). The full DepMap includes other checks, left out here for clarity, to further prevent monitoring other DepMap code such as invocations of iostat. In this case, DepMap has be configured to monitor only “java” executables, using the execname built-in variable.

Next, I save the two call arguments provided by the syscall provider as arg0 and arg1 in raw form (int64_t) into the thread-local variables self->arg0 and self->arg1. I also save the current system time provided by the DTrace built-in variable timestamp to the thread-local variable self->ts. I have now saved all relevant state information for this call, the probe is complete, and control is returned to the kernel, which invokes the system call as it normally would.
When the system call completes, in this case associating a file descriptor based on the path or name and flags provided, the exact same filter is applied. Thus, we are certain that if we enter the \textit{return} probe we have already saved thread-local variables self->arg0, self->arg1 and self->ts. I then invoke the PRINT\_COMMON macro, which outputs the current time (at which the system call completed), the delta (approximate time spent in the system call, recognizing that DepMap may have impact this somewhat), identifiers for the parent process, current process, and current thread, the executable name, and the name of the system call. Up to here, all system calls are handled fairly consistently, with some variations and additional macros invoked for those system calls that use existing file or socket descriptors.

The next two lines are manually coded for these system calls. I use the Dtrace \textit{copyinstr} function to treat my saved copy of the first argument (former arg0) to \textit{open} as a pointer to a null-terminated string, and copy this string into my own clause-local variable this->path. In the case of \textit{open}, this will be the pathname of the file that has just been opened. I then output the return value, the current arg0. This is a file descriptor identifier, if the call was a success, or -1 if it was a failure. The output the path we attempted to open, and the flags associated with this attempt. Together, this provides sufficient information for the Analyzer to determine when this call took place, how long it took to complete, whether it succeeded, what sort of use is intended for this file (based on the flags), and that a given file descriptor is now associated with this path.

Next, the return value is printed. Though in this case we have already printed this as a file descriptor, in all DepMap probes the return value for the call is printed as the last field on the line, prefixed by ":=". I invoke the CLEARARGS\_2 macro, which zeroes out all of the thread-local variables I created, freeing the memory they consumed. From here, flow proceeds as normal and the system returns to the user code that triggered the system call. This completes the normal lifecycle of DepMap dtrace-based monitoring of a system call.

While the \textit{open} call is relatively simple, monitoring of other relevant calls is similar if more complex. I will describe two additional examples in more detail: \textit{close}, and the \textit{accept/connect/bind} calls. The complexity with \textit{close} is that we would like to output information based on the file descriptor, but data associated with that file descriptor will be destroyed by the time the \textit{syscall::close:return} probe is triggered. In this case, I save more information up front in the \textit{:entry} probe before it is destroyed. Note that some of this could
have been rebuilt later by the Analyzer based on the *open* call associated with this descriptor; I chose to output potentially redundant information to verify that information about the descriptor was not changed while the file was open. This had the added benefit of highlighting that the *fds[[]* built-in variable will not have full pathnames (matching that used on *open*) for all file types on all platforms.

```c
syscall::close:entry,
syscall::close_nocancel:entry
/FILTER/
{
    ...
    this->pathname = fds[self->arg0].fi_pathname;
    this->fi_oflags = fds[self->arg0].fi_oflags; */
}
```

As a more complex example, in monitoring the *accept/connect/bind* calls I dealt with the most common families of sockets as independent cases so that I could unpack the fields of the specific *sockaddr_* types. This was necessary in order to display Unix local, IPv4 and IPv6 socket addresses correctly. In the example below, I have removed some lines for clarity including references to the *accept* and *bind* names.

```c
syscall::connect:return,
/FILTER && self->arg1/
{
    ...
    this->sock = (struct sockaddr*)copyin(self->arg1, sizeof(struct sockaddr));
    self->sa_family = this->sock->sa_family;
    ...
} syscall::connect:return,
/FILTER && self->sa_family==AF_UNIX/
{
    ...
    this->sock_un = (struct sockaddr_un *)copyin(self->arg1, sizeof(struct sockaddr_un));
    printf("%s\t\t", this->sock_un->sun_path);
    ...
} syscall::connect:return,
/FILTER && self->sa_family==AF_INET/
{
    ...
    this->sock_in = (struct sockaddr_in *)copyin(self->arg1, sizeof(sockaddr_in));
    this->addr_string= inet_ntop(self->sa_family,&(this->sock_in)->sin_addr);
    ...
} syscall::connect:return,
/FILTER && self->sa_family==AF_INET6/
{
    ...
    this->sock_in6 = (struct sockaddr_in6 *)copyin(self->arg1, sizeof(sockaddr_in6));
    this->addr_string = inet_ntop(self->sa_family,&(this->sock_in6)->sin6_addr);
    ...
} syscall::connect:return,
```
/FILTER & & { 
    self->sa_family != AF_UNIX 
    & & self->sa_family != AF_INET 
    & & self->sa_family != AF_INET6 
  }/

{ 
  ... 
  printf("\tUNSUPPORTED FAMILY %d\t", self->sa_family);
  ... 
}

Note that I have not only saved the inbound arguments on :entry, but also unpacked one to determine the sockaddr family associated with this call for evaluation in subsequent predicates. While DTrace does not allow if/then/else constructs, it does guarantee that probe blocks are evaluated in the order of appearance. Conditionals can then be expressed by saving values and then referring to them in predicates of later probe statements.

Also note that, where possible, I use DTrace and system-provided functions to handle network versus system byte order differences (e.g. inet_ntop). Separately, where I do not explicitly handle a case, such as socket types other than _UNIX, _INET, and INET6, I use a catch-all to output generic information. This information will in some cases be sufficient for the needs of DepMap, and where it is not it will draw attention to the case such that it can be considered for additional handling.

When DepMap runs in flaw injection mode (with the "-x" command line argument) I use the chill() DTrace function within the syscall:::return probes to introduce additional delays. Even with these sorts of destructive actions explicitly enabled, DTrace enforces safety thresholds to limit the potential for causing the system to hang. With destructive actions enabled it is also possible to modify user memory and thus introduce flaws into system call arguments or results. Data can be subtly changed, or broadly corrupted. I did not implement this form of fault injection, but discuss it further in Future Work.

To illustrate the use of latency injection, consider a simple case of causing all calls to open to be artificially delayed. To begin with, the results of a trace showing the entry and exit time for a call to open64 on "/dev/null". Note that the time change between these is 49946, or 49 usec.

```
> 94836597987992       4608    4692    1       cat     open64  /dev/null
< 948365979837938       4608    4692    1       cat     open64
```

Next, attempting to introduce latency without DTrace destructive actions enabled (no calls to chill will be allowed):
syscall::open*:entry
{
    printf("%s\t%ld\t%ld\t%ld\t%ld\t%ld\n", \n    timestamp, ppid, pid, tid, execname, probefunc, copyinstr(arg0));
    chill(1000000000);
}

# ./open.d
dtrace: could not enable tracing: Destructive actions not allowed

Now, with destructive actions enabled but exceeding the allowable threshold for delays per probe:

# ./open.d
dtrace: error on enabled probe ID 2 (ID 185683: syscall::open:entry): illegal operation in action #8
< 1273444502858029687 18928 18929 18929 dtrace open

Again, this time with chill (100000), or 100 usec, resulting in a 245 usec total duration of the call. Note that some variation exists from call to call, irrespective of introduced delays.

> 948439586167422 4608 4709 1 cat open64 /dev/null
< 948439586412393 4608 4709 1 cat open64

Now introducing a 0.10 second delay:

> 948540416411500 4608 4728 1 cat open64 /dev/null
< 948540516660245 4608 4728 1 cat open64
dtrace: error on enabled probe ID 2 (ID 60351: syscall::open:entry): illegal operation in action #8
< 948540523879919 4608 4729 1 hostname open

Note that the first delay succeeded as intended, but that a subsequent delay affecting a call to open made by hostname (in this case, due to the hostname being displayed as part of the shell prompt in a separate user session) was cancelled as DTrace detected that this would exceed a safety threshold for delays to the system. That call to open proceeded without extra delay. This demonstrates the necessity of being very focused when injecting flaws; it would not normally be a good or useful idea to inject delays in all instances of any call.

This also demonstrates that DepMap has a constrained ability to introduce delays in system calls via DTrace, but that introducing delays can be done in a fairly safe manner without significant risk of forcing the system into an unrecoverable state. This limit and the basis behind it are described in (Sun Microsystems, 2008): "...interrupts are disabled while in DTrace probe context, any use of chill() will induce interrupt latency, scheduling latency,
and dispatch latency. Therefore, chill() can cause unexpected systemic effects and it should not be used indiscriminately. Because system activity relies on periodic interrupt handling, DTrace will refuse to execute the chill() action for more than 500 milliseconds out of each one-second interval on any given CPU.”

I encountered a minor issue with buffer sizing on the OS X port. DTrace uses a buffer to hold DTrace script byte-code. There is a default limit set on this space on all platforms. On Solaris and Linux this is tunable, but on OS X as yet there does not appear to be a supported option for increasing buffer size. I worked around this limitation by streamlining my DTrace scripts generated for OS X to be small in (compiled) size. This sacrificed some readability of the .d script, but as the “clean” code template used by the Generator remains verbose maintainability is not significantly impacted. I also separated functionality into multiple scripts such that only necessary functionality is loaded, to allow for future expansion of the scripts. The full set of DTrace scripts generated for each platform (“dmap_watch.d”) runs from approximately 900 (Solaris) to 1100 (OS X) lines of commented .d code, depending on options enabled.

4.4. Collector

The Collector is straightforward, consisting of shell scripts to invoke ssh/sftp to copy files between servers along with the included metadata regarding host name and collection times. Usage of the Collector is optional, and all analysis may be performed on the system under observation.

4.5. Analyzer

The Analyzer is implemented as a set of Perl scripts that verify the integrity of the data and then perform dependency identification and characterization, and summarization to time series data. The primary scripts are dmap_dep.pl (approximately 600 lines, commented) and dmap_ts.pl (approximately 400 lines), dealing with dependencies and time series data respectively. Both depend on a shared library of subroutines, as well as separate files that the user may customize to provide hints about mappings of values (e.g. sets of connections, processes, or devices that should be treated as similar, and aggregated in results) or the relative significance of certain actions. These comprise approximately 800 lines of sparse and very readable code.
The dmap_dep.pl script operates as a series of accumulators and state repositories. As it processes raw data from the Collector, it reorders items into chronological order and populates a set of simplified operating system data structures, such as a file descriptor table (unique per process ID) and a process tree. It transforms some of the information about each descriptor into a more readable format (e.g. a subset of the file flags are translated into "RD", "WR", "RDWR"), and uses this information to draw conclusions about the directionality of connections.

Based on mappings specified in a supporting .pl file, two types of weights are assigned to each actor (process) and connection (resource: socket, file, device). The weights are positive weights, signifying the relative importance of a successful operation (e.g. the size of a read operation or other measure of productive work), and negative weights (when an operation fails, e.g. -1 return from a system call or abnormally long time elapsed for call to complete). Calls are further differentiated as of type “actor” (causing some change of state in a shared resource, such as writing to a file), “observer” (primarily checking the existence or status of a resource), or neutral (such as closing a file). The result of this is a summary of the activities parsed from the Collector data, and annotated dependency map data in text form that can be used to understand some of the most active “contributor” and “detractor” actors and connections in the system.

Time series data is tab delimited with time as the first field, and file and field headers. As this is trivial I do not provide examples. On dependencies, below is a subset of the parsing summary from observations taken while using the Safari web browser to check my calpoly.edu email. The numbers are counts of records of that type. Note that this includes activity both against network services and local browser cache. The discrepancy between records read and calls parsed is due to header and footer lines.

Records read: 7171
Calls parsed: 7169

syscall counts:
read 2727 (actor)
fcntl 1595 (actor)
write 598 (actor)
close 355
pread 344 (actor)
socket 333
fstat64 320 (observer)
The dependency map consists of process (actor) records, and dependency (connection) records. For the above example, this map consisted of 74 records, many of which are one time file accesses by Safari, for example to check user preference files. A small subset of the output is shown below in tab-delimited format, including items with the largest positive weights.

<table>
<thead>
<tr>
<th>proc</th>
<th>dep</th>
<th>addr</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1097/?</td>
<td>6389.6389_any</td>
<td>129.65.64.31:443</td>
<td>186</td>
</tr>
<tr>
<td>dep</td>
<td>6389.6389_any</td>
<td>129.65.64.10:443</td>
<td>186</td>
</tr>
<tr>
<td>dep</td>
<td>6389.6389_any</td>
<td>129.65.64.61:443</td>
<td>234</td>
</tr>
<tr>
<td>dep</td>
<td>6389.6389_any</td>
<td>??/com.apple.Safari/Cache.db</td>
<td>1784</td>
</tr>
<tr>
<td>dep</td>
<td>6389.6389_any</td>
<td>??/Safari/WebpageIcons.db</td>
<td>83</td>
</tr>
<tr>
<td>dep</td>
<td>6389.6389_any</td>
<td>??/random</td>
<td>60</td>
</tr>
</tbody>
</table>

In this output, DepMap was instructed to collapse all threads into the enclosing process, thus 6839 was the process ID for Safari and all threads (TIDs) are treated as “6389_any.” The parent process of Safari is 1097, and as it as not observed in detail it is denoted with a “?”.

We see that socket activity is occurring over :443 (https/SSL) as expected, and that a relatively higher weight is associated with the email server than the login and portal servers, though all are significant. File prefixes are suppressed for readability, but we can interpret the top three file dependencies as being the local browsing cache, which is heavily accessed, a cache of page icons (e.g. favicon provided by the web server to the browser), and the random number generator which is likely somewhat time consuming and used in the SSL setup process.

4.6. Reporter

In addition to the result summaries and data formats previously described, two types of graphical reports are implemented: dependency graphs, and time series charts.

I utilize the Graphviz package for generating visualizations of dependency maps (AT&T Research and Graphviz.org Team). The Graphviz tools are available on all common computer platforms, and support a variety of graph types through different commands. For DepMap I use the “dot” and “neato” commands. Dot produces graphs with a hierarchical layout (layered top to bottom or left to right) and is used by DepMap for process-centric and maps, which are inherently hierarchical. This works less well on “wide” graphs, where processes have many dependencies to show. For these I use neato, which performs layout based on spring models. Both operate on either directed or undirected graphs; the only distinction in the case of DepMap reports is for visualization purposes, so I use undirected
graphs with directionality (forward, back, both, or none) specified for each edge. Recall that arrows to communicate the predominant directionality of some connections (e.g. a read only file) or the initiator (e.g. a listening socket will be back toward the process).

Both Graphviz tools take input in "dot" files and can produce a variety of outputs; I use PNG (bitmap) and SVG (vector). To generate .dot files from dependency files, I implemented dmap_mkdot.pl. Dependency information is translated into edge weights, pen thickness, directionality (arrowheads), and line type (e.g. solid to represent process relationship, dashed to represent a temporary connection such as a series of file or socket operations). Pen thickness for edges is scaled based on weight of the connection relative to all connection weights in the system, with a minimum of 1 and a maximum of 20. A .dot file based on the previous example is shown below, with boilerplate items removed for readability:

```
"1097" [ label="1097\n?" shape="box3d" ];
"6389" [ label="6389\nSafari" shape="box3d" ];
"1097" -- "6389" [ dir="forward" ];
"6389.6389_any" [ label="6389.6389_any" ];
"6389" -- "6389.6389_any" [ dir="forward" ];
"6389.6389_any" -- "129.65.64.31:443" [ style=dashed label=186 penwidth=2.3 dir="forward" ];
"6389.6389_any" -- "129.65.64.10:443" [ style=dashed label=186 penwidth=2.3 dir="forward" ];
"6389.6389_any" -- "129.65.64.61:443" [ style=dashed label=234 penwidth=2.9 dir="forward" ];
"6389.6389_any" -- "??/com.apple.Safari/Cache.db" [ style=dashed label=1784 penwidth=20.0 ];
"6389.6389_any" -- "??/Safari/WebpageIcons.db" [ style=dashed label=83 penwidth=1 ];
"6389.6389_any" -- "??/random" [ style=dashed label=60 penwidth=1 ];
```

The resulting graph generated by `dot` is shown in Figure 14.

![Figure 14: Hierarchical Graph of Checking Email](image-url)
Time series data reporting is implemented in Perl scripts including chart_ts.pl. This in turn relies heavily on the Perl Chart modules, originally developed by David Bonner and now maintained by the Chart Group at BKG-Wettzell (Group, 2006). This reporting can optionally perform smoothing and descriptive statistics using Perl subroutines I developed separately. For smoothing, DepMap supports moving average (mean), median, and chunked minimum, maximum, and mean. It also supports delta (derivative) and summing (integral). All of these filters can be stacked. Full examples are provided in Results and the appendices, but some cropped charts are provided in Figure 15 through Figure 19 to illustrate some of these outputs types. Each of these charts represent hundreds of thousands of raw data points per series:

![Median of Sequential Read Throughput Across 5 Runs](image1.png)

**Figure 15: Median of Sequential Read Throughput Across 5 Runs**

![CPU IOWait Status Versus Usr, System Time](image2.png)

**Figure 16: CPU IOWait Status Versus Usr, System Time**
Figure 17: IO Wait Time Relative to Processing, Two Week Period

Figure 18: Histogram of Read Timing

Figure 19: Scatter Plot of 1 Second Event Data
5. Results and Analysis

The best way to deduce the system’s purpose is to watch for a while to see how the system behaves. ... purposes are due to start from behavior, not from rhetoric or stated goals (Meadows, 2008).

DepMap was developed and tested on three platforms: Solaris X86 (on virtualized hardware), Linux (on virtualized and physical hardware), and Mac OS X (two sets of physical hardware). It was further applied in monitoring and diagnosing issues on two systems supporting users. Evaluation was performed on representative synthetic environments to assess the performance impact of running with DepMap.

5.1. Limitations on Operational Details

I describe the effectiveness of DepMap when used on actual production applications. In some cases I have omitted or changed details such as vendor names and IP addresses. This is done for two reasons. First, operational information on currently running systems may unnecessarily expose weaknesses of those systems. Second, license agreements for some vendors preclude publishing of benchmark results without authorization.

I believe that much of this detail is not necessary to understanding and verifying my results, and that which is relevant (such as IP addresses, port numbers, and protocol names) is neutral with regard to the interpretation of my results. In each case I note where results have been modified for these reasons.

5.2. Test Platforms

For MacOS/Darwin, the latest version of MacOS 10.6 (“Snow Leopard”) was used, and maintained with current patches throughout the implementation. Two hosts were utilized to ensure correct operation on differing hardware: a MacBook Pro laptop with an Intel Core2Duo processor (2 cores) and 2GB of RAM, and a MacPro with two Intel Xeon processors (4 cores) and six locally attached storage devices. The laptop was used for initial implementation and testing, while the MacPro was used for verification of concurrency and performance under high-I/O activity.
For Solaris, the latest release of OpenSolaris 2009.06 (SunOS 5.11) with patches was used, on a virtualized two processor server with 1GB or RAM. This represents a more mature DTrace implementation, and was used primarily to test database scenarios.

For Linux, 2.6.x kernels were used, with 32-bit on virtualized hardware (two processor, 1GB of RAM) and 64-bit on Pentium 4 physical hardware (two cores, 4GB of RAM). Physical hardware was used here in an effort to establish the most consistent possible timing information to measure impact.

5.3. Synthetic Testing for Basic Function and Overhead

The choices we make of what models and what modeling conventions to use to understand a particular problem are very important. This is because, as we endeavor to populate a model repository model formats, content, and syntax we choose largely defines what information we expect to retrieve ... if the model stresses one aspect of the system and ignores another, we are likely to stress and ignore the same attributes (Armour, 2003).

Basic functionality has been demonstrated previously through examples in Section 4, Implementation. Both time series and dependency data is collected, analyzed, and graphed, and agrees with sources that had previously been utilized, including developer knowledge of applications.

Correctness validation was performed by writing test cases to exercise specific portions of DepMap. These focused on basic I/O, INET sockets, shared memory, and other common types of connections that are introduced via system call. For example, one of the test programs for basic I/O is shown below, with error checking and boilerplate removed for clarity:

```c
r_fd = open(READ_PATH, O_RDONLY);
for (index=0; index < READ_COUNT; index++) {
    result = read(r_fd, r_buf, READ_BLOCK_SIZE);
}

w_fd = open(WRITE_PATH, O_WRONLY);
for (index=0; index < WRITE_COUNT; index++) {
    result = write(w_fd, w_buf, WRITE_BLOCK_SIZE);
    fsync(w_fd);
}

close(w_fd);
close(r_fd);
```

The dependency map produced (removing shared libraries for brevity) is shown in Figure 20. Note increased weight assigned to /dev/null due to calls to fsync: 
<table>
<thead>
<tr>
<th>proc</th>
<th>8799/?</th>
<th>13521/sample</th>
<th>285196</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep</td>
<td>13521.285196</td>
<td>/dev/zero</td>
<td>65</td>
</tr>
<tr>
<td>dep</td>
<td>13521.285196</td>
<td>.</td>
<td>3</td>
</tr>
<tr>
<td>dep</td>
<td>13521.285196</td>
<td>/dev/null</td>
<td>257</td>
</tr>
<tr>
<td>dep</td>
<td>13521.285196</td>
<td>??/ttys002</td>
<td>1</td>
</tr>
</tbody>
</table>

Graph visualization is:

![Graph visualization](image)

**Figure 20: Visualization of Simple Read Test Case**

Test cases were implemented in a variety of languages, including C, Java, Perl, and shell scripts. The intent of these tests was to test important cases (common modes of communication), not full coverage. In all, 20 test cases were run on each platform. In order to verify further that no relevant system calls were missed in the dmap_watch.d script for each platform, an additional output from Generator was created to watch and count only those system calls that are *not* monitored by dmap_watch. This produced no interesting results when run on the test cases, and many interesting results that were not relevant to DepMap when run on large software applications such as Firefox, Thunderbird, and Microsoft Office. Details are not relevant here, but suffice to say that most large end-user software packages appear to be very inefficient in the use of resources as judged by redundant system calls.

Testing was performed to determine performance overhead in three situations: for a single-threaded CPU-bound process, for a single-threaded I/O bound process, and for a multi-threaded web application workload. These tests were run in two environments: a virtual Solaris host, as the Solaris platform has the most mature DTrace implementation,
and a physical Linux host, which provides consistent timing information without potential for shared resource and other virtualization effects.

These test focused on two extreme scenarios, plus a real-world mixed scenario: high user processor utilization with few system call (prime factorization), high system call utilization in an I/O constrained single-threaded file reader, and a simulated workload against the Tomcat application server.

The CPU-bound test consisted of performing prime factorization. This workload involves few system calls, and has some sensitivity to cache coherency due to localized data access. In this scenario, it was expected that there would be little overhead, as no DTrace triggers should occur on a pure processing workload. Results showed negligible (<0.1% total time) impact due to DTrace and DepMap. This matches the design criteria and existing tests for DTrace, which suggest little or no impact when probes are not triggered (Sun Microsystems, 2008). This was reproduced on both test platforms.

The I/O-bound scenario consisted of sequential reads against a hardware RAID storage (four 1TB drives in a RAID 10 configuration) connected via eSATA. This scenario showed an average impact of 8.7% on Solaris due to detailed monitoring versus no monitoring, and an increase in variability. This is shown in Figure 21 based on data from the benchmark program to allow for comparison with and without DepMap running. Note that in some figures I have removed the horizontal axis for reasons of legibility; where relevant, full reproductions including the axis are included in Appendix A.

![Reads (MB/s) from First 10GB - By Second](image)

**Figure 21: Performance Impact from DepMap Monitoring, I/O Test**

Vertical units are MB/s read from the storage array, while samples along the horizontal access are per second. The high, stable line is without monitoring (192-236 MB/s, mean 231 MB/s), while the fluctuating line is with DepMap targeting the benchmark executable (198-221 MB/s, mean 211 MB/s). I believe the fluctuation is due primarily to high rates of data output by DepMap being flushed to disk, while the overall slower performance is due to call monitoring overhead. If the frequent drops could be prevented, it appears that the
sustained throughput under detailed monitoring would be roughly 218 MB/s, a 5% impact compared to the unmonitored case. It may be possible to accomplish this goal by aggregating within DTrace the activity against high-throughput descriptors and reporting it on a periodic basis using the built-in sampling facility (i.e. profile-n).

This test did not finish to completion on Linux, as DTrace reported drops and in one case caused a kernel panic. This demonstrated that the Linux port of DTrace clearly does not meet the safety guarantees of the Solaris implementation. On many test cases, however, the Linux port performed acceptably suggesting that a race condition exists in the way that code interacts with the kernel. I reported this issue to the developer.

The Tomcat test showed two results. First, startup of Tomcat was approximately 50% slower with detailed monitoring than without. DepMap results show that this is due to the large (9,000+) number of system calls made by Tomcat at startup, primarily to check for the existence of a localization files and to load a broad range of Java class files including independent reads of those within compressed archives such as classes.jar. After startup, while running a synthetic load against a test application, the impact was reduced to a 5% increase in time to complete the workload.

These tests establish a lower (~0.1%) and upper (8.7%) bound on performance impact for expected cases. Impact may be higher in cases where an application does little processing and many small I/O operations; often, this would be the sort of behavior that DepMap could help to identify so that it could be improved by the application developer.
5.4. Real-World Applications

“...there is evidence that our ‘ignorance’ always exceeds our knowledge” (Armour, 2003).

DepMap has so far been applied to identify or confirm root cause of performance issues for two quite different systems. The first will be described briefly, as it illustrates the range of techniques that DepMap enables in diagnosing an issue, and a common type of result. The second will be described in more detail. It was not a use case for which DepMap was originally envisioned, but DepMap analysis tools were able to isolate a long-standing and costly problem.

The first scenario involves a web-based application that provides a critical service to a large number of users. This application is hosted across multiple hosts for performance and redundancy against host failures. Some information to satisfy transactions is replicated at sub-second timings between the hosts, while other information is obtained from separate network-based services. Very infrequently, on the order of once per month, this application stops responding on all of the hosts. This does not appear to correlate with time of day, level of user activity, type of user activity, or involved users. The application had been running with no significant modifications on the same hosts for four months prior to this issue first occurring.

Using DepMap techniques against the application running in an equivalent test environment showed that replication between the hosts was slower than expected, and more highly variable than expected. As these results were observed at the system call (kernel) level, this ruled out responsibility on application code: the number of calls was as expected, but the delay in responses to calls was not. This showed that the most likely root cause was network or hosting issue, or hidden (indirect) dependencies behind these.

After describing my results and connection characterizations to a developer, he was able to create a synthetic (simplified) benchmark that reproduced this behavior. Graphs comparing performance on the previous hosting environment (first graph) and the newer environment (second graph) show a striking difference:
The vertical scale on both is zero to 400 milliseconds, though in each case where a vertical green bar crosses 400 it actually recorded as 2,000 milliseconds (two seconds), the point at which the network connection timeouts. The horizontal scale is the same for both an is in minutes. Effectively, where the old hosting environment shows infrequent slow performance and very occasional packet drops, the new environment shows more regular failures and periods of sustained failure.

As is often the case, while exploring this issue in detail and working to model it in a test environment, I found two other systemic issues on other large-scale services, apparently unrelated to this issue. One involved highly variable response times from a service that has low server resource demands but high consistency demands (NTP, the network time protocol). This was due to a poor architecture decision (shared server hosting service with quite different service requirements), compounded by the second issue. The second issue was consistently poor performance (high latency) of a critical service (DNS, the domain name resolution service). This was due to ineffectual load balancing, compounded by inefficient application performance. Work is in progress to resolve these issues.

The next scenario involved a database server hosting a data warehouse. This system pulls in hundreds of gigabytes of data on a regular basis. Data can be pulled only after processing has completed in the source databases, thus the start time for the pull is constrained. This data then undergoes transformations to produce a variety of outputs, which must all be completed well before the start of a normal business day, so that consumers of the data can meet their operational needs when they begin providing service to their customers. This occurs in a production environment, as well as development and
validation environments. Each environment has entirely separate server hardware, but shared network and storage infrastructure. Run times in all three environments had been growing worse, to the point that production service targets were sometimes not met (nightly batch processing requiring over twelve hours), and the development environment was run sparingly as processing could take over eighteen hours and cause performance impact to production runs. The trial and error approach of making configuration changes at various levels of the technology stack ("easy fixes") had been exhausted. When I became involved with this effort, this issue had been impacting users and developers for over a year. My goals for this effort were to characterize the workload, find measures to quantify the performance issue, determine most significant causes of overall poor performance and recommend corrective actions.

The first step was to determine overall system behavior. As runs on one system appeared to cause impact to runs on another, I worked under the hypothesis that a resource shared between the servers was a primary factor. An issue with shared storage seemed the most likely cause, though this left a number of specific components as possible contributors: fiber channel switches, cables, cards, caches, controllers, and hard disks were all shared to some extent between environments as well as with other unrelated workloads.

Previous data collected from the storage infrastructure at five-minute intervals showed moderate I/O throughput and no resource contention issues. Diagnostic information from the database product suggested high I/O wait times as causing performance issues, but this information was largely ignored based on benchmarks showing that the storage infrastructure (disks, and path to disks) was capable of much better performance than was being realized.

Data collected using DepMap’s extended iostat collection mode showed that the system was in fact spending much of its time waiting on I/O. This is easily charted based on the value reported as “iowait%”. On a system with eight processor cores this value would be 12.5% (i.e. 1/8th of the system) when a single process is spending all of its time waiting on I/O. This data alone ruled out a number of recurring theories that the cause of poor performance was a bottleneck within the database software. The impact of this situation can be magnified for visualization by dividing the time spent waiting on I/O by the sum of the time spent on all other activities (excepting idle). The following graphs show this metric, with the vertical axis from zero to 40 (signifying a system spending 40 times as long
waiting on I/O as doing useful processing), and the horizontal axis times from 16:30 to 10:00 the following day:

Figure 24: Proportion of System Time Spent IOWait - 1 Minute Samples
These two graphs illustrate an important point: averages are very misleading when monitoring cyclical behaviors, and five minutes is an eternity in system time. By sampling at one-second intervals, the problem becomes even more obvious, and specific times of highest impact are identified. During certain times, this metric held at 100, meaning the system was spending 100 units of time waiting on I/O for every unit of time doing productive processing. Effectively, no work was being done during these times. No root causes can be identified based on this alone, but a check of other processes running at these times showed a correlation with server backup schedules. Backups were rescheduled to spread out their impact (minimize peak load), with the result shown below. This chart shows the same metric with the same vertical scale, but over a seven day period. The day after the change can be clearly seen, as the extreme spikes have disappeared.

Performance as measured by workload runtimes was also improved, though still poor. The most glaring issue had been addressed, and a few hours of runtime had been recovered, yet the system continued to spend 5-10 units of time waiting on I/O for each unit spent processing.

I now focused on characterizing the workload from a system perspective. A subset of the DepMap functionality showed that the I/O requests taking the most time to complete, consistently throughout the workload, were of two types: synchronous sequential writes, and asynchronous sequential reads. At the one-second sample rate, the completion time for these operations varied widely and was slow on average. I addressed these two observations separately.
First, the extreme variation in completion time is not normal for sequential operations on isolated disk sets, and should not be so extreme on such a large set of shared disks, in this case over 50 disks. By comparing timing for the I/O operations performed at equivalent points in separate runs of the workload, I began to see patterns of repeating better and worse throughput from the shared storage device. I then created a benchmark to create total reproducible workload. This benchmark performed related workloads at various layers of the technology stack: selects from large tables at the database layer, sequential reads through the data files at the file system layer, and sequential reads against the raw device interface to the storage (e.g. "/dev/sdxy") at the operating system device layer. The results were similar for each, which demonstrated that this pattern was not due to issues with database or file system block layout. The first graph below shows the raw data for one of these benchmarks, with logical block position along the X axis and throughput in KB/s on the Y axis. Five runs are shown, each in its own series. The next graph shows the data after I applied a median smoothing filter:

![Figure 27: Sequential Reads Against Device - Raw Timing](image-url)
Figure 28: Sequential Reads on Device - Median Filtered

Note how, at some positions within the sequential read, many of the test runs correlate. The arrow emphasizes one of the more dramatic points; at this point all five runs correlate. Generally, three of the runs correlate very well but with some notable variations; I believe these types of variations are caused by other load present on the storage device that coincides with (or is conducive to, in the case of peeks in throughput) the access patterns of my sequential workload. The degree of commonality between runs becomes more apparent by additionally using a delta (derivative) filter, as shown in the next graph:
The two runs that appear different towards the center are from periods of time when I ran I/O requests concurrently from the server against other volumes on the storage hardware. They both saw higher average throughput of the benchmark workload, in spite of having more total locally generated I/O, suggesting that some element in the chain between this server and the disks rewards a greedy consumer in its queuing model. Also note that the throughput generally trends downwards as we proceed through the volume; I'll return to that in the next set of observations.

Overall, this correlation in performance difference between runs strongly suggests suboptimal block layout across this storage system. On an isolated disk system, this would be called fragmentation; in this case, there are too many variables and too little ability to observe at a fine-grained level within the storage system to establish whether it is really an issue on individual disks, or with the layout across disks. This theory was strongly disputed, as the management software for this storage system reported a measure
suggesting this was not an issue. Running the equivalent of a forced defragmentation across one device showed a 25% improvement in sequential read performance, validating the theory.

On the second observation, about slow average performance: Synchronous write operations require that execution not continue until data is actually stored on disk, or at least in a battery-backed cache that can be applied to disk (and read back by the database) even after a system failure. These writes are performed by a database as a means of implementing the ACID properties. For example, the durability property requires that before a change can be considered committed, the database ensure that a record of the change will exist on disk, even if the system power were to fail in the instant after the “commit” is processed.

Synchronous writes appear in system call patterns as write followed by fsync, or equivalently operations that configure a file descriptor to be considered as synchronous, such as open with the flag O_SYNC. DepMap readily identifies processes and files that match these patterns.

Synchronous writes tend to be a worst case from a performance perspective, as they can stall an entire system until they complete; high latency for these operations can cause a system to perform little work. The workload in this scenario involved hundreds of gigabytes of changes within transactions, and thus hundreds of gigabytes of synchronous writes. This situation should not be the truly worst case, however, as the writes are sequential. Conceptually, the database simply appends to a file until it is full, at which time it uses a different file. Given a single spinning disk, we can envision the head tracking smoothly across the disk surface as the database calls write. In practice, these write operations are still problematic when they are mixed into queues with other I/O, and in some cases must force all other pending operations to complete before they can complete.

Worse, on a shared storage system it is extremely unlikely that these writes have a disk to themselves. The prevailing model among storage vendors and system architects is often to spread all I/O across as many mechanical disks (“spindles”) as possible, with striping and mirroring for performance and ability to recover from disk failures. This works well for a large number of small I/O operations, as occurs with many random readers (which see a cumulative throughput approaching sequential), but actually works less well than our imagined single disk for sustained sets of sequential writes. As workload from other

77
servers is mixed in to ours, performance approaches purely random access. Visualize two competing systems writing sequentially to different locations on disk. As their requests are interleaved, the heads must continually seek back and forth to satisfy both. This is the worst case on a mechanical disk. Much of this holds for sequential read operations as well, particularly when they are sustained to sizes that exhaust the caches of servers and shared storage devices as they did in this case. That is, read-ahead “anticipatory” caching would not always be possible due to constrained memory.

Thus the most common elements of this workload, which should be well handled by strategically allocated individual drives, are a worst case for a shared storage infrastructure. Effectively, a set of 50+ very fast drives behind a large memory buffer and connected via a fast interconnect performs more slowly than a single commodity drive for certain workloads, and when the fast drives and cache are shared with other workloads. I demonstrated this through benchmarks:

![Figure 30: Sequential Read Throughput from Various Devices](image)

The vertical axis is MB/s, with marked intervals at 50 MB/s and the top at 250 MB/s. The horizontal axis shows seconds elapsed, with faster devices completing the benchmark in less time and the bottom two lines, hovering below 50 MB/s, not completing within the graph scale. The bottom two lines are the large shared storage device, the middle line is a single 7200 RPM SATA (commodity) drive, and the top lines include runs on three different
devices: RAID 10 set of four 7200 RPM drives, RAID 5 set of same, and 200 GB solid state drive (SSD). The results above the 50 MB/s line are from hardware costing less than $1000 today. The results below the line are from hardware costing orders of magnitude more. This demonstrates the sort of significant performance and cost benefit that can be achieved by selecting system architectures based on workload.

Returning to the observation that throughput trends gradually downward as my benchmark read through the shared storage device, below is a graph showing write throughput across entire devices, again including a single disk, various RAID sets of four disks, and an SSD. The shared storage device is not included here, as this sort of destructive (write) test was not possible there.

![Figure 31: Sequential Writes to Fill Device](image)

Note that throughput for the single drive (lowest line) falls slowly at first, and then accelerates. This is due to the linear velocity of the disk tracks decreasing as the head moves from the outer edge of the disk to the inner edge. Put another way, the disk spins at a constant rotational velocity, and has constant spatial data density, so there is more data per revolution at the outside edge. This effect is less pronounced in the RAID sets as the writes are distributed across more disks, which are thus individually less full. The SSD shows none of this effect, though other effects are possible with nearly-full SSDs, SSDs with suboptimal controllers and allocation strategies, and SSDs that are nearing their write cycle.
limits. The performance from the shared storage device suggests that each virtual volume, even if spread across a number of physical disks, is in some cases allocated according to a "spread across the entire disk" strategy rather than a "keep this volume on a fast part of the disk strategy."

Resolution: these theories were further confirmed by running portions of the actual workload in a very limited prototype environment. A small number of external, commodity, SSD devices were added to one of the systems performing actual work, and a subset of data accounting for a small majority of activity was moved to this storage. The result was dramatic improvements in build performance, going from a high of eighteen hours to a low of less than eight. The system is now totally idle at points in the build (as static schedules have not been adjusted to this new level of performance), or CPU bound. No further tuning is being attempted, other than to further measure potential for improvement. New servers with totally isolated storage have been ordered, and based on models I built using information from DepMap is will be possible to complete a build in less than four hours, roughly 20% of the worst case time seen previously. The cost of this approach is also substantially lower than expanding the shared storage device, which was previously suggested as a solution to the performance issue.

5.5. Conclusion

I have demonstrated that DepMap is functional within its requirements, carries moderate performance impact that is acceptable on many systems, and is extremely applicable to both identifying and characterizing the communication dependencies in a software system. I have also shown that basic visualization approaches such as dependency graphs and time series charts are valuable in understanding the behaviors of complex systems. Beyond synthetic tests, I have demonstrated the usefulness of portions of the DepMap toolset in isolating and solving real world problems to save both time and money.
6. Future Work

*First, one must perform perfectly. ... if one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect (Brooks, 1995).*

In the course of this work I identified many avenues for further research, either in the support of my goals for DepMap or to reuse elements from this effort to other solutions.

From previous work with CAMP (Gabel & Haungs, 2007) I recognize the value of a simple, consistent API for accessing performance data. In effect, DepMap implements two additional API families that are relevant to CAMP. At present this API is in the form of file formats that internal to DepMap, and are parsed in large batches. There is no facility to easily poll in anything approaching real time. I understand that further work is in progress to add a distributed call interface to CAMP (Haungs, 2010). By implementing a stream- or poll-mode in the processing of DepMap data, CAMP APIs might be built for DepMap data. The time series DepMap functionality appears to be the best fit for the CAMP model.

Sampling could be used more widely to reduce the impact of DepMap on application performance. It is common within DTrace scripts to collect information into an aggregate variable; I suspect that much of the overhead in the dmap_watch.d scripts are in the operations required to output results after each probe rather than the probe invocations themselves. The profile-N feature in DTrace could be used to report at periodic intervals for high frequency falls such as read and write. At the same time, it is important to remember the dangers of over aggregating or averaging, in masking periodic or intermittent issues. These might be balanced by switching between the two modes, or where sets of descriptors are consistent performing sampling on most and full observation on a minority.

DTrace includes many providers besides syscall, and new providers can be implemented. These could be used to supplement or in some cases replace some of the syscall probes currently used by DepMap. Application probes in particular should be highly beneficial in correlating application dependencies and issues with operating system observations.

For flaw injection, at present DepMap only injects delays in returning from user-specified system calls with user-specified filters. This could be automated to, for example,
use the weights calculated by the Analyzer to automatically inject flaws in the highest or lowest weighted connections. Data modification injections could also be introduced; the most obvious is to force a return code of -1 from some system calls, signifying an error. This would expose a number of cases where application resiliency could be improved, with moderate risk of data loss (only useful on test systems). Beyond this, arguments and data returned to user processes could be intentionally corrupted. It would be much more difficult to generalize this, but in specific cases it could be a very useful tool for reproducing an intermittent issue. There appears to already be some thinking about the need for a generalized framework for black-box failure injection on distributed systems, including (Lipcon, 2010).

While DTrace is a well-designed and well-documented tool, support on Linux is uncertain. The existing port was a project of one person, and progress appears to be stalled. It does not provide the same safety guarantees as DTrace on other platforms. The future of DTrace is relatively assured due to extensive use within Solaris and OS X, and growing use within the BSD community. As Oracle Corporation now owns Sun and an interest in DTrace, but is also a founding member of the SystemTap project, there is potential for significant developments with DTrace on Linux. In the meantime, for small-scale instrumentation efforts on existing Linux system, SystemTap may be a more practical solution. Given DepMap’s model of generating code per-platform, it may be feasible to create a SystemTap-based implementation for Linux.

While DepMap is usable and the code is well documented, it is not currently packaged in a form or documentation such that a typical system administrator or developer could easily use it. While my emphasis so far has been on evaluation of this approach and application to scenarios, I have identified some “sweet spots” where a subset of DepMap could be packaged for use on common cases with minimal DepMap knowledge required by the user. For mapping of dependencies, these include “show all file dependencies with emphasis on frequency of use and directionality,” “show all network dependencies,” “show all network connections initiated” and “show all listening connections.” For characterization of a specific dependency (specified by a pattern match), these include “show distribution of latency for requests involving dependency,” “show number of requests over time,” and “show frequency of volume of data via dependency over time.”
DepMap is prone to generating graphs that can overwhelm, particularly on large systems that access many files. I address this to some extent with options to group files by common directories (e.g. "/tmp/x and 10 others here"), but a more useful solution might be to generate hyper-linked views of dependencies. A high level map would show only the most strongly weighted (positive and negative) dependencies, with links to drill in to see all dependencies of a specific type. The .dot and .svg formats both have functionality to label and link between documents, and thus this mode of use could be supported via a web browser accessing HTML and SVG content.

DepMap handles correlation of dependencies within a host by linking on shared connection identifiers. For example, two processes accessing a common pipe both link to that pipe. This same approach would work to correlate dependencies between hosts, though the issue of aliases I described in Challenges becomes more severe. This could be lessened somewhat through hints or additional information collection to handle common cases. For example, if host A knows it has an interface at 192.168.1.1, and receives a connection request on a socket at "0.0.0.0:80" (request to port 80 on any interface), while host B knows it initiated a connection to 192.168.1.1:80, there is sufficient information allowing for timestamp variation for the Analyzer to correlate these events and build a map involving data from multiple hosts.

DepMap is a rich source of information. At present I use a set of hard-coded heuristics to establish weights and correlate dependencies. More advanced analysis techniques, such as Bayesian models or neural networks could be used to interpret this data. Additional smoothing and prediction techniques, such as Holt-Winters, could be applied to time-series data.

Much more is possible in the area of visualizing DepMap results. I am especially intrigued by the possibility of animating dependencies and time series information to show changes over time as additional information is observed from the target system. Further work could answer the question: How do interpretations change as information is added?

Tools like BaconMap (Zenisek, Carr, & Sanchez, 2010) are often used to manage high level dependencies (e.g. serverA depends on serverB). DepMap may be able to inform the automated population of these sorts of tools once connections can be correlated between servers.
7. Bibliography


Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Design. Addison-Wesley.


8. Appendix A: Full Page Diagrams and Charts with Explanation

Many of the graphical outputs from DepMap are best suited to a large, high-resolution color display. This appendix provides full-page renditions of sample output, each with a brief description.
Hierarchical Graph of Checking Web-based Cal Poly Email Using Safari Web Browser: Network and File Dependencies

The Safari web browser is process ID 6389. The heaviest dashed line depicts a dependency on the local cache, which is accessed frequently by Safari and provides a large amount of data. 129.65.xx addresses are campus web servers. WebpageIcons.db is used to cache small icons associated with web sites. We can see that Safari uses the random device to seed SSL connection setup for the web sites.
The Safari web browser is process ID 6389. Compared to the Cal Poly web-based email service, a typical Gmail session involves interactions by the client with more IP addresses. The heaviest dashed line represents connections to a host which apparently serves most of the content, from 66.102.7.97, though all of the hosts are in the 66.102.7.x class C subnet. As of the time of this session, reverse DNS resolved all of these addresses to 1e100.net, a domain operated by Google (1E100 interpreted as a number in scientific notation is a “googol”), with host names containing "lax" suggesting service from a data center close to the client.
Hierarchical Graph of Starting Up and Shutting Down a Tomcat Application Server: Network Dependencies Only

This graph shows all of the network connections involved in starting up and then immediately shutting down a Tomcat 6 application server. Threads within a process are shown as boxes with labels beginning ".", for example the ".255944" thread appears to service most of the network connections. The "11600 java" process is spawned by the shutdown command. This process connects to the running Tomcat instance and issues commands to instruct it to shut down. We also see that thread ".255944" listens on both :8080 (any IPv4 address available on the host, on TCP port 8080) and on :fff:127.0.0.1:8080, the IPv6 localhost alias. These are the ports on which HTTP requests are accepted.
This graph shows DNS query response time for three DNS servers. The primary server is shown in red. The vertical axis is in milliseconds, up to 5000 which is a common client timeout value for DNS requests. Response times on the primary server often reach 300 millisecond peaks, which is somewhat high and suggests the primary server is overloaded. The large peak one-third of the way from the right side is associated with a failed upgrade attempt, which resulted in an outage of the primary server and a peak in demand as the new configuration failed to handle requests correctly. The quiet period on the right is an academic holiday week.
This graph shows DNS query response time with the vertical scale to 1000 milliseconds. The three thirds, from left to right, are the end of an academic year, a quarter break, and completion of a DNS server upgrade. From left to right we see typical peaks of 200-300 milliseconds, followed by low utilization (but with regular delay spikes on the secondary server in green). On the right we see that the upgrade (which involved both a software and hosting environment change) has caused response times to be less consistent, and with frequent peaks above one second.
This graph shows DNS query response time with the vertical scale to 2000 milliseconds. The left half shows poor performance following a software and hosting environment change. The right half shows consistently good performance on the primary (red) and secondary (green) servers, with response times typically below 50 ms and few peaks to 250 ms. The blue peaks are due to a separate systems issue.
This graph shows system processor utilization during a 30 minute data transformation job. This system has 24 CPU cores. Green points represent time spent doing user or system processing, while red dots represent processors waiting on I/O. We can see three modes the system operated in during this period: low processor utilization (green and red dots near the horizontal axis), high processing utilization (green dots above the “8” near the left side show eight of 24 cores doing productive work), and high I/O wait (red dot peaks). We see that we attain productive parallelism above two cores infrequently, and these are often associated with high I/O wait times. This suggests two things: that this workload is not effectively utilizing the processor cores for much of this process, and (related) that the system is I/O bound under heavy process concurrency.
This graph shows storage device utilization during a 30 minute data transformation job. Four devices (each representing a RAID set) are represented: a SSD (solid state) used for data storage in green, 15K RPM hard drives used for data storage in red, SSD used for redo logs in blue, and 10K RPM hard drives used for backup/recovery in black. The vertical scale is utilization, with 100% representing that requests were issued to that device continuously (that is, as soon as the device completes on request another is made, suggesting that the device is saturated). The time frame of this chart matches that on the previous page, allowing us to see that even under periods of low parallelism we have high utilization of the 15K RPM drives. This may be caused by random data access patterns or a similar effect due to interleaved streams of sequential requests. We see that high parallelism is achieved when access is primarily concentrated on SSD devices (which have less penalty for non-sequential access), and that during those times we have high utilization of the redo SSD devices (associated with completed database updates).
This chart shows cumulative I/O activity to a storage device over a period of time. Green represents writes, red represents reads, and black is the sum of both. The vertical scale is gigabytes, up to 2000, and horizontal scale is time of day from 20:00 to 08:00 the following morning. We can see that roughly 1750 gigabytes of total I/O occurred during this time, and that writes were somewhat more common until 04:00, after which reads were more common.
This chart shows I/O activity per over a seven day period. Once again, red is writes, green is reads, black is total. Peaks are associated with nightly data transformation activity on this database server, while periods of high green and low red are associated with query and data analysis activity (no updates, no writes).
This chart shows read throughput in KB/s, reading data sequentially from a shared storage device in five separate runs (with caching affects minimized). Data has been smoothed using a median filter to better show recurring patterns between runs. The three black lines are from runs with no other activity running from this server against the shared storage device, while the red and green lines are from runs with other concurrent I/O activity from the server. We can clearly see a repeating pattern in troughs in throughput. This suggested fragmentation, or suboptimal block layout on the storage devices causing logical sequential I/O to result in partially randomized physical I/O.
This chart shows the proportion of time a server spent waiting on I/O, as a multiple of time spent doing productive processing. On the left we see three substantial peaks, to over 40 (signifying that for every unit of time spent processing, the system spend 40 times as long waiting on I/O). This was caused by competing I/O performed by another system against shared storage devices, which starved this system for I/O. On the right we see that the most severe peaks have been alleviated after the competing I/O had been rescheduled to run at a time with less impact.
This chart shows the distribution of response times for read requests against a shared storage device. Requests for cached data are shown at the extreme left, with wait times of 0-2 ms (depending on where in the system the caching occurred). Requests involving sequential access are spread across the left side of the chart, while random access is clustered in the middle (12-16 ms) due to seek times on media. There are a significant number of requests requiring over 30 ms to complete, suggesting a bottleneck in the system for some types of I/O loads, or some time periods. In this case, other data showed that synchronous write operations were completing slowly.
Throughput (MB/s) for 1TB of Sequential Write Operations

This chart shows throughput in megabytes per second for sequential write operations to a variety of storage devices. The orange line (lowest) shows the typical throughput falloff for a rotating hard disk drive, with best throughput as the heads are near the outermost track (higher linear velocity) and lower throughput near the innermost track (lower linear velocity). In this case, a 7200 RPM 1TB drive ranges in speed from 105 to 55 MB/s. The blue line shows throughput for the same operations on a RAID 5 (striping, parity) set composed of four 7200 RPM 1TB drives. We see that performance is overall better, as load is spread across more drives, but less consistent due to affects of the RAID controller (such as parity calculation and writing in RAID5). The green line shows the same operations performed on the same drives, but this time in a RAID 10 (striping, mirrored) configuration which has less penalty for write operations at the expense of lower available usable space. The red line at the top left is for a single 200GB MLC SSD (solid state) drive, which consistently achieves 230+ MB/s in writes.