Using Time Travel to Diagnose Computer Problems

Andrew Whitaker, Richard S. Cox, and Steven D. Gribble
University of Washington
{andrew,rick,gribble}@cs.washington.edu

1 Introduction

The solution to a number of modern computer problems takes the form of a manual, expert-guided search through a large space of computer configurations. For example, if a desktop computer is crashing or malfunctioning, a troubleshooter will use her knowledge of system features such as configuration files, registries, and dynamic library versions to apply a series of configuration changes until the system once again begins functioning. As another example, to obtain good performance from a complex system like a database or a web application, a specialized and highly paid administrator will explore the set of application and operating system parameters to find the optimal values.

Our goal is to move the burden of this search process from humans to machines. If we can provide appropriate mechanisms to automate the search process, many systems issues that are currently complex, expensive, and time-consuming will be simplified and made accessible to non-experts. In effect, we want to apply goal-directed optimization techniques to the problem of finding a good system configuration out of the space of possible configurations.

Related projects have tackled similar problems by modeling system behavior [1, 8]. However, modeling is time consuming and error prone, as it requires a person to generate an accurate enough model to capture the system's relevant behavioral properties. Instead, we propose using virtual machine monitors (VMMs) to directly execute the system itself inside a virtual machine [4, 13, 15]. Assuming that the VMM is able to faithfully recreate the physical machine's behavior, our approach can capture all nuances of a system without requiring deep knowledge of how it works.

In the rest of this paper, we focus on one problem in this general class: diagnosing computer configuration errors. In Section 2, we describe the Chronus diagnosis tool, which finds errors by searching through the timeline of previous system states. In Section 3, we relate some of our early successes and experiences with Chronus, and we describe some of its inherent limitations. After discussing related work, we conclude, and we describe future work on the more general problem of finding good configurations in a large search space.

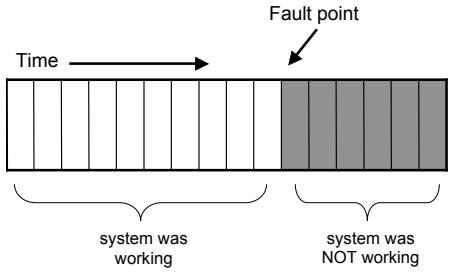

Figure 1: **Fault Diagnosis With Time Travel:** Chronus logs all changes to system state so that it can emulate system behavior at arbitrary points in the past. By using search, Chronus determines the instant the fault was introduced.

2 The Chronus Diagnosis Tool

Computer failures are often caused by changes in the computer's configuration or runtime environment, such as dynamic library upgrades, Windows registry modifications, or errors in Unix "`/etc/rc`" files. Troubleshooting such errors requires a deep understanding of arcane system features, and asking ordinary users to master this knowledge is like asking a non-mechanic to repair his own car. Our goal is to automate the process of diagnosing configuration errors by navigating through the space of possible configurations, attempting to find one that results in a functioning system.

In Figure 1, we illustrate our strategy for performing this search. We assume that a configuration fault, such as installing an incompatible library, takes a computer from a functioning state to a malfunctioning state. If we can maintain a complete log of system states over time, once a fault is detected, we can search through the past states of the system for the precise instant that the system first entered the faulty state. There are two benefits to this approach: we can use binary search to quickly "hone in" on the point in time where the fault occurred, and we can use the log of state changes to map from an observed behavior, such as an application crash, to a low-level state event, such as an update to the `libc` library.

2.1 Chronus Architecture

Our tool, called Chronus, explores configurations the system experienced over time, and diagnoses fail-

ures by comparing the system state before and after a problem arose. We rely on four components:

- **Time-travel disks.** Chronus logs all disk updates of a running system, giving it the ability to recreate any past system disk state.
- **Virtual machines.** By using a virtual machine monitor in combination with time-travel disks, Chronus can create a VM that emulates the system at some point in its history.
- **An "analysis" engine.** To find a fault, the analysis engine navigates through past configurations to find the state change responsible for causing the system to malfunction.
- **Software probes.** To test configurations, we run probe code *within* the VM to validate whether the system is functioning correctly.

Chronus focuses exclusively on state changes to stable storage. This contrasts with the traditional notion of checkpointing, which also includes memory and CPU state. We observe that many configuration changes require an application or system restart before they have an effect, and therefore instantaneous system snapshots are not necessarily meaningful. An additional benefit of Chronus's disk-only checkpoints is that they impose little overhead beyond the space required to maintain a disk history, which is known to be manageable [11].

We have implemented a prototype of Chronus on top of the μDenali virtual machine monitor [15]. μDenali is an extensible VMM, in that it allows a "parent" VM to modify portions of the virtual architecture of "child" VMs. Figure 2 shows the overall Chronus architecture. The *parent* VM implements the time-travel storage layer in a software module called `TTDisk`. A *child* VM executes normal user programs, and is oblivious to the presence of the time-travel functionality. After a problem is reported, an *Analysis Engine* inside the parent VM automates the task of searching through time for the instant that the problem emerged. At each time step, the Analysis Engine boots a new VM, and runs a software probe to indicate whether the system was in a correct state. We now describe these software components in more detail.

The amount of new implementation beyond the μDenali support libraries is 1645 lines of C code. Chronus runs on the NetBSD operating system.

2.2 Time-travel Disks

A `TTDisk` extends the μDenali disk interface by recording all block writes to an append-only log, in a manner analogous to a log-structured file system [10]. With this model, a "timestamp" is simply an offset into the log, and "time-travel" is implemented by ignoring block writes after a given timestamp.

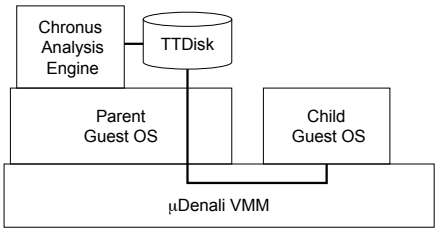

Figure 2: **Chronus Architecture:** During normal operation, disk writes are logged to a Time Travel Disk. During analysis, Chronus rolls back time and runs a user-provided software probe to test whether the system was in a correct state.

Within the μDenali VMM, there is a one-to-one correspondence between a `TTDisk` and a virtual machine. Chronus provides an administrator utility called `forktt`, which creates a new `TTDisk` from a read-only base disk image and an initially empty log disk. The implementation of these storage abstractions is hidden behind the μDenali disk interface. Presently, we map these disks to files in the parent's local file system.

During the analysis phase, it is crucial to quarantine the side effects of search probes. To this end, the `TTDisk` instance is wrapped by a copy-on-write (COW) disk prior to each probe. Once the probe has terminated, the COW delta is discarded, in effect garbage collecting side-effects that occurred during the probe. Of course, the child VM being probed is oblivious to the COW and `TTDisk` storage layers.

2.3 Analysis Engine

The Analysis Engine takes as input a user-provided software *probe*, which tests whether the child VM was in a correct state at a given time step. Using this probe, the Analysis Engine searches across the child's timeline for the instant the system transitioned to a failed state. At each time step, the child VM is booted from the reconstructed past disk image. By running the probe, the Analysis Engine learns whether the search should continue in the future or in the past.

The Analysis Engine quickly isolates configuration errors by using binary search. We start by running the user-provided software probe at the first and last time steps. If the results are the same, Chronus quits because further probes will not yield meaningful results. Otherwise, Chronus uses binary search to recursively find where the fault point must lie. Unlike a traditional binary search, our algorithm is not looking for a particular element, but rather the transition from one state to another. Therefore, the best-case and worst-case runtimes are the same.

Strictly speaking, the automated search only reveals *when* the failure occurred. Using this information, it is possible to uncover the source of the error by comparing the disk state before and after the

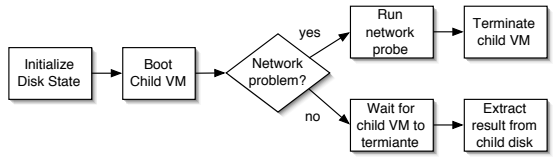

Figure 3: **Control Flow for a Probe:** Chronus distinguishes between external probes, which are run from the testing VM, and internal probes, which are run inside the VM being tested.

failure. Our prototype currently mounts the `TTDisk` before and after the failure, and uses the UNIX `diff` tool to determine what has changed.

2.4 Software Probes

A software probe is system- or application-specific code that tests whether the system is functioning correctly. For example, a probe may validate that the system booted correctly, that a daemon (like `sshd`) runs and permits remote login, or that a web server is correctly serving documents. Software probes allow the system to validate whether or not a specific configuration contains a fault. Crucially, *probes do not attempt to explain the fault cause*; they simply test whether the fault exists.

Chronus distinguishes between two styles of probes. *External* probes are run from the parent VM, probing the child VM over the network; these are typically useful for diagnosing problems with network servers. *Internal* probes are run inside the child VM itself. To extract the result from an internal probe, Chronus allows for a user-provided post-processing routine, which has access to the child's disk state after shutdown. For both styles of probes, Chronus runs an optional pre-processing routine to initialize the child's disk state. A typical pre-processing routine would modify the child's `/etc/rc` file to run a given probe command on system boot.

Figure 3 describes the control flow for internal and external probes. The primary difference is that for internal probes, we destroy the child VM before extracting the probe result to avoid concurrent access to the `TTDisk`. External probes must interact with the live child VM, and therefore the order-of-operations is reversed.

3 Experience

We now describe some of our experiences with the Chronus tool, to give intuition for how the tool works and to demonstrate that Chronus can diagnose simple configuration errors. We emphasize that our evaluation to date is preliminary, and that work is ongoing to increase the scope and realism of our analysis. For these tests, both the parent and child VMs ran the NetBSD operating system, version 1.6.1.

```
#!/bin/sh

TEMPFILE=./QXB50.tmp
rm -f ${TEMPFILE}

ssh root@10.19.13.17 'date' > ${TEMPFILE}

if (test -s ${TEMPFILE})
    then echo "SSHD UP"
else echo "SSHD DOWN"
fi

exit 0
```

Figure 4: **A Chronus Probe Routine:** This is the complete version of a shell script that diagnosed a configuration fault in the ssh daemon.

To create an evaluation workload, we wrote a program called the *etc-smasher*, which simulates making typos in critical system configuration files. Once per second, *etc-smasher* chooses a random file from the /etc directory, which contains system-wide configuration files and application-specific configuration options. For 90% of the tests, the smasher writes back the file without modifying it, creating "background noise" for the system. For the remaining 10%, *etc-smasher* changes the file in a small way, by either removing, adding, or modifying a character.

The first two runs of this program produced the following configuration errors:

Configuration Fault #1: sshd. The child VM's ssh daemon has stopped responding. This prevents a user without terminal access from even attempting a problem diagnosis.

Configuration Fault #2: boot failure. The child VM does not boot correctly. Instead of a login prompt, the user is asked to enter a shell name.

For the sshd fault, we wrote a simple probe that attempts to login via ssh. This probe (shown in Figure 4) is an external probe: it runs on the parent VM. This probe script is simple, and it only deals with the observable behavior of ssh, not with potential causes of sshd's failure.

Figure 5 shows the Chronus output for the sshd fault. Comments (preceded by '#') have been added for clarity. In the first phase, the analysis engine localizes the failure to time step 4920. We then mount the disk at time steps 4919 and 4920, and use a recursive diff to compare the two file systems. In this case, the error resulted from corruption to the file ssh_host_key, which contains the child's private key.

The boot fault required an internal probe, whose functionality is split across two shell scripts. The initialization script modifies the child's boot script to run a command at the end of the boot process. The post-processing script extracts the output of this command from a file in the child's file system. The probe scripts are omitted for space, but they are of comparable complexity to the sshd script shown

```
# binary search phase
% ttsearch netbsd andrew.time

0000: SSHD UP  5267: SSHD DOWN  2633: SSHD UP
3950: SSHD UP  4608: SSHD UP    4937: SSHD DOWN
4772: SSHD UP  4854: SSHD UP    4895: SSHD UP
4916: SSHD UP  4926: SSHD DOWN  4921: SSHD DOWN
4918: SSHD UP  4919: SSHD UP    4920: SSHD DOWN

# attach ttdisk before and after fault
% attach2 andrew.time 4919 4920

# use recursive diff to find what changed
% diff -r --exclude '*dev*' /child1 /child2
Binary file /etc/ssh/ssh_host_key differs
```

Figure 5: **Diagnosing sshd Failure:** This execution correctly identified the fault point after disk write 4919.

above. Figure 6 shows that Chronus was able to localize this fault to a change in the bootconf.sh.

At present, Chronus requires roughly 10 seconds to reconstruct a disk from the past, boot a NetBSD VM, and execute a probe. The sshd failure diagnosis took roughly 2.5 minutes. Much of this time is spent busy waiting, because μDenali does not currently provide a reliable mechanism for the child to signal to a parent that it has finished running a probe. With the addition of this functionality, we should be able to decrease runtime by a factor of 5.

4 Discussion

Chronus relies on user-supplied software probes to characterize the system's correctness. We envision two scenarios for probe authorship. First, an expert user or administrator can create a probe on the fly in response to specific error conditions. An alternate approach is for software vendors to include a set of default probes with their software packages. These probes could be derived from development-time regression tests that already exists. This latter scenario is more applicable for unmanaged machines in a home environment.

Another set of issues relate to inconsistencies that may arise during the search process. One potential problem is that booting from a disk that was not properly shut down could generate spurious errors unrelated to the problem under consideration. This can happen because the file system lacks a transaction mechanism for robustly applying state changes. For example, there is no way to atomically rename multiple files. In the worst case, Chronus could be led down an incorrect path because it has detected a false configuration error. Another potential problem source is non-deterministic errors, which may prevent finding the failure transition point with just a single run of the analysis engine. It may prove possible to address these sources of error by running the analysis engine multiple times and using probabilistic analysis.

In some cases, the information provided by Chronus may be too fine-grained to be useful.

```
# binary search phase
% ttsearch netbsd andrew2.time

0000: SUCCESS  1607: FAILURE  0803: SUCCESS
1205: SUCCESS  1406: SUCCESS  1506: FAILURE
1456: FAILURE  1431: FAILURE  1418: FAILURE
1412: FAILURE  1409: FAILURE  1407: SUCCESS
1408: FAILURE

# attach ttdisk before and after fault
% attach2 andrew2.time 1407 1408

# use recursive diff to find out what changed
% diff -r --exclude '*dev*' /child1 /child2

file: /child1/etc/rc.d/bootconf.sh differs
<       conf=${_DUMMY}
>       conf=${$DUMMY}
```

Figure 6: **Diagnosing Boot Failure:** This execution correctly identified the fault point after disk write 1407.

Chronus tends to implicate microscopic events (e.g., a change to a specific file) rather than macroscopic events (e.g., the installation of a particular software package). The Backtracker tool [6] may prove useful at bridging the gap from low-level state events to high-level user actions. Integrating Chronus and Backtracker is an area for future work.

A fundamental limitation of Chronus is that it cannot diagnose problems that involve external factors such as network failures. In some cases, however, Chronus can be helpful by narrowing down the search space. For example, a network outage can be caused by hardware failure (a fault outside the system) or by an incorrectly specified subnet mask (a fault inside the system). By ruling out internal faults, Chronus can allow human administrators to make better use of their time.

5 Related Work

The state of the art for dealing with change-induced failures is to rollback the system to a known good state [7], possibly applying application-level state replay to avoid losing work [3]. The limitation of such approaches is they require the user to know when the fault was introduced in order to choose an appropriate state snapshot. This is difficult on systems where configuration changes can be introduced by multiple users or by system daemons like automatic software update. Additionally, rollback systems provide little insight as to why a particular action caused a failure — for example, a user may learn only that Service Pack 1 caused the failure. Chronus can shed light on the source of failures by essentially replaying the fault in slow motion.

A different approach to problem diagnosis is to construct software agents that embody the knowledge of a human expert [2]. The limitation of such systems is that they are only as good as their initial problem diagnosis heuristics. Complex systems generate unexpected errors. Chronus can capture these

errors by operating beneath the layer of operating system and application semantics.

Our vision is similar to the no-futz agenda of Margo Seltzer's group [5]. This group advocates rethinking the design and layout of system configuration state to reduce the chance of unintended side-effects. Although this is a worthy design goal, the tight integration of today's application and system functionality suggests this approach won't solve all configuration problems. Also, Chronus provides benefit to systems as they currently exist, without requiring potentially disruptive changes to the mechanisms used for storing system configuration state.

Recently, Redstone et al. proposed a model of collaborative debugging [9]. This approach extracts relevant problem symptoms to serve as a query against a database of known problems. A key challenge for such a system is constructing a database and query engine that give meaningful results. Chronus avoids using databases by directly "querying" the system state at a previous instant in time. The results returned by our system will be more relevant because they pertain exclusively to the system under consideration.

Delta-debugging [16] applies search techniques to the problem of localizing source code edits that induced a failure. Delta-debugging does not assume changes are ordered, and much of the system's complexity derives from having to prune an exponentially large search space. The challenges for Chronus relate to capturing and replaying complete system states using time-travel disks and virtual machines.

Perhaps the closest system to Chronus is Strider [14], which automatically finds configuration errors in the Windows Registry. Unlike Chronus, Strider is targeted at a specific configuration database, and it relies on Registry-specific knowledge to prune the search space. By capturing raw disk blocks, Chronus can diagnose errors for arbitrary applications and OS's — even for software systems that haven't been written yet. Another difference is that Chronus leverages virtual machine monitor technology for running time-travel probes. VMMs enable the detection of low-level errors that arise during system boot, and provide the ability to isolate and discard changes made during analysis.

6 Conclusions and Future Work

Our goal in this work is to move some of the burden for diagnosing computer problems from humans to machines. Our approach is based on the combination of two emerging technology trends. First, large disks make it possible to log all storage activity over extended durations. Second, virtual machine monitor technology makes it safe and fast to test a large number of prior system configurations. We have constructed a problem diagnosis tool called Chronus, and demonstrated that it can accurately diagnose simple system configuration errors.

We are building on the work described in this paper by exposing Chronus to a larger and more realistic battery of tests. For example, we are using Chronus to diagnose errors that arise during the configuration of a web server with database-driven content. We are also performing quantitative benchmarks that analyze overhead during normal operation and the fault diagnosis time.

Finally, we are exploring applications of Chronus to the problem of self-tuning systems. Previous work in this area has operated"in-the-small" by tuning a small number of parameters — for example, TCP socket buffer sizes [12] or the Lotus Notes admission control threshold [8]. We believe it is possible to analyze significantly larger configuration problems. Virtual machine monitors can give us leverage in two ways: (1) considering configuration choices with delayed effects, such as applying software patches or kernel compile options; and (2) considering potentially unsafe configuration choices that could render the system unusable or insecure. We anticipate that the key challenge in this application will be finding mechanisms to specify paths through the configuration space, and for pruning down the space of configuration choices in the presence of noisy or nondeterministic processes.

References

[1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

[2] G. Banga. Auto-diagnosis of field problems in an appliance operating system. In *Proc. of the USENIX Annual Technical Conference*, June 2000.

[3] A.A. Brown and D.A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proc. USENIX Annual Technical Conference*, June 2003.

[4] R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), 1981.

[5] D.A. Holland, W. Josephson, K. Magoutis, M. Seltzer, C.A. Stein, and A. Lim. Research Issues in No-Futz Computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[6] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, October 2003.

[7] Microsoft, Inc. Windows XP system restore. `http://msdn.microsoft.com/library/default.asp?URL=/library/techart/wind%owsxpsystemrestore.htm`, April 2001.

[8] S. Parekh, N. Gandhi, J.L. Hellerstein, D.M. Tilbury, T. S. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Systems*, 23(2):127–141, 2002.

[9] Joshua A. Redstone, Michael M. Swift, and Brian N. Bershad. Using Computers to Diagnose Computer Problems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, 2003.

[10] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[11] D.S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, December 1999.

[12] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. In *Proceedings of the ACM SIGCOMM*, 1988.

[13] VMware, Inc. VMware virtual machine technology. `http://www.vmware.com/`.

[14] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H.J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the USENIX LISA Conference*, October 2003.

[15] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing Services with Interposable Virtual Hardware. In *Proceedings of the First Symposium on Network Systems Design and Implementation*, March 2004.

[16] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference*, September 1999.