# AutoBash: Improving configuration management with operating system causality analysis

Ya-Yunn Su
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI
yysu@eecs.umich.edu

Mona Attariyan
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI
monattar@eecs.umich.edu

Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI
jflinn@eecs.umich.edu

## ABSTRACT

*AutoBash is a set of interactive tools that helps users and system administrators manage configurations. AutoBash leverages causal tracking support implemented within our modified Linux kernel to understand the inputs (causal dependencies) and outputs (causal effects) of configuration actions. It uses OS-level speculative execution to try possible actions, examine their effects, and roll them back when necessary. AutoBash automates many of the tedious parts of trying to fix a misconfiguration, including searching through possible solutions, testing whether a particular solution fixes a problem, and undoing changes to persistent and transient state when a solution fails. Our results show that AutoBash correctly identifies the solution to several CVS, gcc cross-compiler, and Apache configuration errors. We also show that causal analysis reduces AutoBash's search time by an average of 35% and solution verification time by an average of 70%.*

## General Terms

Management, Reliability

## Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software configuration management*; D.4.7 [**Operating Systems**]: Organization and Design

## Keywords

Configuration management, causality, speculative execution

## 1. INTRODUCTION

Users spend too much time configuring their computers [6]. Configuration management is often an isolated process, in which each user discovers on her own the particular incantations that are required to transform her particular computer system from a misconfigured state to a correct one. Configuration management can be especially frustrating because, in the process of trying to reach a more desirable system state, the user may take some action that leaves the system in a state worse than the one that existed before.

While autonomic systems that can configure themselves are an admirable goal [10], experience to date suggests that it is very hard to completely hide the complexity of modern computer systems. Thus, a less ambitious goal may be in order. Rather than try to eliminate configuration entirely, we adopt the more pragmatic approach of providing users and system administrators with a set of interactive tools, which we call AutoBash, that helps them manage configurations.

For instance, consider the actions of a typical user faced with a configuration problem. She might search the Web and query colleagues to find others who have encountered and solved a similar problem. She might then try the most likely solution and test her system to see if the problem has been fixed. If the system still does not operate correctly, she will carefully try to undo any changes that she has made and then try the next possible solution. AutoBash can help this user by automating many of the most tedious and frustrating steps in the above process. It uses causal dependency tracking and analysis implemented within the operating system kernel to speed the search for solutions to configuration problems, and it uses speculative execution to transparently apply and test possible solutions while retaining the ability to undo incorrect actions.

AutoBash has three modes: observation, replay, and health monitoring. In all three modes, AutoBash expresses system health as the results of executing a set of predicates. In observation mode, AutoBash records the actions of users as they adjust their systems to fix a particular problem. AutoBash logs input and output as the user probes the system, makes state changes, and tests the new state. In replay mode, AutoBash tries to fix the same problem on different systems by applying actions that fixed the problem previously. AutoBash lets users learn from each others' experiences: as AutoBash sees more correct solutions to a problem, it has a greater database of potential solutions to draw upon. In health monitoring mode, AutoBash periodically tests the correctness of a computer system in the background to diagnose configuration bugs before they become critical.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SOSP'07,* October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010 ...$5.00.

Configuration management is a thorny problem that requires contributions from many disciplines, including machine learning, user interface design, and distributed information systems. This paper focuses on how the operating system can contribute to a complete solution by tracking and analyzing causal dependencies as they propagate between processes, files, and other entities. AutoBash leverages causality support within the Linux kernel to understand the outputs (causal effects) and inputs (causal dependencies) of executing configuration actions. This leads to the following benefits:

- Causal tracking reduces the amount of testing that must be performed to determine whether a particular action fixed a problem or caused a new problem. When AutoBash performs a configuration action, it tracks the set of entities modified as a result of the action. If none of the entities serves as an input to a predicate, then the evaluation of that predicate will not change as a result of the configuration action. Thus, the predicate need not be retested.
- Causal tracking provides a compact representation of interactive user activity. When a user interacts with an editor or GUI application, AutoBash could potentially record their actions by logging inputs (keystrokes and mouse events) or outputs (system calls). Unfortunately, both approaches often produce a large amount of data that is difficult to understand or replay deterministically. In contrast, AutoBash uses causal tracking to represent interactive activity as a minimal set of changes made to persistent state during the activity.
- Causal tracking helps explain to users how a problem was fixed. When a user fixes the problem interactively, AutoBash shows him which actions possibly contributed to the solution and which did not contribute. Further, it shows the changes to intermediate entities (e.g., diffs of configuration files) that possibly led to the solution. When AutoBash autonomously searches for a solution to a configuration problem, it shows a similar report to the user before committing any solution it finds. This allows the user to understand and confirm any change to configuration state suggested by AutoBash.
- AutoBash transparently recovers from incorrect actions by tracking all causal effects of each action. We use Nightingale *et al.*'s Speculator [15] to roll back all effects of incorrect actions. While tools exist for checkpoint and rollback of persistent registry or file system state, such systems are incomplete unless they also roll back transient state such as process memory. For instance, a corrupted application can cause a problem to remain until the system is rebooted. Even worse, a corrupted application may subsequently write incorrect data to the file system, making the effects of an incorrect action durable.
- Causal tracking, when combined with speculative execution, provides isolation for configuration activities. If a configuration action is found to be incorrect and is rolled back, AutoBash guarantees that any other activity that observed the incorrect state will also be rolled back. This strategy provides optimistic concurrency for speculative configuration actions — an approach that lets AutoBash perform potentially time-consuming configuration actions in the background while the user continues to use the computer system for normal activities.

Our current AutoBash prototype assists in solving configuration bugs that are confined to a single computer system, such as a home computer, personal workstation, or stand-alone server. Our results show that AutoBash dramatically decreases the amount of user interaction required to fix bugs in CVS, the gcc cross-compiler, and the Apache HTTP server by automating many of the most tedious activities. Our results also show that operating system causality analysis decreases the amount of time needed by AutoBash to automatically find solutions by an average of 35% and the time spent on predicate testing by an average of 70%.

## 2. SPECULATOR BACKGROUND

Speculator is a system within the kernel that supports process-level speculative execution through causal dependency tracking and lightweight checkpoint and rollback. A process invokes Speculator to checkpoint its state and continues execution. Later, if the speculation is found to be incorrect, Speculator rolls back the process execution and the process is restarted using the checkpoint state. If the speculation is found to be correct, Speculator discards the checkpoint, marks the process as non-speculative, and continues the process execution. Speculator supports speculative execution throughout the operating system, tracking causal dependencies as a speculative process interacts with the kernel, file system, and other processes. Thus, a speculative process can safely change system state; all modifications can later be undone should the speculation prove wrong.

Speculator ensures that speculative state is never *externalized*, i.e. visible to a user (terminal output) or any external device (network, disk, etc.). If a speculative process attempts to externalize state, Speculator buffers its output in the kernel until the outcome of the speculation it depends on is decided. If a speculative process performs a system call that Speculator cannot handle by either propagating causal dependencies or buffering output, Speculator blocks the process until it becomes non-speculative.

## 3. DESIGN AND IMPLEMENTATION

### 3.1 Overview

The goal of AutoBash is to help users find solutions to the configuration problems they are facing. AutoBash divides configuration activities into *actions* that modify system state and *predicates* that test system correctness. We define a *solution* to be a sequence of actions that transforms a system from an incorrect state, in which one or more predicates evaluate to false, to a correct state, in which all predicates evaluate to true.

Predicates are guaranteed to have no side effects; that is, they are not allowed to visibly modify system state such as files and processes. AutoBash ensures the absence of side effects by speculatively executing each predicate and rolling it back on completion. Predicate execution, return codes, and causal inputs are visible to AutoBash but not to any other process running on the system. (More precisely, any entity that observes predicate execution is rolled back to its state prior to the observation once the predicate completes, and external effects are not visible to the user or other computers.) There are several ways we can obtain predicates. Predicates are similar to test cases and may be provided as part of the software (similar to Windows configuration wizards or test suites for program testing). Alternatively, they may be created by a community of users. Users can also add predicates that demonstrate symptoms that they would like to fix. Since a predicate is simply a Unix process, the normal actions that users take to verify whether a solution is working (e.g., loading a Web server test page) can be transformed into predicates with minimal effort. A predicate can be written as a shell script or can be a binary executable with a return value indicating true or false.

AutoBash maintains a predicate database that contains the tests associated with all applications under its purview. It organizes predicates with the following schema: name, application, predicate file location, last execution result, the time it was last executed, input record file location, and average execution time. The *application* field refers to the application a predicate is designed to test. *Input record file location* is the location of the file containing the inputs a predicate is dependent upon. AutoBash stores the *average execution time* so that it can warn a user if predicate execution takes much longer than usual. Based on our evaluation of three applications in Section 5, five to ten predicates seem to cover most of the configuration bugs we encountered. In our future work, we would like to deploy AutoBash in a real environment to gain more experience on what the predicate space would look like.

AutoBash has three modes of operation: observation, replay, and health monitoring. In its observation mode, AutoBash helps a user manually resolve a configuration problem. The AutoBash shell is a version of the standard Linux `bash` shell that we have modified to support speculative execution and causality tracking. Each command given to the shell is considered an action that may contribute to an eventual solution. AutoBash executes each action speculatively, which allows its user to roll back an action's effects if the action later proves to be incorrect. AutoBash automatically tests for system correctness by executing predicates after each action completes. It also tracks causality to explain how a problem was fixed and what actions are related once a solution is found. The solutions found during observation are canonicalized and saved so that they can be used later to fix similar configuration bugs.

In its replay mode, AutoBash automatically searches for solutions to a configuration problem. Its search space is the set of solutions that were previously found for similar problems. Similar to predicates, potential solutions can come from many sources: the user's prior experience, solutions developed by vendors or a community of users. While AutoBash does not currently provide a distributed system for locating potential solutions, other projects, such as the Friends Troubleshooting Network [7], have shared configuration information through peer-to-peer networks. These projects have also addressed security and privacy concerns related to sharing configuration information. AutoBash stores solutions and their metadata in a local repository similar to how it stores predicates. For each solution, AutoBash keeps track of the solution location and the application the solution is associated with.

AutoBash speculatively executes a solution followed by predicate testing. If any predicate evaluates to false, AutoBash decides that the solution does not fix the configuration bug and rolls back that solution. AutoBash then tries the next solution in its solution database until all predicates evaluate to true. Speculative execution isolates AutoBash's replay activity from other non-configuration tasks — a user may continue to use her computer while AutoBash searches for the solution in the background without worrying that results will be corrupted by observing state that is in the process of being reconfigured. After AutoBash finds a solution, it explains the actions it took, their causal effects, and how they relate to the configuration predicates. Its user can examine this information before committing the configuration changes; if the user disagrees with AutoBash's solution, AutoBash will roll back the candidate solution and continue searching for another fix.

In its health monitoring mode, AutoBash periodically tests all predicates in its predicate database. If any predicate fails the check, AutoBash can be set to enter replay mode to find and suggest a potential solution to the user.

When a solution is found but before it is committed, AutoBash verifies that all predicates that previously succeeded still do; this checks for "solutions" that fix one problem but cause another. AutoBash tracks causality to record and store the system state observed by each predicate. Thus, if a solution does not affect the state observed by a predicate, AutoBash recognizes the predicate need not be run. This limits the number of predicates that must be tested for each solution.

## 3.2 Usage scenario

This section describes a sample scenario in which we used AutoBash to solve a configuration problem. Our research team set up a CVS repository to store our source code and publications. We created a CVS group that owned the repository and added all team members to the CVS group. When we used the CVS repository, we found that other team members would get a "permission denied" error when they attempted to check out any project they had not checked in.

We manually fixed this problem by first analyzing the error message and examining the current system state. We then tried various actions to modify the system settings. After each action, we tried to check out the project to test if the modification solved the problem. After a few trials, we realized that the file permission error was due to an incorrect file ownership – the default group for a user is his original user group and hence any files added to the repository would be owned by the user and inaccessible to other users despite the repository being accessible to all. We finally applied the correct action: `chmod` the bit that sets group id on execution. This action causes CVS to set the group of a newly added file or directory to the same as the parent directory instead of the current process.

There are several disadvantages to this approach. First, any system state modification might adversely affect CVS or other applications running on the machine. If the modification is incorrect, we have to manually undo our changes — this could be difficult as we often do not know the effects of changing system state. Further, if a system change solves the problem, we do not know if the solution breaks other features of the application, or even other applications running on the system. Finally, often times after we have solved a configuration problem, we still do not know what actions we took contributed to the solution.

As an alternative, we can use AutoBash to diagnose and solve this problem. If CVS is packaged with a set of standard solutions and predicates, we can run AutoBash in replay mode to search for a solution. AutoBash first runs all predicates from the standard predicate sets to determine which predicates exemplify the buggy scenario. In the scenario above, any predicate that involves checking in a project as a user and checking out as a different user is sufficient. AutoBash then goes through each solution from the standard solution sets and executes it speculatively. After executing a potential solution, AutoBash first tests predicates that demonstrate the problem. If any such predicate fails, AutoBash rolls back the solution and tries the next one. If a solution causes all those predicates to succeed, AutoBash next tests the remaining predicates for CVS and other applications. If a solution exists, AutoBash can find it and ensure that the solution does not break other existing configuration of CVS and other applications. AutoBash replay mode is a useful tool for users who do not participate much in system administration work.

However, if CVS does not have standard predicate and solution sets, the user can run AutoBash in observation mode to manually fix the configuration problem. First, the user needs to specify one or more predicates that expose faulty application behavior. The user can also write predicates to test basic functionality for CVS and other applications. Next, the user tries different actions in the AutoBash shell; AutoBash executes each action speculatively and tests predicates after each action to determine if the action and prior actions fix the problem. If all user-specified predicates succeed, AutoBash tests against the remaining predicates to check that the fix does not break the existing configuration for CVS and other applications. While running AutoBash in observation mode, the user can roll back any prior action. If subsequent actions observe the effects of a prior action, AutoBash informs the user of this dependency and rolls back those actions. Through this mechanism, AutoBash in observation mode can be a useful tool for system administrators to try out different actions safely. After the user fixes a problem, AutoBash also analyzes causal information tracked by the kernel to explain which actions contribute to solving the problem and how they relate to each other.

## 3.3 Tracking causality

In designing AutoBash, we considered several possible methods of tracking causality. Our goal was to capture sufficient information to allow us to reason about the causal effects of configuration actions and how they relate to predicates that test system health, while also minimizing performance impact on foreground applications and keeping the amount of causal state manageable. We want to track causal effects of actions and causal dependencies of predicates so that we only re-run those predicates affected by an action, reducing the time to find a solution.

We define causality in the operating system kernel in the following way. Processes, files, directory entries, sockets, pipes, and signals are all considered first-class entities. When a process interacts with other entities by executing system calls, we use Speculator to track the causal relationships among the entities. Specifically, if a process observes another entity (e.g., reads a file or receives a signal), we say that the process becomes causally *dependent on* the observed entity. If a process modifies another entity, the modified entity becomes causally dependent on the modifying process. If a process observes an entity while modifying it (for instance, when a process writes to a file, it checks access permissions of a file before writing to it and therefore becomes dependent on the file), the process and the modified entity become mutually dependent.

We track causality for predicates and solutions at the point we start executing them because we believe that most of the relevant interactions usually happen within the execution interval. We only track causal effects and dependencies for entities that are affected by the initial predicate or solution process. The process being tracked marks the start and end of each tracking interval by making `ioctls` on a pseudo-device. The first `ioctl` causes the OS to begin tracking causality. Subsequently, the set of recorded inputs or outputs can be queried by issuing another `ioctl`. Tracking is terminated by issuing a final `ioctl` that releases the data structures used by the OS to maintain causal information. This interface allows user-level AutoBash tools to ask specific questions about the causal events of processes that they are tracking, while minimally affecting the performance of other processes that are not being tracked. In this manner, we can obtain the most relevant causal interactions, and the overhead of causality tracking is only paid during configuration management and maintenance.

### *3.3.1 Tracking output sets*

We leveraged Speculator to track causal effects. When a user-level process asks that its causal outputs be tracked, Speculator assigns a unique id to the request and creates an *output set* that contains pointers to entities that depend on the subsequent execution of the calling process. Initially, the only member of the output set is the process that made the `ioctl`. As the process interacts with other entities by executing system calls, we add to the output set new entities that come to depend on an entity already in the output set (using the definition of "depends on" from Section 3.3). For instance, when a process that is a member of the output set modifies a file, that file is added to the output set. If another process reads the file, the reading process is also added to the output set.

The implementation of output sets is minimal; each entity has a list of pointers to the output sets on which it currently depends. Each output set created by Speculator has a list of reverse pointers to the entities it contains. Thus, adding a new entity to an output set requires only that Speculator add a pointer to two lists. Speculator can track multiple output sets simultaneously to track dependencies for different ranges within the same process or ranges that occur in different processes.

### *3.3.2 Tracking input sets*

A user-level process may also ask that its causal inputs be tracked. We have modified Speculator to create an *input set* that contains all processes, files (both content and metadata) and directory entries that the tracked process comes to depend on during the execution of subsequent system calls. For instance, if a process receives a signal from another process, the sending process is added to its input set. Similarly, if it reads a file, the file and the directory entries used to look up the file within the file system are added to its input set. We handle IPC entities (signals and sockets) like we handle other entities (processes and files), but IPC entities

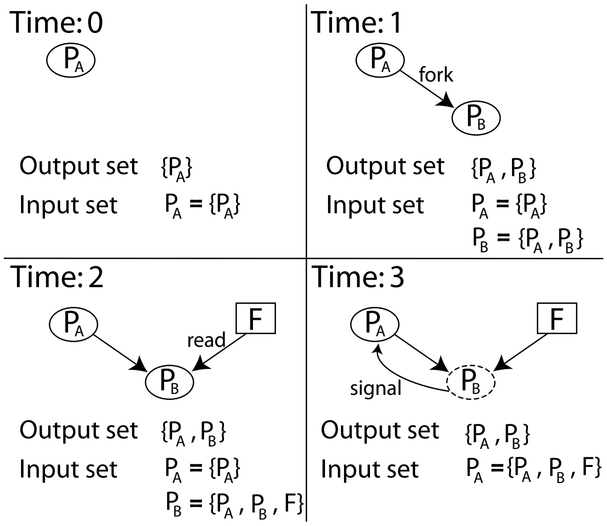


**Figure 1: An input sets tracking example**

are not reported in the final input sets because they are not persistent entities within a computer system.

As defined so far, an input set contains only the causal inputs observed by a single process. However, user-level tasks often do not map directly to individual processes; for instance, a shell may fork a child to perform a task, which in turn will fork other children or interact with other processes via IPC to perform subtasks. In such situations, the input set should capture all the causal inputs of the *collection* of processes that are cooperating to perform the task.

We calculate the input set for cooperating processes as follows. First, any member of the output set of the tracked process is considered to potentially be cooperating to perform a high-level user task — we create an individual input set for each such entity. When an entity in the output set comes to depend on another entity in the output set, its input set becomes the union of the input sets of the two entities. If an entity that is not a member of the output set comes to depend on an entity that is a member, it joins the output set and its input set becomes equal to that of the entity on which it depends.

When a query is issued, Speculator returns the individual input set of the calling process. The set returned consists of not only the entities directly observed by the querying process but also some entities observed by other processes. For instance, if the tracked process forks a child process, the entities observed by the child become part of the parent's input set when the child exits and the parent receives its termination signal. On the other hand, if the child does not interact further with its parent (e.g., it might be a command executed in the background), the entities it observed are not part of its parent's input set; this reflects the intuition that the parent cannot depend on the child if it does not observe its execution in any way. Similarly, if a tracked process makes an RPC to a server, only the entities observed by the server after it receives the request but before it replies to the tracked process will be returned as part of the input set. A pipe, socket, or localhost packet transfers input sets between the RPC client and server but is not part of the input sets.

Figure 1 is an example of how we calculate input sets for cooperating processes. Processes are shown in ellipses and files are shown in rectangles.

- At time 0, process $P_A$ requests that its input and output sets be tracked. Both $P_A$'s output and input sets contain only the process itself.
- At time 1, $P_A$ forks a child process $P_B$. Since $P_B$ is dependent on $P_A$, $P_B$ is added to the output set and its input set is the union of $P_A$'s input set and itself.
- At time 2, $P_B$ reads file F and becomes dependent on F so F is added to $P_B$'s input set. Since F is not affected by any object in the output set, it is not added to the output set.
- At time 3, $P_B$ exits and a signal is sent to its parent $P_A$; therefore, $P_A$ is now dependent on $P_B$. Since $P_B$ is already in the output set, $P_A$'s input set becomes the union of $P_A$'s and $P_B$'s input sets.

Of course, this definition of the input set is a heuristic — a precise characterization of the input set would require a semantic knowledge of application behavior that seems quite difficult to achieve in an operating system kernel. Nevertheless, our results show that this heuristic performs extremely well in practice for the configuration management tasks in which we have employed it.

## 3.4 Observing user actions

In its observation mode, AutoBash assists its user in manually fixing a problem. The user begins configuration management by launching the AutoBash shell. Next, the user specifies which predicates are related to the problem he is attempting to fix. These can be drawn from AutoBash's predicate database by specifying the application that is being configured. Alternatively, the user may use the AutoBash shell to specify new predicates on the fly; this can be useful when the user is attempting to fix a rare problem that he has never previously encountered. AutoBash verifies that at least one of the specified predicates evaluates to false; if all evaluate to true, the user must specify at least one more predicate that exemplifies the problem he is trying to solve.

The user then enters configuration actions using the AutoBash shell. The shell records the command line input for each action. For simple actions, the command line information is all that is needed to capture the action; for instance, the `adduser` and `chown` utilities can be precisely characterized by the command line inputs. However, for actions that start interactive applications, such as a text editor, the command line information is woefully inadequate to characterize what the user is doing. For such interactive applications, AutoBash extracts state deltas as described in Section 3.5.

After each action completes, AutoBash automatically tests to see if the prior actions have corrected the configuration problem. It first runs the predicates initially specified by the user. AutoBash executes each predicate speculatively by forking a child process and invoking Speculator to make the child speculative. After the child exits, the AutoBash shell reads its return code and input set, then asks Speculator to roll back the child's speculation.

If the return codes of the initially specified predicates indicate success, AutoBash next runs the remaining predicates in its database. This checks for solutions that fix one configuration problem but cause another. The initially specified predicates are run first to reduce testing time; since these predicates exemplify the configuration problem the user is

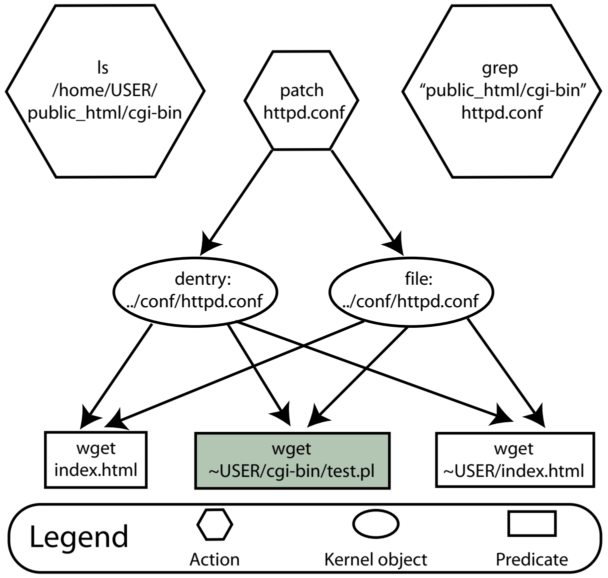


**Figure 2: Sample causal explanation**

trying to fix, they are most likely to fail and eliminate the need to test the remaining predicates.

Since executing predicates is time consuming, AutoBash analyzes causality to limit the number of predicates that are tested. During predicate execution, AutoBash records the predicate's input set. It also records the output set of each action. If an action's output set does not intersect the input set of a predicate, AutoBash does not re-run the predicate because a predicate that depends on the same input set must return the same answer as from the last execution.

Each action is run as a separate speculative execution. AutoBash forks a child process to perform the action, then asks Speculator to execute the child speculatively and track its output set. When the child exits, Speculator provides AutoBash with the output sets of which the child is a member. If the child is in a prior action's output set, it has come to depend on that action by observing some entity that depends on the prior action. From this information, AutoBash determines which actions depend on which prior actions.

Using the AutoBash shell, the user can roll back any prior action. While actions can be rolled back individually, the rollback of an earlier action requires that subsequent actions that depend on that action also be rolled back (since they observed incorrect output of that action). AutoBash can use the action interdependency information it collects to inform the user which actions will be rolled back. Alternatively, a user may simply choose to roll back all actions that occurred after a specified action.

We have often encountered users who have fixed a particularly nasty configuration problem but are not quite sure of what actions they performed that contributed to the solution. To help such users, AutoBash can provide a causal explanation once a solution is found. Figure 2 shows an example explanation produced by AutoBash after an Apache configuration bug was fixed. The symptom of the misconfiguration was that a user can `wget` her own home page but cannot `wget` the results of a CGI script in her home directory. The problem was caused by Apache's configuration file not allowing users to execute CGI, and the fix was to add the appropriate line to the configuration file. The explanation in Figure 2 lists actions (hexagons) that were performed and not rolled back. Each action might affect kernel objects (ellipses) that causally depend (shown by directed arrows) on it. Finally, the explanation lists predicates (rectangles) used to verify that the problem is solved. In Figure 2, the three predicates, from left to right, are `wget` the default home page, `wget` the result of the CGI script in the user's directory and `wget` the user's home page. The grayed predicate evaluates to false before applying any action, while the white predicates evaluate to true. By examining the explanation, we can infer that certain actions (`ls` and `grep`) modify no kernel objects that these predicates depend on; from this, we can conclude that they do not contribute to the solution. Since `patch` is the only action that affects kernel objects that all predicates depend on, we can also conclude that patching httpd.conf is what fixed the bug.

In addition to the graphical output, AutoBash also produces a textual explanation that includes file differences and other more detailed information. After finding a solution, the user can view the explanation before deciding to commit his changes and exit the AutoBash shell.

When a solution is found, AutoBash saves it in a form that can later be used by the replay tool. The saved version contains only those actions that were executed and not later rolled back. Like PeerPressure [18], AutoBash canonicalizes solutions to replace specific identifiers such as userids, home directories, and IP addresses with generic variables.

By default, Speculator does not allow any output that depends on speculative actions to be externalized. While theoretically correct, this policy does not work well with an interactive debugging tool such as AutoBash. Therefore, we modified Speculator to allow speculative output to the screen when the user runs AutoBash in observation mode. However, predicate testing normally does not externalize any output to the screen. We only use this capability on predicates when we are testing and developing them.

## 3.5 Extracting state deltas

For non-interactive applications, the command line information recorded by AutoBash is sufficient to describe how the action can be replayed later to fix a similar problem. However, for interactive applications such as editors and GUI configuration tools, AutoBash must capture how the tool is used in order to perform similarly during replay.

Potentially, AutoBash could record all inputs to an interactive application; e.g., it could record key strokes, mouse movements, and button presses for a local application, or network packets for a distributed one. Such an approach could lead to AutoBash recording a large amount of information. More problematically, such low-level inputs are not easily understandable by a user. Further, our previous work [17] showed that it can be quite hard to deterministically replay graphical input since events such as button presses must be timed to occur after widgets such as dropdown menus and screen windows appear.

An alternative approach would be to record *outputs*, such as the system calls made by the interactive application. During replay, the recorded system calls could be reissued to try to duplicate the observed behavior of the interactive application. Like the technique of recording user inputs, this approach can generate a large amount of data that is difficult to understand. Further work would be required to abstract away non-deterministic OS variables such as process identifiers and file handles.

Instead of recording outputs directly, AutoBash captures a *state delta*, which is the difference in the state of the system caused by the execution of the interactive application. When the AutoBash shell receives the signal from an exiting child process executing a configuration action, it queries Speculator to determine if the process has received any input from an interactive source (e.g., from a console, network interface, or similar external device). If no such input has been received, the command line information is assumed to be sufficient to characterize the action. Otherwise, AutoBash generates a state delta.

First, AutoBash queries the output set for the action. This lists all entities that causally depend on the execution of the action. Next, AutoBash computes a diff for each entity. For files, it uses the standard `diff` tool to generate a patch file, and supplements the information with the changes made to the file attributes. For directory entries, it records entries added and deleted from the directory and the directory attributes. For processes, it records the sequence of causal inputs to the process generated by the user action. For example, it records signals sent to the process and data communicated via IPC mechanisms such as pipes.

AutoBash uses several optimizations to reduce the size of state deltas. First, it eliminates temporary entities that are created and destroyed during the action such as temporary files and processes that are forked and terminated. Next, it eliminates all events that precede a deletion. For example, not all input sent to a process that terminates is saved — only the final termination event is retained. Finally, events that cancel each other are eliminated; e.g., two `chown` operations that change a file's attributes to a new value and then back to the original value. This process is not unlike log optimization in the Coda file system [13], except that we apply such optimizations to OS kernel entities rather than just file system objects.

After optimization, the state delta gives a terse representation of the causal effects of an interactive action. For instance, using a GUI tool to change the settings of a server might produce a state delta that contains only a diff showing changes made to a configuration file and a signal sent to the server to cause it to re-read the configuration file.

## 3.6 Finding a good solution

AutoBash's replay mode searches through a database of potential solutions to find one that fixes a problem being encountered on a user's computer. Such solutions may come from peers who have encountered similar problems, from software developers who are supporting their product, or from the user previously fixing the same problem. Since solutions can be scripts or binaries, AutoBash handles a wide variety of replay inputs.

The AutoBash replay mode is designed to execute in the background; the user can still use her computer while AutoBash reconfigures. Speculative execution of predicates and potential solutions provides isolation: if any process observes the causal effects of executing a predicate or configuration action, that process is transparently rolled back to the point before the observation and re-executed. Since Speculator does not externalize speculative state, if a foreground application comes to depend on AutoBash's background task and tries to externalize output, Speculator buffers the foreground task's output until the outcome of the speculation is decided. Thus, the effects of speculative execution are not visible to non-AutoBash processes, the user, other computers, or any entity external to AutoBash and the kernel. Of course, speculative execution consumes resources on the computer and some work performed by other processes may need to be rolled back, so the performance of non-configuration tasks will be impacted by a background AutoBash execution. The performance impact will depend on how resource-intensive the background task is.

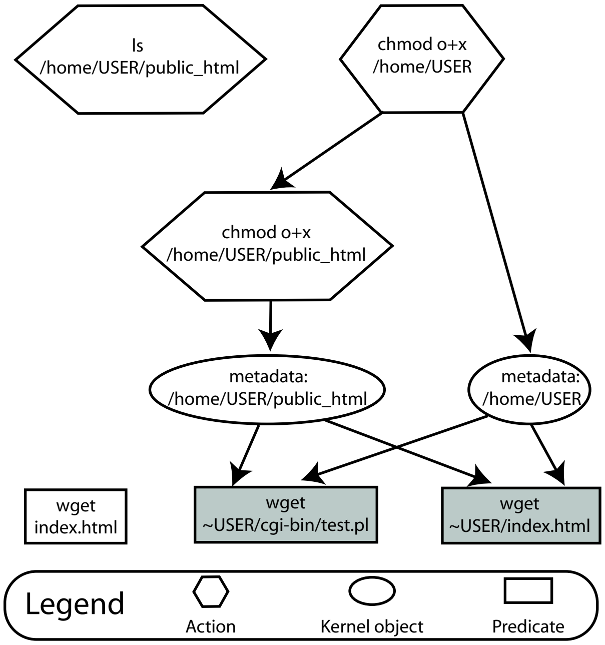
When using AutoBash in replay mode, the user first specifies the predicates that exhibit faulty application behavior. As described in Section 3.4, these predicates can be drawn from those used to test the application in the computer's existing predicate database, or they can be new predicates specified by the user. AutoBash first runs all specified predicates and records whether each succeeds or fails. Speculator is used to roll back each predicate after execution and to report the input set for each predicate. If no predicates fail, AutoBash asks the user to specify an additional predicate that exemplifies the problem being debugged. We call this process *initial predicate testing*.

AutoBash next iterates through the solutions in the solution database to find one that makes all specified predicates succeed. AutoBash speculatively executes a potential solution by forking a child process and invoking Speculator to make it speculative. It then waits for the child process and queries Speculator for the solution's output set. If the output set of the solution and the input set of a predicate do not intersect, AutoBash does not re-run the predicate. Assuming that the predicate depends on the same input set, its behavior will not change as a result of executing the solution. Thus, if there is no intersection between the output set of the solution and the input set of a failed predicate, AutoBash immediately considers the solution unsuccessful (it did not fix at least one predicate).

After applying a solution, AutoBash first tests the predicates that failed in the initial predicate testing, and then tests against the rest of the predicates in the predicate database. AutoBash only re-runs those predicates whose input sets intersect with the solution's output set. If all such predicates succeed, AutoBash declares the solution successful. If any predicate fails, AutoBash declares the solution a failure, rolls back that solution, and tries the next solution. AutoBash currently applies only one solution at a time in replay mode, though it could potentially execute combinations of solutions. We plan to explore the effectiveness of this idea and the impact of an increasing search space on AutoBash's scalability in the future.

Once a solution is found, AutoBash outputs the solution to the user as a potential fix. AutoBash provides a causal explanation similar to the one in Figure 2 that shows the actions executed and all predicates affected. After viewing this information, the user can confirm the fix, in which case AutoBash commits the speculation and exits, or decline the fix, in which case AutoBash rolls back the speculation and continues searching for a solution.

Figure 3 shows how AutoBash determines what predicates to run after applying a potential solution for an Apache configuration bug. The symptom of the misconfiguration is that a user cannot `wget` her own home page nor the result of a CGI script in her home directory. The bug was that



This figure shows how AutoBash leverages causality analysis to determine which predicates to run after applying a potential solution to an Apache configuration bug.

**Figure 3: Sample causality analysis**

the Apache HTTP server does not have search permission to access the user's home directory where the user's home page and CGI script are located. The action to fix this bug is to give search permission to others for this user's home directory. The grayed predicates failed in the initial predicate testing, while the predicate shown in white succeeded. Figure 3 illustrates that the action `chmod o+x /home/USER/public_html` is dependent on the action `chmod o+x /home/USER`. Also, both `chmod` actions have causal effects on the kernel objects which the grayed predicates depend on. From this causality analysis, AutoBash infers that it only needs to re-run the last two predicates.

We expect that many applications will have a set of standard predicates that can be used to validate application configuration. As part of initial predicate testing, AutoBash records whether each standard predicate succeeds or fails. The results of the initial predicate testing are aggregated across multiple runs and packaged with each solution.

AutoBash uses its knowledge of which standard predicates succeed and fail to order the solutions it tries. If the user does not specify any predicates to test and hence does not hint at which application might be failing, the first step in AutoBash's replay mode is to run the standard predicate sets for all applications. AutoBash executes all standard predicates, $\{P_0, P_1, ..., P_n\}$, and aggregates their results as a binary result vector $R_{current} = \{1, 0, ..., 1\}$ (1 = succeed, 0 = fail). AutoBash also maintains a set of solutions $S_i$ from $\{S_0, S_1, ..., S_m\}$ from prior replays with their own standard predicate results $R_i$. Next, AutoBash compares the current result vector $R_{current}$ to the result vectors from prior replays, $R_i$, and computes their Hamming distance. AutoBash uses this Hamming distance to order the set of solutions it tries, starting with the solution that has an $R_i$ closest to $R_{current}$. The intuition behind this approach is that solutions that fix a particular problem tend to repair the same set of failing predicates; thus, by observing the results of standard predicate execution, AutoBash can guess which solutions are most likely to succeed.

More sophisticated heuristics could potentially be applied. For instance, AutoBash could also record and compare input sets for each standard predicate. The intuition is that similar bugs will cause predicates to fail in similar ways. Thus, there might be substantial similarity between the set of kernel objects observed by each predicate on computer systems that are exhibiting the same buggy behavior. We plan to investigate such advanced search heuristics in the future.

### 3.7 Validating system health

Much like a check-up with a family physician, AutoBash's final mode of operation validates that a properly configured system is continuing to behave correctly. AutoBash periodically (as a daily cron job) runs all the predicates in a computer's database. If any of these predicates fail, it produces a report that alerts the computer's administrator about the problem. We have set up one of our authors' desktop machines to run AutoBash health monitoring mode at three o'clock in the morning daily. AutoBash updates the predicate database to reflect the system health after each run.

We configured the AutoBash cron job to run at night so as to minimally perturb the performance of foreground applications running on the computer. Speculative predicate execution assures that there is no causal impact from periodically running predicates on the rest of the computer system.

The input set and result of each predicate is collected after a predicate is run. This information is used during replay mode to determine which predicates previously failed prior to attempting a solution, and also to determine which predicates need not be re-run while testing a potential fix.

## 4. LIMITATIONS

We assume that a predicate is written to evaluate certain desired properties of the system correctly; that is, the predicate should deterministically evaluate to true if those properties hold and false otherwise. If a predicate does not return the correct answer, AutoBash may miss a correct solution or falsely identify an incorrect solution. Also, AutoBash currently does not support debugging non-deterministic errors. Combining AutoBash with a system such as Rx [16] might help tease out such non-determinism. Potentially, one could intentionally vary non-deterministic inputs such as thread scheduling and the time of day to help explain errors to an administrator or user.

AutoBash is intended to debug user-level configuration errors on a single computer. Since AutoBash uses a layer-above approach implemented inside the operating system to track causality and roll back modifications, it cannot track causal effects for or undo kernel-level actions such as the insertion of a kernel module. Potentially, a multi-level approach to rollback, for instance, by using a virtual machine monitor to checkpoint and roll back kernel state [20], could allow AutoBash to handle such scenarios. AutoBash also does not support configuration of distributed applications that span more than one computer. However, if Speculator could be extended so that the operating systems of multiple computers could share speculative state as in systems

| Bug | CVS configuration problem description |
|---|---|
| 1 | Repository not properly initialized |
| 2 | User not added to CVS group |
| 3 | CVS performs unwanted keywords substitution |
| 4 | Setgid bit not set on repository, so group for new files is incorrect |
| 5 | $TMPDIR environment variable set incorrectly |
| 6 | $CVSROOT misconfigured for a CVS user |
| 7 | $CVSROOT not set for a different CVS user |
| 8 | $CVSROOT variable set but not exported correctly |
| 9 | Repository permissions allow global access |
| 10 | Repository created using wrong group |
| Bug | Gcc cross-compiler problem description |
| 1 | Cross-compiler tools not in the default path |
| 2 | Cross-compiler setup overwrites default path instead of appending |
| 3 | Dangling libcrypt.so symlink does not point to correct library |
| 4 | Archive tool (ar) not in the default location |
| 5 | Kernel header module.h contains wrong content |
| 6 | Compiler cannot invoke linker due to bad location |
| 7 | Cross-compiler specs file does not contain XScale architecture definitions |
| 8 | Cross-compiler not configured to accept -pthread option |
| 9 | C compiler configured correctly, but C++ compiler is not |
| 10 | Cross-compiler not configured to pass the static link flag to the linker |
| Bug | Apache HTTP server problem description |
| 1 | Apache cannot search a user's home directory due to incorrect permissions |
| 2 | Apache cannot read CGI scripts due to incorrect permissions |
| 3 | Symlink used to point to CGI scripts in a user's home directory, but Apache is not configured to follow symlinks |
| 4 | Apache configuration does not allow CGI execution in user home directories |
| 5 | Misconfiguration treats CGI scripts as regular Web pages |
| 6 | Apache not configured to load PHP module |
| 7 | Handler not set for PHP pages |
| 8 | Apache not configured to use index.php as default |
| 9 | User has insufficient permission to use .htaccess authorization |
| 10 | File .htaccess in a user's home directory configured incorrectly |

**Table 1: Description of injected bugs**

such as Time Warp [8], AutoBash could tackle distributed configuration management.

AutoBash is currently only targeting "functional" problems associated with misconfiguration and does not handle performance problems. To correctly diagnose performance problems, we need accurate accounting of shared resources (CPU, memory, etc.) with the system. Since AutoBash also consumes these resources for speculation and roll back, it would interfere with performance debugging.

Finally, AutoBash assumes that all configuration actions happen under its purview. One can imagine, for instance, a delayed configuration action such as a process that reads a configuration file once every 24 hours. AutoBash would not observe this interaction since it limits causality tracking to the time period when AutoBash is employed to fix a problem. While one could track *all* causal interactions in the system to capture such activities, it would be quite difficult to separate out configuration activity from other causal interactions that have no effect on the problem being debugged.

| Predicate | CVS predicate description |
|---|---|
| 1 | a user checks in a project and checks it out again |
| 2 | a user checks in a project, and a different user checks it out |
| 3 | same as predicate 1, but assumes a default repository is defined |
| 4 | same as predicate 3, but also checks that unauthorized users cannot access repository |
| 5 | checks if CVS performs unwanted keyword substitutions |
| Predicate | gcc cross-compiler predicate description |
|  | *Note: For all predicates, we check that the compilation succeeds and the compiled executable is the right file format* |
| 1 | take a "hello world" .c file, compile it with explicit path names |
| 2 | take a "hello world" .c file, compile it using default paths |
| 3 | take a kernel module .c file, compile it |
| 4 | take a .c file, compile it, link it to a shared cryptography library |
| 5 | take several .c files, compile them into object files, archive the object files into a static library, compile a program that links in the static library |
| 6 | take a .cc file, compile it with a c++ cross compiler |
| 7 | take a .c file, compile it, statically link in a math library, check if the compilation succeeds and the compiled executable is statically linked to the math library |
| 8 | take a multi-threaded .c file, compile it for the XScale architecture |
| Predicate | Apache HTTP server predicate description |
| 1 | `wget` Apache's default home page |
| 2 | `wget` a user's default home page |
| 3 | `wget` the result of a CGI script from Apache's default root directory and `diff` the output with the expected output |
| 4 | `wget` the result of a CGI script from a user's home directory and `diff` the output with the expected output |
| 5 | `wget` the result of a PHP test page |
| 6 | `wget` a PHP test page that is set to be the default page |

**Table 2: Description of predicates for each application**

# 5. EVALUATION

Our evaluation answers the following questions:

- How well can AutoBash identify solutions to configuration problems?
- How effective is causal dependency analysis in reducing predicate testing?
- Can AutoBash identify solutions to configuration problems involving multiple applications?

## 5.1 Setup

All of our experiments were run on a Dell Precision 370 desktop computer with a 3.00 GHz Pentium 4 processor and 2 GB of memory. The computer runs a Red Hat Enterprise 3 Linux kernel version 2.4.21. All trials using AutoBash run with a modified version of the kernel that includes Speculator — all other trials run with the default 2.4.21 kernel.

## 5.2 Effectiveness of AutoBash

To validate AutoBash, we injected bugs into our computer system for three applications: the CVS version control system, the gcc arm-linux cross-compiler, and the Apache HTTP server. Since we did not have an existing bug database to draw upon, we identified 10 common bugs for each application by searching through FAQs, manuals, and troubleshooting reports on the Web. Most of the bugs are limited to one application, but some bugs in one application also cause another application to fail. AutoBash can fix inter-application misconfigurations; we demonstrate one such example in Section 5.3. Table 1 contains a short description of each bug we tested. We also created 5–8 predicates that test the configuration of each application — these predicates are shown in Table 2. These predicates are sufficient to cover all the bugs in Table 1; i.e., each bug causes at least one (and often several) predicates to fail.

For each bug, we created a script that injected the bug, as well as a solution script that fixed the problem. In these experiments, we do not assume that there exists a set of standard predicates for the application as described in Section 3.6; thus, AutoBash uses the following heuristic to order the solution search space. As AutoBash has no hint about which application is misconfigured, it runs all predicates from Table 2 in the initial predicate testing. Based on the results of this initial predicate testing, AutoBash tries solutions from the application with the most failed predicates. When verifying the correctness of a solution, AutoBash runs the failed predicates first and only re-runs those predicates whose input sets intersect with a solution's output set.

We evaluated the correctness of AutoBash by injecting each bug from Table 1 into our test computer system and using AutoBash in replay mode to fix the bug. In every case, AutoBash was able to find a solution that corrected the misconfiguration.

Next, we evaluated the performance impact of speculative execution and benefit gained by causality analysis to a baseline AutoBash implementation. We created three versions of AutoBash:

- **No speculative execution**: This version uses neither speculative execution nor causality analysis. We created scripts to undo each solution and predicate. Without speculative execution, we found it can sometimes be quite hard to undo the effects of predicate testing because some applications do not provide an interface to undo actions. For example, to undo the effects of a CVS predicate testing, our undo script removes the directory added in predicate testing from the CVS repository and removes lines from CVS's history file.

- **No causal analysis**: This version uses speculative execution but not causality analysis. Comparing the execution time of the first and second bars in each dataset gives the performance overhead of speculative execution.

- **AutoBash**: This version uses both speculative execution and causality analysis. Thus, comparing the execution time of the second and third bars in each dataset shows the amount of time saved by tracking output sets of solutions and input sets of predicates.

The trouble we had in developing manual undo scripts for the "No speculative execution" version demonstrated to us the importance of speculative execution; with AutoBash, the operating system tracks the causal effects of a predicate and undoes them after testing the predicate.

We evaluated these three versions as follows. For each bug in Table 1, we compare the total search time to find the solution. Our results for each version are shown by the three bars for each bug in Figures 4, 6, and 8. For each bar, we break the total search time into three parts: initial predicate testing (before any solution is tried), solution execution and undo, and predicate testing (to verify a potential solution).

We also show the number of predicates run in initial predicate testing and solution searching in Figures 5, 7, and 9. We coalesce the first two versions into "No causal analysis" because they run the same number of predicates.

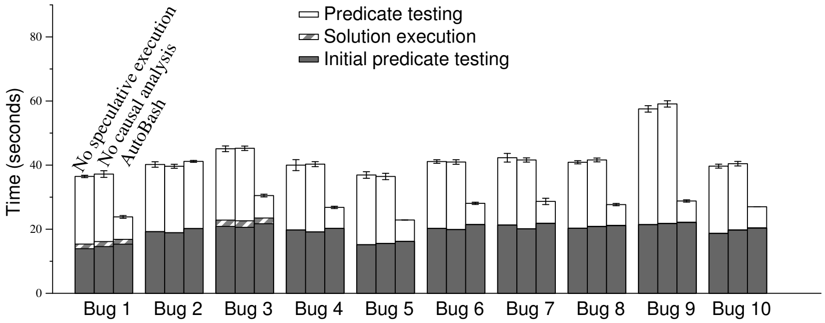
### *5.2.1 CVS*

Figure 4 shows the time needed to solve the ten CVS configuration bugs described in Table 1. Comparing the first two bars in each dataset shows that speculative execution adds minimal overhead for this benchmark – the maximum overhead of speculation is 2.7% for bug 9. The initial predicate testing time is unavoidable because we assume that AutoBash receives no hints from the user and hence runs through all predicates. We found that solution execution time is negligible. Hence, the only component of our replay mode that can be improved is predicate testing. This observation led us to focus on using causality analysis to reduce the number of predicates that need to be tested.

Without causality analysis, the number of predicates that must be tested tends to increase roughly in proportion to the number of solutions tried. Since AutoBash tries the solution from the application with the most failed predicates in the initial predicate testing, AutoBash is able to find the right solution in an average of 4.5 trials. Also, AutoBash tests the failed predicates first after trying a solution so it is able to quickly determine if the solution is incorrect.

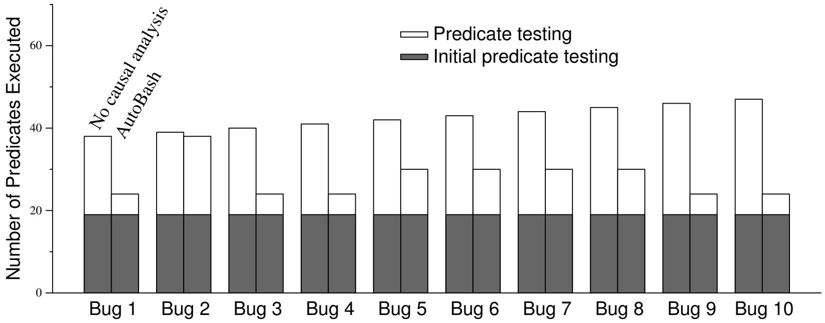
An interesting observation from our evaluation is that the subtlety of the bug impacts execution time — for instance, bug 9 allows unauthorized access to the repository; therefore, even though AutoBash only needs to re-run one predicate to determine a solution does not work, it takes longer for that predicate to detect a problem. However, bugs 1 and 5 are catastrophic; since all CVS actions fail, any predicate fails immediately.

Comparing the second and third bar in each dataset shows the performance benefit of causality analysis. For most bugs, causality analysis reduces the time to find a solution by 31–51% and predicate testing time by 67–82%. Figure 5 clarifies this benefit by comparing the number of predicates run with and without causality analysis. With causality support, AutoBash runs fewer predicates. Thus, the performance of AutoBash tends to degrade much more slowly with the number of solutions tried.



This figure shows the time to find a solution for the 10 CVS bugs in Table 1. The left bar in each data set shows the time to find a solution without speculative execution or causality analysis, the middle bar shows the time with only speculative execution, and the right bar shows the time with both. Each result is the mean of five trials — the error bars show 95% confidence intervals.

**Figure 4: AutoBash performance for CVS benchmark**



This figure shows the number of predicates executed by AutoBash while finding a solution for the 10 CVS bugs in Table 1. Five trials were run for each bug; however, the number of predicates executed was identical in each trial.

**Figure 5: Number of predicates executed for CVS benchmark**

### 5.2.2 *Cross-compilation*

Figures 6 and 7 show results for the ten gcc cross-compiler bugs in Table 1. The performance impact of speculative execution, as shown by the difference between the first two bars in each dataset in Figure 6, is within experimental error for all bugs. For bug 2, which overwrites the default path environment variable, AutoBash is able to identify that the bug affects CVS and re-runs CVS predicates. As with the CVS experiments, without causality support, the number of predicates that needed to be tested to find a solution tends to increase with the number of solutions tried and the total execution time is affected by the subtlety of the injected bug. For most bugs, causality analysis improves solution search time by 34–48.5%.

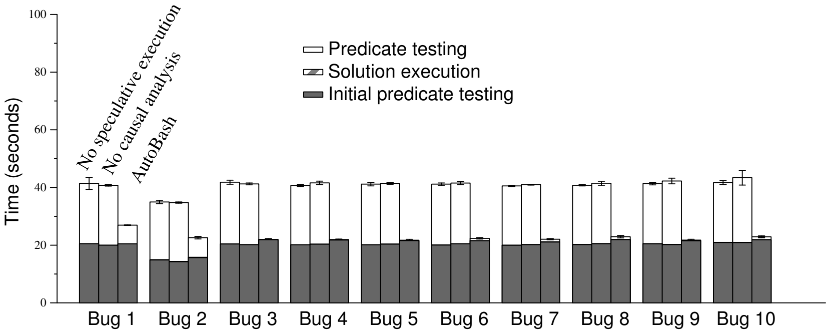
An interesting observation highlighted by our cross-compilation evaluation was that predicate testing time for different applications can vary greatly. For instance, the cross-compilation predicates run much faster than Apache's. With causality analysis we improve predicate testing time by 67–99% because AutoBash did not re-run any Apache predicates as they were not causally dependent on any of the cross-compilation solutions.

### 5.2.3 *Apache*

Figures 8 and 9 show results for the ten Apache bugs in Table 1. The time to find a solution without causality tracking scales roughly linearly with the number of bugs because Apache has a large amount of configuration state. Thus, the bugs we injected are mostly subtle; a misconfiguration often affects only a single predicate or two. The solution for bug 1 changes the permission of /home/USER, where all our predicates are located. Therefore, the output set of the solution for bug 1 intersects with the input sets of all 19 predicates. So, bug 1 does not benefit from causality analysis as all 19 predicates need to be run. However, bug 10 shows the largest benefit — its total execution time decreases by 58% and predicate testing decreased by 95% because causality analysis reveals that only one predicate needs to be re-run.

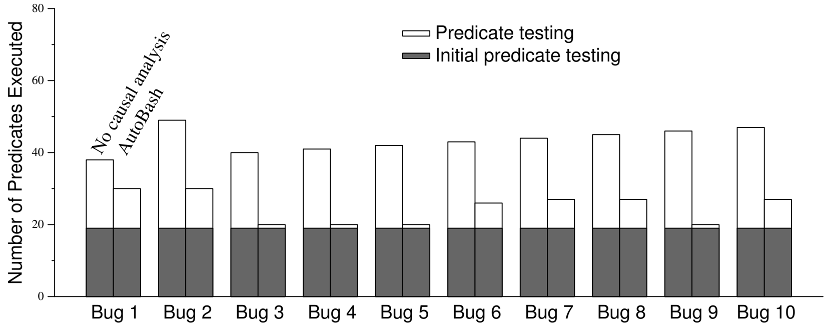
## 5.3 Case study

Configuration problems involving multiple applications can be more difficult to solve. We investigate the effectiveness of AutoBash in solving such problems by developing a misconfiguration scenario involving both CVS and Apache. In our scenario, the user sets up a CVS repository to manage the source tree for a website. The configuration bug is that the user did not turn off CVS's keyword substitution feature so that whenever a developer checks in a CGI Perl script, CVS automatically substitutes keywords such as $Id$ in the script with the username of the user who checks in the file. Later, when the user checks out the website and executes the CGI Perl script, he will encounter an incorrectly formatted HTML page.



This figure shows the time to find a solution for the 10 gcc bugs in Table 1. The left bar in each data set shows the time to find a solution without speculative execution or causality tracking, the middle bar shows the time with only speculative execution, and the right bar shows the time with both. Each result is the mean of five trials — the error bars show 95% confidence intervals.

**Figure 6: AutoBash performance for gcc benchmark**



This figure shows the number of predicates executed by AutoBash while finding a solution for the 10 gcc bugs in Table 1. Five trials were run for each bug; however, the number of predicates executed was identical in each trial.

**Figure 7: Number of predicates executed for gcc benchmark**

This bug can be resolved by running AutoBash in replay mode. We hypothesize an existing predicate in a standard predicate set shipped with Apache that works as follows: an Apache user checks in a Perl script to a CVS repository on the machine. The predicate, which runs as root, checks out the Perl script into a CGI scripts directory, `wget`s the results of executing that CGI Perl script, and `diff`s the output with the expected output.

We ran AutoBash in replay mode with the solutions for bugs described in Table 1 and predicates in Table 2. During initial testing, AutoBash finds that CVS predicate 5 fails and the remaining predicates succeed. Therefore, AutoBash tries solutions associated with CVS first and finds the solution that fixed CVS's bug 3 solves the problem. Even though the misconfiguration manifests as an Apache problem, AutoBash is able to identify that it is actually due to a misconfiguration in CVS.

## 6. RELATED WORK

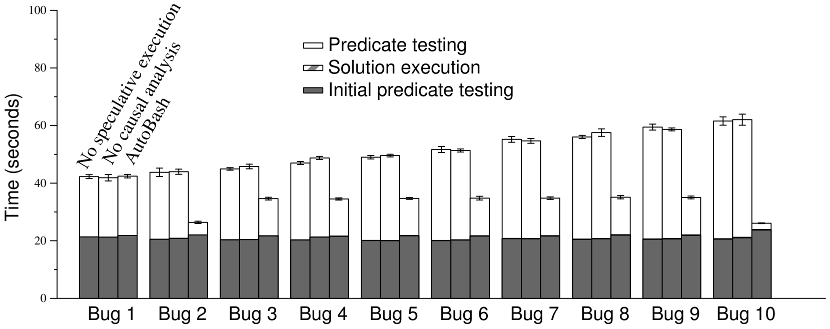
To the best of our knowledge, AutoBash is the first project to leverage operating-system-level causality tracking and speculative execution to improve configuration management.

Chronus [20] also looked at the use of checkpoint and rollback for configuration management. Chronus uses a virtual machine monitor to implement rollback at the granularity of the entire computer system. The VM implementation allows Chronus to diagnose kernel bugs, but would make it much harder to extract the causal information used by AutoBash to guide its search. Like AutoBash, Chronus uses user-defined predicates to test the behavior of the system. Chronus attacks a more limited problem: finding the point in time where a previously-working system ceased to operate correctly. AutoBash tries more generally to allow one system to learn from others by speculatively applying and testing fixes that have worked elsewhere for similar problems. Chronus shares our use of rollback to eliminate predicate effects, as does the work of Joshi *et al.* [9] in the IntroVirt project.

We have used Speculator [15] to implement checkpoint and rollback. Rx [16] and Pulse [12] apply operating system speculation to different domains: transparent recovery from non-deterministic application failure and deadlock detection, respectively.

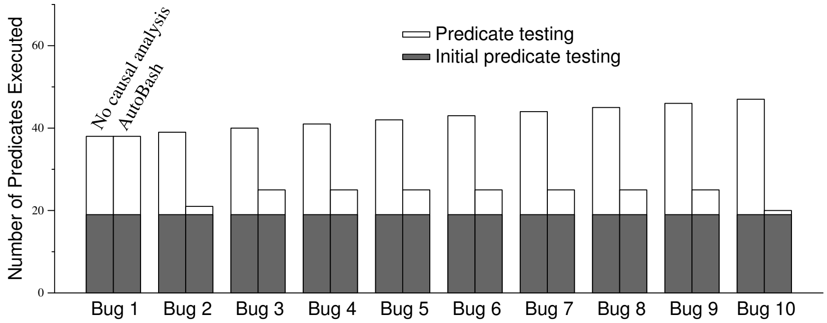
Brown and Patterson's Operator Undo [2] uses a form of checkpoint and rollback to allow administrators to fix mail configuration errors. However, their approach requires application modification, whereas AutoBash functionality requires no application-level support.

PeerPressure [18] and its predecessor, Strider [19], share the same overarching goal as AutoBash: making configuration management easier. However, AutoBash takes a fun-



This figure shows the time to find a solution for the 10 Apache bugs in Table 1. The left bar in each data set shows the time to find a solution without speculative execution or causality tracking, the middle bar shows the time with only speculative execution, and the right bar shows the time with both. Each result is the mean of five trials — the error bars show 95% confidence intervals.

**Figure 8: AutoBash performance for Apache benchmark**



This figure shows the number of predicates executed by AutoBash while finding a solution for the 10 Apache bugs in Table 1. Five trials were run for each bug; however, the number of predicates executed was identical in each trial.

**Figure 9: Number of predicates executed for Apache benchmark**

damentally different approach than PeerPressure or Strider; AutoBash reasons about *actions* rather than *state*. AutoBash therefore uses causal tracking and analysis to understand how user actions affect predicates that test system correctness, while Strider and PeerPressure apply statistical methods to reason about similarities between configuration state on different machines. The AutoBash approach works well in environments such as Linux where configuration state may be hard to find because it is scattered throughout the file system, rather than coalesced in a central registry. Like AutoBash, Strider and PeerPressure observe causality to determine the system state that causally affects buggy applications. However, unlike AutoBash, these tools do not follow causal links across processes. So, if one process observes state and causally affects another process that exhibits a bug, PeerPressure cannot trace the appropriate causal chain back to the misconfigured state. Ultimately, the problem of configuration management seems complex enough that it may be best to combine multiple approaches such as AutoBash causal analysis and PeerPressure state analysis.

Many prior systems reason about causal interactions. For instance, King's BackTracker [11] traces causal interactions to determine what state has been changed during an intrusion. Aguilera *et al.* [1] use causal tracing of RPCs to debug performance problems. Causeway [3] allows applications to inject metadata that follows causal paths for distributed applications. PASS [14] uses causality to annotate files with provenance that describes their causal inputs: our input sets try to capture similar information, but limit the scope of information collected to specific periods of time.

Clarify [5] improves error reporting by monitoring software execution to generate a behavior profile when an error occurs. It then applies a classifier to match the bug profile to erroneous execution reports previously submitted by other users. One can regard Clarify as focusing on causality within process execution, whereas AutoBash monitors causal interactions external to a process. AutoBash may benefit from using similar classification and machine learning techniques to those employed by Clarify.

AutoBash's replay mode currently uses a simple approach, Hamming distance calculated over a vector of predicate results, to determine which solutions to try first. Other approaches, drawing from machine learning and information retrieval, could potentially do a better job of identifying bugs and their corresponding solutions. For example, Cohen *et al.* [4] showed that an approach that uses statistical methods to capture relationships between low-level performance metrics and high-level behaviors outperformed an approach that used only the low-level metrics in tasks such as root-cause analysis and behavioral clustering.

## 7. CONCLUSION

This paper has explored how the operating system can contribute to reducing the burden of configuration management. AutoBash is a set of tools that leverages operating system support for speculative execution and causality tracking to automate many of the time-consuming tasks that occur when users deal with the complexity of modern computer systems. Our results show that OS support can reduce both the *time* and *user effort* needed to fix configuration errors.

At the same time, AutoBash tackles only a piece of the larger configuration management problem. Advances in other domains, such as machine learning, distributed information systems, and user interfaces will help provide the other pieces of the puzzle. Our future work therefore lies in integrating AutoBash with solutions from these domains.

## Acknowledgments

We thank Manish Anand, Brett Higgins, Ed Nightingale, Dan Peek, Kaushik Veeraraghavan, Benjamin Wester, our shepherd, Jeffrey Mogul, and the anonymous reviewers for feedback on this paper. We especially thank Peter Chen and Brian Noble for their last-minute editing help. The work has been supported by the National Science Foundation under award CNS-0509093. Jason Flinn is supported by NSF CAREER award CNS-0346686. Intel Corp. has provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, the University of Michigan, or the U.S. government. Ya-Yunn Su and Mona Attariyan were both awarded SOSP student travel scholarships, supported by Microsoft Corporation, to present this paper at the conference.

## 8. REFERENCES

[1] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 74–89.

[2] Brown, A. B., and Patterson, D. A. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Technical Conference* (San Antonio, TX, June 2003).

[3] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (Santa Fe, NM, June 2005).

[4] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 105–118.

[5] Ha, J., Rossbach, C. J., Davis, J. V., Roy, I., Ramadan, H. E., Porter, D. E., Chen, D. L., and Witchel, E. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007* (San Diego, CA, 2007).

[6] Holland, D. A., Josephson, W., Magoutis, K., Seltzer, M., Stein, C., and Lim, A. Research issues in no-futz computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 106–110.

[7] Huang, Q., Wang, H. J., and Borisov, N. Privacy-preserving friends troubleshooting network. In *Proceedings of the 12th Network and Distributed System Security Symposium* (San Diego, CA, February 2005), pp. 245–257.

[8] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., P.Hontalas, Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Weidel, J., Younger, H., and Bellenot, S. Time Warp operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX, November 1987), pp. 77–93.

[9] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 91–104.

[10] Kephart, J., and Chess, D. M. The vision of autonomic computing. *Computer 36*, 1 (2003), 45–52.

[11] King, S. T., and Chen, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.

[12] Li, T., Ellis, C. S., Lebeck, A. R., and Sorin, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Technical Conference* (31–44, April 2005).

[13] Mummert, L., Ebling, M., and Satyanarayanan, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995).

[14] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[15] Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[16] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 235–248.

[17] Su, Y.-Y., and Flinn, J. Slingshot: Deploying stateful services in wireless hotspots. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services* (Seattle, WA, June 2005), pp. 79–92.

[18] Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., and Wang, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 245–257.

[19] Wang, Y.-M., Verbowski, C., Dunagan, J., Chen, Y., Wang, H. J., Yuan, C., and Zhang, Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of Usenix Large Installation Systems Administration Conference* (October 2003), pp. 159–172.

[20] Whitaker, A., Cox, R. S., and Gribble, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 77–90.