AdaptGuard: Guarding Adaptive Systems from Instability

Jin Heo and Tarek Abdelzaher
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL, USA
jinheo@cs.uiuc.edu, zaher@cs.uiuc.edu

ABSTRACT
In this paper, we design, implement and evaluate AdaptGuard, a software service for guarding adaptive systems, such as QoS-adaptive servers, from instability caused by software anomalies and faults. Adaptive systems are of growing importance due to the need to adjust performance to a larger range of changing environmental conditions without human intervention. Such systems, however, implicitly assume a model of system behavior that may be violated, causing adaptation loops to perform poorly or fail. The purpose of AdaptGuard is simple: in the absence of an a priori model of the adaptive software system, anticipate system instability, attribute it correctly to the right “runaway” adaptation loop, and disconnect it, replacing it with conservative but stable open-loop control until further notice. We evaluate AdaptGuard by injecting various software faults into adaptive systems that are managed by typical adaptation loops. Results demonstrate that it can successfully anticipate instability caused by the injected faults and recover from performance degradation. Further, a case study is presented using an Apache Web server serving multiple classes of traffic. A performance anomaly is demonstrated, caused by unexpected interactions between an admission controller and the Linux anti-livelock mechanism. In the absence of a model that describes this mechanism, AdaptGuard is able to correctly attribute the unexpected problem to the right runaway loop and fix it.

Categories and Subject Descriptors
D.4.8 [Operating System]: Performance—Monitors; D.4.5 [Operating System]: Reliability—Fault-tolerance

General Terms
Design, Experimentation, Performance, Reliability

Keywords
Adaptive Systems, Causality Assumptions, Adaptation Graphs

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.

1. INTRODUCTION
As QoS-aware systems become larger and their applications become more complicated, an increasing need arises for self-tuning and control components to accommodate environmental dynamics and uncertainty [13]. A significant amount of recent work on server systems addressed closed loop control of system behavior [1, 12, 19, 18]. Such systems employ various regulation policies to tune different software performance knobs automatically to maintain acceptable performance and timing behavior in the face of a changing external environment. We call such systems adaptive systems.

Regulation policies in such systems are carefully designed by domain experts to prevent the system from becoming unstable. Naturally, the design of regulation policies makes implicit assumptions about their effect on other system components and the external environment. For example, one may assume that turning off a server may increase load on its mirrors [9]. When such seemingly correct assumptions are violated, the system may respond inappropriately to external stimuli possibly driving performance in the “wrong direction” in ways not predicted by the designer. When the problem occurs, the mission of AdaptGuard is threefold. In the absence of a user-supplied model that describes the system, AdaptGuard should (i) anticipate imminent performance degradation, (ii) attribute it correctly to the responsible chain reaction, and (iii) stop it. It is key to notice here that since performance degradation may occur in ways not predicted by the designer, it might not, in fact, be measurable (e.g., because the designer did not anticipate to measure the right variable). A key requirement of AdaptGuard is therefore to fulfill its threefold mission without the benefit of actually observing the performance degradation. AdaptGuard must use means other than direct observations to anticipate degradation. This distinguishes it from the trivial case where a control-loop-gone-bad is stopped because of a measured performance problem.

While many problem detection and diagnosis mechanisms are reported in the literature [2, 20, 10, 7, 22, 16, 23], AdaptGuard is different in the scope of problems it diagnoses. Being specializing in (runaway) adaptive behavior, it understands application-independent preconditions of instability, and can automatically detect them without actually understanding the root cause of the problem. This goal of root-cause-agnostic recovery is to be contrasted, for example, with learning or data mining techniques that seek to explain why the problem occurred [4, 2, 7, 20].

Also, unlike verification techniques where safety properties must be explicitly stated [8, 14, 3], AdaptGuard automatically infers the implicit assumptions a designer must have made that have a bearing on loop stability. When they are violated, AdaptGuard expects instability to occur and performs intervention, breaking the actual runaway loop. Hence, while performance degradation due to insta-
AdaptGuard is to detect such violations and recover acceptable adaptive systems exhibit poor performance. The primary concern of corrective actions. When these assumptions are violated due to system faults. We first consider two simple software faults, the missing file descriptor. Ordinarily, the secondary effect does not appear significant. Since most systems are open-loop stable. However, it is important to notice that, unlike previous techniques that address adequately-modeled systems, this paper is about handling unexpected modeling problems. Our recovery from instability must therefore occur in the absence of full observability of system state, including observability of performance variables that exhibit degradation.

AdaptGuard runs as a separate software service to properly monitor the target adaptive system. It exposes to the programmer the abstractions of performance sensors (something to be measured), regulation policies (controllers that decide what tuning needs to be performed in response to a measurement), and actuators (something to be correspondingly tuned). Using the interfaces provided by AdaptGuard, any regulation policy can be connected to any performance sensor or actuator registered by the programmer. Hence, AdaptGuard knows which sensor-regulation policy-actuator loops are in place. It then uses a simple heuristic to monitor stability of such loops. Unstable loops are opened and stability is restored since software systems are generally open-loop stable.

To better illustrate our approach, we implement a QoS-adaptive Web server testbed using AdaptGuard. We implement two typical regulation policies frequently encountered in the literature [12, 9]: an admission control policy and a power saving policy. We then artificially cause assumption violations by injecting software faults. We first consider two simple software faults, the missing file and the busy loop faults, followed by an interesting case where the two regulation policies are combined together creating instability. Our evaluation results demonstrate the efficacy of AdaptGuard in detecting assumption violations caused by the injected faults and restoring acceptable performance.

Further, we present a running case study to demonstrate that AdaptGuard is also useful in real-world scenarios. We present an admission controller for Web servers that preferentially drops lower priority requests over higher priority ones under overload. We show, however, that due to an interesting kernel-level mechanism [17] to protect against livelock and denial of service attacks, the feedback control loop of the admission controller destabilizes, causing the server to plunge deeper into overload and dropping indiscriminately a significant fraction of both low-priority and high-priority requests. When AdaptGuard detects preconditions of instability (namely, positive feedback), it takes action to stop the offending feedback loop. Correspondingly, it is shown that the server is able to provide better service to high priority clients.

The rest of the paper is organized as follows. Section 2 highlights key features of AdaptGuard. Section 3 describes the design of AdaptGuard. Section 4 discusses the implementation details. Section 5 evaluates our approach and Section 6 presents related work. Finally, we conclude with Section 7.

2. OVERVIEW

At the core of QoS-adaptive systems lie regulation policies implemented by regulation policy modules as depicted in Figure 1. They close feedback loops that continuously watch the system and take corrective actions when it leaves the desirable operating region. These policies make assumptions about the effect of their corrective actions. When these assumptions are violated due to system anomalies or faults that were not captured in the assumptions, adaptive systems exhibit poor performance. The primary concern of AdaptGuard is to detect such violations and recover acceptable performance of the target adaptive system upon assumption violations.

When building adaptive systems with AdaptGuard, the implementation of regulation policies (using sensor, regulation policy, and actuator object interfaces provided by AdaptGuard) can be easily and automatically translated into adaptation graphs that represent the underlying causality assumptions. AdaptGuard does not understand the user implementation of these objects. It merely implements the interface that carries communication between them. This communication happens to convey the performance variables being measured and controlled. It implements a causality chain from measurements to responses. This chain is, in essence, a feedback control loop, closed by a target application (e.g., process) that is monitored by AdaptGuard. Feedback control theory tells us that a stable feedback control loop must be negative. In contrast, positive feedback is unstable.

With the observation, AdaptGuard uses a heuristic-based approach. Given the communication chain between sensors, regulation policies, and actuators that a programmer implements (using the interfaces exported by AdaptGuard), it monitors the correlations between variables in the chain. When any correlation coefficient changes sign (or an odd number of them in the same chain do), the sign of the corresponding adaptation loop must have changed and hence, the loop must have become unstable. AdaptGuard learns the correct signs of correlation coefficients by monitoring system execution for a sufficiently long amount of time. This approach assumes a normally-correct system, which is reasonable because AdaptGuard is geared for protecting deployment-ready systems, as opposed to those in their early stages of debugging. It takes only a few minutes to learn the right coefficient signs, which is negligible compared to the time-to-failure of modern embedded and server systems. Hence, AdaptGuard discovers the causality assumptions made in the design of an regulation policy without needing explicit user input.

3. DESIGN

Adaptive systems make assumptions regarding the external effects of adaptive actions. We call these assumptions causality assumptions. For example, when designing QoS-adaptive Web servers, we know (from queuing theory) that the response time, $D$, decreases with increased system speed (e.g., CPU service rate), $\mu$, and increases with increased workload (e.g., request arrival rate), $\lambda$. This queuing-theoretic fact is a common foundation to much prior work on server admission control [12], performance adaptation [1], and energy control [9].

While queueing theory is correct, a designer might not always take everything into consideration. For example, a designer’s model might have ignored a secondary effect (such as virtual memory swapping overhead or a limit on the maximum number of open file descriptors). Ordinarily, the secondary effect does not appreciably affect performance, but it may become dominant in certain corner cases thus invalidating the model that ignored it. Since most
models are merely abstractions of the actual implementation, approximations inevitably exist and may cause assumption violations. Assumption violations may cause feedback (i.e., adaptation) loops to become unstable.

Section 3.1 defines more formally what we mean by causality assumptions that AdaptGuard automatically identifies and explains the procedures of inferring causality assumptions in Section 3.2. Section 3.3 reviews the notion of adaptation graphs. Section 3.4 describes our mechanism for detecting such violations at runtime. Recovery is described in Section 3.5.

### 3.1 Causality Assumptions

A causality assumption, \( A \rightarrow B \), (i) states that changes in variable \( A \) cause subsequent changes in variable \( B \), and (ii) indicates the direction of change (i.e., whether the two variables change in the same direction or in opposite directions). This definition is geared towards computing systems where most relations between parameters are algebraic (as opposed to, for example, relations expressed by differential equations). While estimation and control theory offer much more precise tools for model estimation (such as Kalman filters and recursive least squares estimators), our goal is to allow for a wide range of linear and non-linear functions to be represented by a simple and general model. A violation of assumption, \( A \rightarrow B \), therefore occurs if and only if either the causal relation between the two variables is broken (e.g., the two variables become independent), or the direction of change is reversed (e.g., they become inversely proportional instead of directly proportional).

Focusing on the stability of adaptation loops (feedback control loops), AdaptGuard only considers the variables used in adaptation loops for inferring causality assumptions. Since AdaptGuard implements the interface between performance sensor, actuator, and regulation policy objects, once the application designer has implemented the interface between performance sensor, actuator, and regulation policy objects, for example, to determine the key variables that affect system, or the percentage of requests dropped by admission control, the identified variables to detect the relation between them. Causal-relations among the variables, for example, to determine the key variables that affect system performance using adaptive loops.

Observe that, in terms of identifying the variables of interest for inferring causality assumptions, AdaptGuard is significantly different from the previous approaches that make substantial efforts to discover variables of interest and their correlations from a large number of variables extracted from system logs and measurements [11, 15]. Those techniques, however, may be used by application designers, for example, to determine the key variables that affect system performance when designing an adaptation loop.

### 3.2 Inferring Causality Assumptions

With the identified variables (that are visible to regulation policies), AdaptGuard infers (pair-wise) causal relations among the variables. Given a pair of variables, \( x \) and \( y \), AdaptGuard tries to figure out if causal relations, \( x \rightarrow y \) and \( y \rightarrow x \), exist. It calculates the correlation coefficient between the variable at the head of the arc and the time-displaced (into the past) variable at the tail, since past values of the latter causally affect the former. The correlation coefficient tells us the relationship between the two variables. A value of 1.0 corresponds to a perfectly positive correlation and -1.0 represent a perfectly negative correlation. Zero (or a very small value) indicates that no (or very small) correlation exists. The correlation coefficient between variables (time-displaced) \( x \) and \( y \) is estimated at the \( k \)th period as follows:

\[
R^t_{xy}(k) = \frac{\sum_{i=1}^{N}(x_{k-i-j} - E[x_{k-i}]) \cdot (y_{k-i} - E[y_{k-i}])}{(N-1)s_{x_{k-i}}s_{y_{k-i}}}
\]

where \( x_{k-i-j} \) and \( y_k \) are the value of the variables at instant \( k \rightarrow j \) and \( k \), respectively. \( E[x_{k-i}] \) and \( E[y_{k-i}] \) are sample means for the recent \( N \) (sample) values at instant \( k \rightarrow j \) and \( k \), respectively, and \( s_{x_{k-i}} \) and \( s_{y_{k-i}} \) are the sample standard deviations. Hence, \( R^t_{xy}(k) \) describes the causal effect of \( x \) on \( y \) after \( j \) time units. The delay of the causal effect varies depending on the two variables that are monitored. For example, an increase in workload (e.g., the number of requests) may affect the CPU utilization almost immediately. In comparison, turning on a machine in a server cluster would not increase the total processing capacity instantly. To capture the causal relations more effectively, a number of different correlation values (in terms of delays) can be calculated. For example, AdaptGuard can try \( R^t_{xy}(k) \) and \( R^t_{yx}(k) \) to infer the causality assumption, \( x \rightarrow y \). AdaptGuard then picks the one with the highest absolute value that shows the strongest correlation.

Then, AdaptGuard maps the continuous correlation value, \( R^t_{xy} \) (with the highest absolute value) to a discrete set of causal relations, \( \{+,-\} \). To do so, a cut-off threshold, \( \tau \), is used such that correlation values above \( \tau \) are interpreted as positive, below \( -\tau \) as negative, and values in between as no correlation (no arc). The same procedures are applied to infer \( y \rightarrow x \).

Observe that AdaptGuard in fact does not really develop a “quantitative” model to infer causality assumptions. It merely learns the sign of temporal correlation between a pair of variables. Differently from our approach, authors in [11] used a linear regression model to search relationships between system variables for fault detection and diagnosis. Also, authors in [6] presented an approach for capturing probabilistic relationships among system variables using a Gaussian mixture model for fault analysis. While those approaches may provide more precise and richer explanations, if the underlying model changes (e.g., due to workload changes), the inferred model (e.g., a model learned against one type of workload) would not hold any more, possibly giving false alarms in detecting faults. In comparison, our approach can still learn the sign correctly as long as the underlying causal relation remains unchanged, hence more robust to the changing operating environment.

Typically regulation policies need a small number of variables (usually less than 5 variables) to implement adaptation loops [1, 12]. Since AdaptGuard only focuses on the variables used in regulation policies, the number of the inferred causality assumptions that should be monitored tends to be small. Hence, in terms of scalability, AdaptGuard has a clear advantage over other fault detection mechanisms that explore all the possible variables to figure out variables of interest [11, 6, 21] or statistical or learning-based mechanisms that try to pin down the root cause of the problem [4, 2, 7, 20].

### 3.3 Adaptation Graphs

In order to reason about stability, it is useful to use the notion of adaptation graphs. A detailed description of adaptation graphs is presented in our previous work [9]. Briefly, vertices in an adaptation graph represent system variables of interest (automatically identified by AdaptGuard as discussed above). In a particular implementation, these typically include key performance metrics (e.g., response time, utilization, etc) and performance control knobs (e.g., number of threads allocated to a task, CPU speed in a DVS-capable system, or the percentage of requests dropped by admission control). AdaptGuard does not understand the semantics of these variables. It merely sees the values communicated across the interfaces.
it provides to the programmer for sensor, actuator and regulation policy objects. The directed edges (arcs) in the adaptation graph represent the causality assumptions between such variables. The edges are labeled positive, "+", if the changes are in the same direction and negative, "-", if they are in opposite directions. These edges are automatically discovered by AdaptGuard by performing the aforementioned inferring techniques. A cycle in the adaptation graph represents a corresponding adaptation loop and the sign of the loop is defined as the product of the signs of the arcs in the cycle. Since a stable feedback loop is negative, the sign of an adaptation loop should be maintained negative.

3.4 Detecting Assumption Violations

Once variables of interest are identified and correlations are discovered between these variables, an adaptation graph can be constructed. The automated-detection module monitors the sign of each arc in the adaptation graph to determine if a violation occurred. At each period, the automated-detection estimates the (current) sign of an arc \( x \rightarrow y \) (in the same way when it constructs the adaptation graph) and compares it against the (previously learned) sign of the arc given in the adaptation graph. If they are different, there is a violation. Further, if the sign of the corresponding adaptation loop becomes positive, it indicates a positive feedback loop, meaning system instability is observed. This process of checking causality assumption violations and positive feedback loops can be efficiently done online, since it is performed on a small number of inferred assumptions and is not computationally complex.

Observe that the sample size \( N \) in Eq. (1) for calculating correlation values may affect both the stability and the responsiveness of the automated-detection, since we use the cut-off threshold to determine the sign of causality assumptions. If the sample size is too small and the calculated correlation values keep varying around the boundary of the threshold, it may create instabilities. In comparison, with a too large sample size, the automated-detection may not act promptly upon assumption violations. Further, it can incur more overhead. We will show in the evaluation section (Section 5) that a reasonably small sample size around 50 works well.

Typically, the reason a correlation sign is flipped is because the designer’s model (and hence variables measured) is inadequate. The actual system model has more inputs and some of these may not be measured. Notwithstanding this lack of observability, positive feedback is still detected from the measured loop sign and hence instability can be assumed even if measured performance variables look “normal”. We demonstrate such a scenario extensively in Section 5.

While the broken assumptions can give us some hint why the loop is destabilized, it does not tell us what exactly caused the problem. However, the root-cause-agnostic attribute of AdaptGuard is desirable for online detection purposes, considering that finding out the root cause of the problem is time-consuming and usually not suitable for online detection purposes.

Observe that AdaptGuard will not catch all instability problems. In particular, unstable negative feedback is not detected. It is not equipped to detect unstable negative feedback because it only uses signs for stability analysis (and not loop gain). As long as the signs of all causality arcs remain the same, AdaptGuard will not intervene.

3.5 Recovery of Stability

When a violation is detected (i.e., an arc changes sign), the feedback control loops involving the bad arc are opened by AdaptGuard (which implements sensor and actuator communication as we described earlier), a backup control action is taken as defined a 

oro for each loop. In our current implementation, the loop is then resumed after a pause, in hope that the transient condition (e.g., thrashing) that caused the anomalous behavior may have passed. Figure 2 illustrates this procedure. The sequence of opening, fixing, and closing the loop is repeated with an exponential back-off in resuming the feedback. If the violation is transient, this approach usually works well as the feedback resumes correctly. If, after a pre-configured number of trials, the system cannot recover, feedback is not resumed. Since open-loop actions are stable, it is less likely that the execution of backup control actions severely disturbs the normal execution of the system.

The backup control action executed in the open-loop mode depends on the loop broken and must be specified in advance. Observe that, in the case of overload control loops in general, a safe open-loop action would be to reduce load using a non-regulation policy (e.g., drop all/most low priority requests). The backup control action may be different based on the main performance metric of applications. Suppose an adaptive system managed by an energy-saving policy that uses dynamic voltage/frequency scaling. One can assume that power consumption is the main performance metric. Then, the backup control action is to set the voltage/frequency level to its lowest value in order to minimize energy consumption. If it is most important to process as many user requests as possible when violations occur, the backup control action should be to set the voltage/frequency level to its highest value.

4. IMPLEMENTATION

AdaptGuard runs as a separate process to properly monitor and act on the target adaptive system when causality assumptions are violated. It provides simple object-based API interfaces with which application developers and administrators to register any (application-specific or system-wide) performance sensors and actuators used. An example of a sensor might be a function that measures server utilization or one that returns a measured server request rate. An example of an actuator might be a function that changes CPU frequency (in DVS-capable systems) or one that accepts a desired fraction as input and drops a corresponding fraction of inbound server requests as consequence. Adaptive algorithms involving these sensors and actuators are then implemented by regulation policies that control the sensor and actuator knobs registered with AdaptGuard. It specifies a standard object-based interface for sensors, actuators and regulation policies (controllers). It further exports method calls for communication among such objects, hence enabling flexible development of adaptive applications. Moreover, since method calls that connect sensors, actuators and regulation policies are implemented via AdaptGuard, it can automatically infer adaptation graphs and open any loop when needed (simply by not delivering the call from the controller to the actuator, and instead using a default open-loop setting).
Figure 3: The system architecture of AdaptGuard

Figure 3 describes the architecture of AdaptGuard. The node manager manages a list of nodes that represent physical machines in the system. It helps AdaptGuard to efficiently monitor the target adaptive system that runs a distributed application on a network of machines. The resource monitor subsystem contains handlers to registered sensor objects. The resource allocator subsystem similarly contains handlers to registered actuator objects. The event manager maintains a list of timer events and checks to see if there are timer expirations. Objects that require periodic invocation subscribe to the event manager to receive timer events. For example, sensors may subscribe to the timer clock to periodically gather data from sources. In AdaptGuard, communication between the modules on different machines or processes is performed through an RPC module that implements remote method calls using XML-RPC. The automated-detection module checks the signs of all arcs in the adaptation graph. Once it has learned the “normal” signs, it compares subsequent measurements against them to determine whether there are violations of causality assumptions.

The current prototype implementation of AdaptGuard is written in Python. Our measurement shows that the implementation written in Python consumes less than 3% of CPU utilization most of the time.

5. EVALUATION

In this section, we thoroughly evaluate AdaptGuard using both artificial fault injection and a case-study on a QoS-adaptive Web server testbed. We first present the results of the fault inject followed by the results of a case study. Then we summarize them at the end of the section.

5.1 Fault Injection

In this section, we evaluate AdaptGuard by injecting various faults that cause causality assumption violations. First, we explain the application implementation as seen by AdaptGuard then proceed with a description of testbed set-up and the experiments performed.

5.1.1 Application Implementation

We implement two typical regulation policies that are frequently found in the literature using AdaptGuard to manage the performance of a Apache Web server. The admission control policy [12] implements an admission control mechanism that adjusts the workload of the server by dropping a portion of incoming requests to prevent the system from being overloaded. The implemented control loop measures the current CPU utilization $Util$. The measurement is implemented as a sensor object that outputs $Util$ and sent to the admission control policy module that outputs a drop probability $Pd$ to the actual admission controller (implemented as an actuator module), which drops a fraction $Pd$ of incoming requests. Hence, the object outputs visible to AdaptGuard are $Util$ and $Pd$. AdaptGuard monitors correlations between these variables. Figure 4(a) shows the dominant causal relations. Indeed, as shown in figure, The arc $Util \rightarrow Pd$ reflects the admission control policy output explaining that increased utilization, $Util$, will raise the drop probability, $Pd$, by the feedback loop. The arc $Pd \rightarrow Util$ illustrates that the utilization, $Util$, is enforced based on the drop probability, $Pd$. The two arcs form a cycle (a feedback loop) where the product of the signs of the arcs is negative, indicating a negative feedback loop.

Figure 4: Adaptation graphs of the QoS-adaptive Web server (a) with the admission control policy and (b) with the DVS policy

The DVS policy [9] implements a power saving mechanism using dynamic voltage and frequency scaling (DVS). Similar to the admission control policy, the DVS policy measures the current CPU utilization $Util$. With the increased utilization beyond the target value, the DVS policy increases the frequency/voltage level $Freq$ to decrease the utilization and vice versa. Fig. 4(b) shows the dominant causal relations among them that form a negative feedback cycle.

5.1.2 Testbed Setup

AdaptGuard is installed to monitor Apache Web server that runs on the same machine. The machine is equipped with an Intel Celeron 2.53GHZ CPU and 512MB of RAM. We instrument several sensors using AdaptGuard to collect measurements such as the inbound request rate (as seen by Apache) and CPU utilization. Actuators are implemented to enforce the decisions made by the two policies respectively. We use httperf to generate HTTP requests on a client machine with Intel Pentium IV 3GHZ CPU and 2GB of RAM. The inter-arrival time of the generated requests is exponentially distributed with a mean of 0.01 sec. The admission control policy tries to maintain the utilization around 0.6 and the DVS policy aims to maintain the utilization around 0.8. All machines are equipped with Redhat Fedora Core 4 Linux.

For the first set of experiments, we consider two different types of faults: the busy loop fault and the missing file fault. The missing file fault happens, for example, when an administrator mistakenly removes files from the HTML directory in a Web server. The busy loop fault happens when the exit condition of a loop is never met so that the program runs infinitely. The two faults are injected individually into the QoS-adaptive Web server that runs either the admission control policy or the DVS policy. No backup control actions are taken, except that the server is restarted if a fault occurs.

For the second set of experiments, we investigate a more interesting scenario where the two regulation policies run at the same time in the QoS-adaptive Web server, causing the target system (the Web server) to destabilize due to conflicts between the two policies. This could happen even when there are no explicit software faults. Our previous paper [9] studied how one can detect such conflict at
design time. In this paper, we show that AdaptGuard can detect causality violations caused by such conflict and restore acceptable performance at run-time. For comparison, we conduct experiments with and without the automated-detection. A backup control action is installed for the DVS policy that forces the system to operate at the maximum frequency for 30 seconds, assuming the main concern is the achieved throughput when violations occur. No backup control action is installed for the admission control policy.

We set the cutoff threshold $\tau$ to 0.15 to determine the sign of each edge from the correlation coefficient value and the sample size $N$ is set to 50.

5.1.3 Experimental Results

Simple Fault Injection: We present the results when the two faults, the missing file and the busy loop faults, are injected to the QoS-adaptive Web server with the DVS and the admission control policies. For each experiment, one of the two faults is injected into the Web server managed by one of the two policies. Hence, we conduct total 4 experiments.

![Correlation graphs](image)

(a) Missing file: $Pd \rightarrow Util$ is violated

(b) Busy loop: $Pd \rightarrow Util$ is violated

Figure 5: Faults are injected to a QoS-adaptive Web server with the admission control policy

![Correlation graphs](image)

(a) Missing file: $Freq \rightarrow Util$ is violated

(b) Busy loop: $Freq \rightarrow Util$ is violated

Figure 6: Faults are injected to a QoS-adaptive Web server with the DVS policy

Fig 5 depicts the result when the admission control policy runs and Fig 6 shows the result when the DVS policy runs. Faults are injected at 450th second and the experiments are conducted for total 900 seconds. For all four cases shown in Fig 5 and Fig 6, AdaptGuard successfully detects violations as one of the two correlation coefficient values changes its sign shortly after the fault is injected. The missing file fault drives the system to experience low utilization regardless of the changes in the related variables ($Pd$ for the admission control policy and $Freq$ for the DVS policy), as the requests for the missing files are rejected. Hence, the causality assumptions, $Pd \rightarrow_{AC} Util$ for the admission control policy and $Freq \rightarrow_{DVS} Util$ for the DVS policy, are violated. Similarly, the busy loop fault makes the causality assumptions, $Pd \rightarrow_{AC} Util$ for the admission control policy and $Freq \rightarrow_{DVS} Util$ for the DVS policy, broken as the utilization remains 100% once the fault is injected. A while after the faults are injected, all correlation values become undefined value (plotted as zero in the figures), since it is impossible to compute correlation values when one of the two variables does not change over time ($Util$ and $Freq$ remain unchanged).

Both types of faults are not easily recognizable without carefully investigating all the related variables. The difficult part is to differentiate the symptom from the normal behavior of the system. For example, with the busy loop fault, the 100% CPU utilization can be simply interpreted as the overload condition due to increased user requests. However, we demonstrated that AdaptGuard effectively detected the violations by monitoring only a small number of causality assumptions automatically identified.

Combined Regulation Policies: We present the results when the two (well-working) regulation policies are combined together. We first run the DVS policy alone for 450 seconds and start the admission policy at 450th second. Hence, after 450th second the two policies run together. The testbed previously explained is used.

Destabilization of the two regulation policies is clearly seen in Fig. 7. In Fig. 7(a), both of the two correlations of the DVS policy start to fluctuate after 450th second when the admission control policy is launched. Their sign changes unpredictably showing that the feedback loop implemented by the DVS policy destabilizes. The admission control policy also destabilizes as seen in Fig. 7(b). To help readers understand why this happens, we depict the values of the key variables such as the CPU frequency and the drop probability in Fig. 8. For comparison, we also show the values when the automated-detection module is present in the figure together.

![Correlation graphs](image)

(a) Correlations of the DVS policy

(b) Correlations of the admission control policy

Figure 7: Correlations when the two regulation policies conflict and destabilize

![Graphs](image)

(a) CPU frequency

(b) Drop probability

Figure 8: Comparison of the key variables between when AdaptGuard is not used and AdaptGuard is used

In Fig. 8, without AdaptGuard, it is interesting to see that the frequency drops to the lowest level, while almost all incoming requests are dropped (the drop probability goes up to 1.0). This
means that the application (Web server) does not accomplish any useful work. In detail, as the admission control policy starts to run, it sees that the utilization is around 80% since the DVS policy was already running. Hence, it drops requests to meet its target, 60%. As a result, the DVS policy gets to see the utilization is lower than its target, 80%, decreasing the frequency further to increase the utilization. This chain of actions forms a positive feedback cycle across the two policies, destabilizing the entire system to get all requests dropped eventually (see Fig. 8(b)), though the system is not really overloaded.

Merely monitoring performance metrics such as throughput, the CPU frequency, and the drop probability would not easily reveal the problem unless you have certain models established to compare with. AdaptGuard automatically learns the causality assumptions and detects their violations without any a priori model. When the assumption violations are detected in the DVS policy, the backup control action replaces the original DVS policy to set the CPU frequency to its maximum. As a result, it allows the CPU utilization to decrease, causing the admission control policy to drop less requests than the case without the automated-detection module.

5.2 Case-Study

In this section, we present a running case study to better demonstrate the efficacy and the limitations of our approach. We will demonstrate that AdaptGuard can successfully detect assumption violations caused by subtle interactions between a kernel-level mechanism for overload control and an admission control policy that manages a QoS-adaptive Web server. Further, we will show that AdaptGuard can effectively recover from performance degradation caused by the violations.

5.2.1 Testbed Setup

In addition to CPU utilization Util, the admission control policy measures the current service request rate Req to adjust the drop probability Pd in a more fine-grained way. The drop probability Pd enforces the number of accepted requests Req which in turn affect the CPU utilization Util. Hence, the variables visible to AdaptGuard are Req, Util and Pd. Figure 9 shows the identified causal relations. Further, the admission control policy serves client requests with multiple priority classes.

Figure 9: Adaptation graph of the QoS-adaptive Web server

A QoS-adaptive Web server is implemented using AdaptGuard in the same way as presented in Section 5.1. In order to overload the Web server, we use total three client machines with Intel Pentium IV 3GHZ CPU and 2GB of RAM. Requests are generated using multiple instances of httpf here such that the average request rate increases linearly over time to see reactions of the admission controller to the various ranges of workload. The total request rate starts at 30 req/s and becomes 1500 req/s after 1000 sec. One client machine generates high priority class requests and the other two generate lower priority class requests.

For comparison, we ran the system with and without the automated-detection module. Since the loop in our example is an overload control loop, backup control simply sets the drop probability Pd to a fixed high value for 30 seconds when the automated-detection module is present.

5.2.2 Experimental Results

We observed that the closed loop admission control policy performs well most of the time, protecting the system from being overloaded while keeping the utilization around the desired value of 0.8. Not all runs, however, are repeatable. On some occasions, the feedback loop is destabilized (at the same level of load that was previously handled successfully). A sharp indiscriminate increase is observed at the client side in dropped high-priority and low-priority requests alike. The high drop rate persists for both types of requests even though the external rate of high-priority requests alone should not overload the system. This is in direct contradiction to the design intent of the admission controller that is to drop low-priority requests first. As we show in the rest of the evaluation, the anomaly is due to an assumption violation triggered by the kernel’s anti-livelock mechanism.

We then observed that when the automated-detection module is used, performance is quickly restored and the drop probability of higher-priority clients goes back to (approximately) zero. These results are shown in Figure 10(a) and Figure 10(b). Figure 10(a) shows the number of low-priority and high-priority requests dropped indiscriminately at the kernel interface when the feedback loop works well, when it does not work, and when AdaptGuard automated detection is used. Figure 10(b) shows the same for the total number of dropped requests as seen by the client (i.e., including both kernel and admission control drops).

The interesting part about this example, as we show shortly, is that that stability was restored. After all, AdaptGuard simply opens the unstable loop and we have only one feedback control loop in this simple example. The interesting part was that instability was detected at all. The issue is, to the server, all performance metrics
Probability

the new Linux API called NAPI of kernel 2.6 that prevents live-

vention drop in Figure 12(a) was not due to a decrease in load. It is

shown in Figure 10(a)), is that a significant number of requests are

Figure 11(b)). What the admission controller does not know (as

controller to believe that the system is underloaded. Hence the admis-

bibilized. The figure shows that a sharp drop in utilization occurs

Figure 11(c).

Figure 12 presents a case when the admission controller is desta-

Closed Loop Control: Figure 11 depicts closed loop perfor-

performance when the admission controller works well. The utiliza-

was moderate, the input request rate seen was low, the admission

controller was fully open most of the time, and the delay was within

bounds. The signs of the problem were manifest only at the client

side, where most requests timed out. The disparity was because we

did not have a sensor to measure drops from the kernel queue. The

regular socket API does not offer such interface, and since server

software is designed for portability it cannot depend on it. Drops

from the kernel queue were substantial and completely invisible to

the server. Nevertheless, AdaptGuard “conjectured” (given only

those measurements that the designer used in their control loop)

that the control loop must be unstable and disconnected it. In the

following, we explain in more detail the above scenario and perfor-

ance observations.

Closed Loop Control: Figure 11 depicts closed loop perfor-

mance when the admission controller works well. The utilization

is successfully kept around the desired value of 0.8 (Figure 11(a)).

As the incoming request rate increases, the admission controller

increases the drop probability (Figure 11(b)) to keep the utiliza-

tion around 0.8. Observe also that as the request rate increases the

kernel overhead due to network interrupts increases as shown in

Figure 11(c).

Figure 12 presents a case when the admission controller is desta-

bibilized. The figure shows that a sharp drop in utilization occurs

(compare Figure 12(a) to Figure 11(a)) causing the admission con-

roller to believe that the system is underloaded. Hence the admission

controller stops dropping requests (compare Figure 12(b) to

Figure 11(b)). What the admission controller does not know (as

shown in Figure 10(a)), is that a significant number of requests are

now being dropped at the network card in the kernel. The utiliza-

tion drop in Figure 12(a) was not due to a decrease in load. It is

in fact triggered by an increase in load! The drop is attributed to

the new Linux API called NAPI of kernel 2.6 that prevents live-

locks by resorting to polling under heavy workload with reduced

network interrupts. This results in an assumption violation since

utilization decreased (instead of increasing) with increased request

rate. The switch from interrupt-driven to mostly polling I/O is well

illustrated in Figure 12(c), where we see a sharp drop in interrupt

overhead.

While the admission controller believes that utilization has dropped

and attempts to increase it, polling does not in fact catch all packets

successfully, resulting in indiscriminate drops of packets at the net-

work interface. Both high-priority and low-priority clients suffer as

was shown in Figure 10(a) and Figure 10(b).

Closed Loop with AdaptGuard: Figure 13 presents the result

of running a closed loop experiment with AdaptGuard support for

automated-detection. Figure 13(c) depicts the estimated correlation

coefficient values of the arcs in the adaptation graph. We do not ex-

plicitly show the correlation coefficient for the arc $\text{Util} \rightarrow \text{Pd}$

since it is implemented by the admission control software, and thus

does not constitute an interesting measurement. The correlation co-

efficient of arc $\text{Pd} \rightarrow \text{Req}$ becomes positive as the utilization drops

sharply and stays negative after that, while that of $\text{Req} \rightarrow \text{Util}$

starts fluctuating after the load settles in. We set the threshold $\tau$ to

0.25. AdaptGuard checks causality assumption violation resolving

the violation when detected. Note that the zero values (undefined

values) of arc $\text{Pd} \rightarrow \text{Req}$ are not considered as violations since

the controller was not in action. The resulting drop probability is

depicted in Figure 13(b). Figure 13(a) depicts the utilization and

Figure 13(d) shows network interrupts.

The impact of AdaptGuard was illustrated in Figure 10(a) and

10(b). The third bar in each set shows the drops experienced with

AdaptGuard. It can be seen from Figure 10(a) that the silent ker-

nel drops are virtually eliminated. This explains the performance

in Figure 10(b) showing that only low-priority requests are dropped

while the high-priority requests are not, despite the assumption vio-
5.3 Summary

In summary, we developed and demonstrated, using a number of examples, AdaptGuard capable of detecting instability due to assumption violations. Such instability is insidious because it progresses while all performance measurements look “normal”. In essence, AdaptGuard does not see the performance degradation either and hence does not know for a fact that the target system is unstable. This is because the target system does not offer access to the sensors needed to measure the performance degradation, and even if such measurements could be made, it is not clear in advance what sensors AdaptGuard should have to see unexpected problems. The contribution, therefore, lies in detecting the problem using only those sensors that the target system uses for their normal operation. Given only those sensors, AdaptGuard automatically “reverse-engines” the causality assumptions the designer must have made, and monitors them for violations. When a violation is detected using those sensors, AdaptGuard “guesses” that the system may now be unstable (even though it cannot observe the instability), and takes action to restore stability successfully.

Figure 13: Closed loop with AdaptGuard

6. RELATED WORK

Several automated detection and diagnosis mechanisms have been proposed recently. Some of them have focused on troubleshooting problems caused by system misconfiguration [22, 16, 23]. For example, Wang et al. [22] proposed an automated troubleshooting approach to diagnose the root cause of system misconfiguration. It uses a statistical method to derive rankings on probable causes using empirical Bayesian estimation. On the other hand, several techniques tackled the isolation of performance problems or system failures. [2, 7, 20, 4]. For example, Aguilera et al. [2] proposed a tool that isolates the performance bottleneck in a distributed system composed of heterogeneous components (from different vendors without source code). It traces messages exchanged between the nodes to find a critical path that caused the system’s latency. Vertical profiling [7] is a profiling technique to correlate various system measurements with each other to explain performance anomalies using statistical and visualization techniques. All the above automated detection and diagnosis techniques are usually performed offline since they employ a complex statistical method. AdaptGuard complements those offline tools rather than replacing them, as AdaptGuard only provides a temporary recovery mechanism at run-time when causality assumptions are broken. Furthermore, the main goal of AdaptGuard differs in that it merely attempts to recover from causality assumption violations. It does not care to pin down the root cause of the problem.

Run-time monitoring/verification techniques have been developed to provide assurance of the correctness of program execution [8, 14, 3]. In those systems, monitoring is usually performed based on a specification of system requirements (correct behaviors) and the target software system needs to be instrumented to notify the monitoring system of its state changes. For example, Java PathExplorer [8] monitors the execution of a Java program to check the system state against the desired system properties written in temporal logic provided by users. The Java bytecode is instrumented to provide an execution trace of the program, generating a sequence of events used by the monitoring process. Our work is similar, with the exception that causality assumptions to be monitored can be inferred by AdaptGuard in an automated fashion.

A number of approaches [21, 15, 11, 6] have been developed for the purpose of automatic fault detection and performance management without a priori knowledge or the help of domain experts, similar to our work. For example, Sun et al. [21] developed a mechanism for problem detection in distributed systems by constructing a state machine. However, it needs detailed system logs to build a state machine, while AdaptGuard does not need any to build adaptation graphs. Kumar et al. [15] developed an approach to automatically infer the relationship between the variables of interest and the controllable variables using probabilistic modeling techniques to derive component-level objectives. Although their approach can suggest new component-level objectives to meet the service level objectives (SLAs) according to changes in the operating environment, it cannot differentiate SLA violations from system instability.

In [11, 6], authors presented fault detection mechanisms using linear models [11] and probabilistic models [6]. These techniques aim to extract invariants, which are temporal correlations between a pair of system variables that hold all the time. By tracking abrupt changes in invariants, one can detect faults in distributed systems. However, they may trigger false alarms when the model is invalidated for benign changes, for example, such as user workload changes [5]. AdaptGuard differs in that it does not use any quantitative model, only deriving the sign of temporal correlation. Hence, it is more robust to changes in the operating environment.

QoS-adaptive systems typically aim to achieve consistent performance levels over time without human intervention. Several researchers used online feedback control to meet those goals [1, 12, 19, 18]. For example, Abdelzaher et al. [1] presented a Proportional-Integration (PI) control loop for an Apache Web server that adaptively changes the number of assigned processes (or threads) to meet soft real-time latency requirements. Karma et al. [12] used a feedback control loop to calculate admission probability to adjust load on a multi-tier Web server system to guarantee end-to-end delay requirements.
7. CONCLUSIONS

Adaptive software systems are designed subject to causality assumptions that describe the behavior of regulation policies. Normally correct causality assumptions can sometimes be broken at run-time due to various implementation peculiarities of complex systems. This causes severe performance degradation. This paper presented a software service, AdaptGuard that provides an automated mechanism to detect and correct causality assumption violations at runtime with a simple online statistical method. We evaluated our approach thoroughly using fault injection and a case study on a QoS-adaptive Web server. We injected several different faults from simple ones to a more complicated scenario to demonstrate the efficacy of our approach. The automated detection mechanism of AdaptGuard successfully detected performance degradation caused by the faults and recover acceptable performance quickly. Furthermore, a case study was conducted on a QoS-adaptive Web server with an admission controller that serves multiple priority classes for users. At high workload, the server’s admission controller sometimes experienced performance degradation caused by a Linux livelock protection mechanism. Consequently, both high and low priority requests were indiscriminately dropped. AdaptGuard caught the problem at runtime, improving system performance by serving high priority requests as it should while not increasing the drop rate of lower-priority ones.

8. REFERENCES


