HotDep '06: Second Workshop on Hot Topics in System Dependability

Seattle, Washington
November 8, 2006

FINDING THE NEEDLE IN THE HAYSTACK

Summarized by Yin Wang

■ ***Comprehensive Depiction of Configuration-dependent Performance Anomalies in Distributed Server Systems***

Christopher Stewart, Ming Zhong, Kai Shen, and Thomas O'Neill, University of Rochester

Presenter: Chris Stewart

Distributed server systems such as J2EE application-server systems have wide-ranging workload conditions. The assumption here is the reasonable performance expectation based on knowledge of the system design (e.g., Little's Law). The problem is performance anomalies, that is, when performance falls below expectation. Previous work shows that anomaly characterization can aid the debugging process and guide online avoidance. Chris's goal is to depict *all* anomalous conditions.

A three-step process was taken: (1) generate performance expectations by a whole-system performance model; (2) search for anomalous run-time conditions; and (3) extrapolate a comprehensive depiction. An example was shown of a submodel hierarchy (a four-level submodel for J2EE application servers), with its advantages and limitations. The next step is to determine the anomaly error threshold, which is different for online avoidance and debugging. Then Chris explained decision-tree-based depictions, why they chose to use decision trees, and how they classify anomaly conditions. For the case study of JBoss, where three performance anomalies were found and fixed, the decision tree displayed with the three anomalies described. This approach cannot detect nondeterministic anomalies, and the model accuracy requires manual investigation. Furthermore, debugging remains manual. The take-away message is that depiction of anomalies can aid debugging and avoidance.

Jay Wylie asked how to interpret anomalies that are good. Chris responded, "We don't consider that. The anomalies here are out-of-expectation anomalies." To Ken Birman's question of whether the magnitude between anomalies and normal performance is similar, Chris said that it is based on empirical observation. What about a known source of anomalies? For example, when garbage collection kicks in, does the performance degrade? Is this something you can't model? Chris agreed that this was a problem, but he said that his group hopes "to block out those known anomalies." John Wilkes asked how hard it is to build models. "The model is borrowed heavily from the NSDI '05 paper. Actually, a simple model is adequate for it, like Little's Law," was the response. In reply to a question on how controlled searches must be in order to detect anomalies, Chris commented that this work involved a benchmark-controlled environment. For a long trace of system execution, the performance variation may be huge.

Geoff Voelker asked, "In terms of the size of configurations explored, they seem to be relatively small. In a large system you may have lots of choices, for example, cache parameters. What do you do in this case?" Chris replied, "We hope to investigate a systematic method to explore system configurations. Within this work, we have eight run-time conditions and 7 million possible configurations." The final question, "Do you know when to stop exploration?" elicited a response of "It depends on end use. If you want to do avoidance, you want to stop when you manage to satisfy the performance goal. For debugging, it depends on the quality of code you want."

■ ***Static Analysis Meets Distributed Fault-Tolerance: Enabling State-Machine Replication with Nondeterminism***

Joseph G. Slember and Priya Narasimhan, Carnegie Mellon University

Presenter: Joseph Slember

State-machine replication is a standard way to add fault tolerance, but if replication is not 100% deterministic, it is difficult to do. The goal here is to target nondeterminism when it matters, and programmer intent must be respected. Joseph showed a picture of a three-tier replicated server system and explained the complexity of the problem. The approach adopted is compile-time static analysis with run-time compensation.

Next Joseph explained the taxonomy of nondeterminism (abbreviated as ND hereafter). ND–I includes pure (or first-hand) ND, for example, `random()`, `gettime()`, and contaminated (or second-hand) ND, which is the ND induced by pure ND. ND–II includes superficial ND and other ND types. For static analysis, they built an ND dictionary of C and C++. They then added data structures to store results of ND actions. Code snippets are generated and inserted as functions. For run-time compensation, there is a tradeoff between checkpoint-to-compensate (high bandwidth) and reexecute-to-compensate (high CPU).

Ken and Lorenzo broke in with "How much is the compensation overhead?" The answer is that it depends on the application-level characteristics. For example, with Apache there is no compensation at all. To the question "What is the advantage of your technique over the backup method?" Joseph explained that the backup does not work on multitier systems. In response to "If the compensation falls behind, and the replica is ahead, can the other one catch up or they will be inconsistent?" Joseph said that as soon as the replica is compensated, it is consistent. The concurrency is increased by doing it this way.

Joseph continued with the preliminary evaluation. The tier number is between 2 and 4, with clients between 2 and 4. He showed a graph on experiments with 5% forward and backward ND. The graph displayed that the technique scales well. Another graph on 60% forward and backward ND shows increased overhead. An insight from these results is that lower amounts of ND cause much less overhead. Thus application characteristics will determine the overhead.

Jay Wylie asked whether all tiers get analyzed at the same time or independently. It turns out that you can do it independently. The worst case is that tiers are fully transparent. Ken and Lorenzo asked whether there has been prior work doing replication on ND programs, writing down what the program did, then the backup waiting for the primary to know what to do. The cost seems to be not that great. Joseph added that breaking down the overhead is a subject of future work. In reply to "How does your approach compare with the method where the master decides and the slave asks for the answer?" Joseph said that the slave has to ask for everything in that case, and we don't need that, so future work will aim toward making it more efficient. Miguel Castro asked whether the project does compensation at the same time. It does, and this helps to increase the concurrency of the program. Geoff Voelker asked how to know which ND a function call is going to depend on. Joseph replied that they map calls to different ND.

■ ***Correlating Multi-Session Attacks via Replay***

Fareha Shafique, Kenneth Po, and Ashvin Goel, University of Toronto

Presenter: Ashvin Goel

Typical attack characteristics include low-level or stealthy behavior, small footprint, and multiple sessions. The idea in this paper is to replay the attacks. The basic replay method is to compare outputs with replay run and the original run. But if the replay is nondeterministic, the output could differ. The solution is training using nondeterministic inputs to obtain output statistics. The outlier is classified as the attack.

Ashvin showed the experiments of unit tests on different applications and multisession attacks. The unit test result is a matrix showing the sensitivity of the method to changes in inputs. The multisession result is a diagram with attack multisession and user multisession. There is a great output difference between the attack and the user. In concluding, Ashvin proposes replaying sessions with changed inputs to correlate attacks. Future work includes nondeterministic replay and the case of long-running sessions.

Chris Stewart asked whether it is important that legitimate user outputs are not modified. If you have a false positive, you roll back, and the output is modified. Ashvin explained that false positives do not matter if it is security-critical. For analysis, we are not concerned about 100% correctness. You probably need a human to identify the attack at the end. George Candea asked about the cost of acting on false positives. What if you roll back actions that are really important? Ashvin said, "We don't get too many false positives. False negatives are more important for the work. Because we have roll-back recovery, we can reverse the roll-back, but the cost is huge. It is important not to have many false positives. We still need a human." Another question concerned a paper from USENIX on trying to undo operator mistake and then traveling back in time, but it is hard to redo a change you should not have undone and the cost is high. Ashvin explained that the system has to be offline once you have an intrusion. It depends on how long it takes you to fix it. Jim Thornton's question on what to do if a legitimate change results in a different output was deferred to an offline discussion.

■ ***Automatic On-line Failure Diagnosis at the End-User Site***

Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou, University of Illinois at Urbana-Champaign

Joseph Tucek showed the commonly seen Windows XP error message window that asks you to send an error report, which is mainly the core dump. It is not easy to reproduce a bug with just core dumps. The real insight is that because there is a well-defined set of steps that are generally taken during debugging, why not automate it? The proposed solution is online automated diagnosis. First capture the moment of failure; then run analysis tools only on the relevant portions of the program; finally, au-

tomate the debugging process in a humanlike protocol. For replay and reexecution, they used their checkpoint/reexecute framework in their SOSP '05 Rx paper. For analysis, their project begins with the core dump, which provides an inexpensive starting point. For the example of memory bug detection, it works by monitoring accesses. The analysis process is too expensive for production runs. It has to be modified and inserted during mid-run. It is feasible only because they limit it to recent and relevant execution.

As experimental results, there are three bugs found with failure types correctly identified. Joseph displayed in detail the example of a TAR bug. The take-away message is that a dynamic backward slice can tell a concise story of the problem and that the tool can perform analysis at the end-user side.

Solom Heddaya was interested in the number of times the system failed but no bugs were found. For deterministic bugs, Joseph claimed 100% success rate, but "occasionally, we have to go back more checkpoints, and it is possible that the earliest checkpoint is after the root cause. For the case of nondeterministic bugs, the result is not so good." George Candea said that he thought most bugs are nondeterministic, but the reproducibility varies. Joseph conceded that some nondeterminism comes from the environment, but they are eliminating this source. Ken Birman asked how far one must go in rolling back and checkpoints. Joseph said that, "according to statistics, we go back more when necessary, and give up if you cannot find it far enough." In response to whether backward slicing requires source code, Joseph said it did not and that they use binary instrumentation with the tool PIN from Intel.

Kai Shen stated that there are already many debugging tools out there. So what are the take-home points? Joseph replied that their tools are feasible because of the low overhead. "Debugging is not an art. There is a process we can use." Chris Stewart wondered how the logic here can be different for different systems. "We try to be general," said Joseph. "Java is different. We do not deal with that. If you have more specific knowledge, it would provide much better results."

PRAGMATIC CHOICES FOR THE NEW AGE

Summarized by Avishay Traeger

■ *The Case for Byzantine Fault Detection*

Andreas Haeberlen, Max Planck Institute for Software Systems and Rice University; Petr Kouznetsov and Peter Druschel, Max Planck Institute for Software Systems

Speaker: Andreas Haeberlen

Byzantine Fault Tolerance (BFT) is a well-known technique that is used in distributed systems to mask a bounded number of byzantine faults. BFT incurs large overhead in that it requires $3f+1$ replicas, where f is the number of faults to tolerate, and it does not scale well. This work describes Byzantine Fault Detection (BFD), an alternative approach that aims at detecting these faults, rather than masking them. Whereas detection is not sufficient for irreversible behavior, it is an efficient and scalable alternative for recoverable faults. It can also deter bad behavior. The detection system uses only $f+1$ replicas and requires that only one replica complete a request before returning to the client (rather than requiring that most replicas complete, as in BFT).

Each node in this system has a state machine and a detector. The detector can inspect all messages at the local node. When the detector observes a fault it informs its local application and provides evidence to other detectors. Since the detector only knows about messages on the local node, only observable faults can be detected. In the detector, each action is undeniably associated with the identity of the node that has performed the action, allowing the system to gather irrefutable evidence of faulty behavior. In addition, the detector is complete (finds evidence against faulty nodes whenever faulty behavior is observed) and accurate (does not generate valid evidence against correct nodes).

Byzantine Fault Detection is a good choice for systems with recoverable state, systems already using BFT (to ensure that faults are quickly detected), and systems that span multiple administrative domains. Other considerations include the number of nodes in the system, the delay the nodes can tolerate, and the amount of available bandwidth.

■ *Safe at Any Speed: Fast, Safe Parallelism in Servers*

John Jannotti and Kiran Pamnany, Brown University

Speaker: Kiran Pamnany

Many server applications are multithreaded, with one thread handling each request. Concurrency helps improve performance, but the programmer needs to synchronize access to shared resources, which is error-prone. The programming philosophy presented is that one should start with a serial, correct application and gradually improve parallelism, as opposed to starting with a highly concurrent but buggy program and progressively fixing bugs. In an event-

driven program it is possible to improve parallelism without adding locking: If two handlers do not use the same global variables or contexts, they can run in parallel. This work uses static analysis to determine which handlers should be allowed to run together, and it enforces the constraints at run-time.

The solution uses static analysis to identify aliases, locate event handlers, determine global variables that are read or written by each handler, examine context usage by each handler, and identify system calls made by each handler. When the analysis is complete, any concurrency issues are reported. In cases where static analysis cannot determine if handlers can run concurrently, conservative behavior is used to ensure safety. The analyzer provides detailed feedback to the programmer so that constraints can be removed, either by splitting the handler or by adding an explicit lock. Profiling can help a programmer decide which constraints should be removed.

At run-time, a multithreaded event management library runs handlers concurrently, subject to the constraints generated by the static analysis. Colors and hues are assigned to event handlers: Handlers that may run in parallel are assigned different colors, and those that may run in parallel if and only if their contexts differ are assigned different hues. Two levels of queues are used to schedule event handlers; the first level has one queue for each hue and the second level has one queue for each color. This approach results in a conservative approximation of the constraints, but it enables efficient scheduling without expensive locking.

■ ***Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair***

Val Henson and Arjan van de Ven, Intel Open Source Technology Center; Amit Gud, Kansas State University; Zach Brown, Oracle, Inc.

Speaker: Val Henson

Today, it can take several days to run a file system check (fsck) on production file systems. As disk capacity is growing at a much faster rate than bandwidth and seek time is remaining fairly constant, the situation will only get worse. In addition, as capacity grows, the likelihood of disk errors grows as well. Existing solutions (journaling, copy-on-write, soft updates, etc.) can reduce the frequency with which one must run fsck, but not the duration of the fsck. Finally, an entire file system can fail from a small number of faults. These issues imply that file systems should be designed with repair in mind. Developers can use several techniques to achieve this: They should use on-disk formats that are conducive to repair, use simple on-disk data structures, create optimizations for reading data for repair, allow for fast incremental file system checks, and add features such as checksums, redundancy, and scrubbing.

Chunkfs is a proposed repair-driven filesystem architecture, where the file system is split into several chunks that are as self-contained as possible. Each chunk has its own block number space, allocation bitmaps, and superblock. This allows individual chunks to be fscked, greatly reducing fsck time. Other benefits include being able to change the size of the file system and defragment quickly. Since chunks do not have much common metadata, Chunkfs has built-in multithreaded scalability. It can also allow for per-chunk filesystem formats.

Chunkfs uses continuation inodes to deal with issues such as files, hard links, and renames that cross multiple chunks. To avoid having many of these continuation inodes, chunkfs uses smart allocation and sparse files so that a file has at most one continuation inode per chunk. Because these continuation inodes contain pointers to other chunks, some chunks may need to be checked together. You can find a project page for Chunkfs at http://www.nmt.edu/~val/chunkfs/.

■ ***Towards a Dependable Architecture for Internet-scale Sensing***

Rohan Narayana Murty and Matt Welsh, Harvard University

Speaker: Rohan Narayana Murty

An Internet-scale sensing (ISS) system consists of a large number of geographically distributed data sources tied into a networked framework for collecting, filtering, and processing potentially large volumes of real-time data. The infrastructure is heterogeneous, decentralized, and volatile: Failures are frequent, and the system must be reliable in the sense that it continues to process (possibly incomplete) data. ISS systems need a highly scalable solution for dependability. This work argues that ISS systems should be designed to offer feedback to end users on the fidelity and coverage of the results returned by the system and should make use of simple, lightweight replication techniques.

In many ISS applications, availability is more important than correctness. It is very difficult to guarantee correctness on such a system, and many applications are naturally tolerant to diminished quality of the data. The goal of this work is to provide

mechanisms that mitigate the effects of failures (without diminishing availability) and provide feedback on the quality of answers to the end user. Query results should contain feedback about the fraction of sources represented in the answer, as well as information about the age of the data.

This work has three basic design principles to achieve these goals. First, it uses structured operator replication, which means that more resources are devoted to replications of operators that are higher in the dependency tree since they can lead to larger failures. Second, it uses free-running operators. Operators do not need to maintain consistency with each other, which obviates the need for expensive protocols. However, typical operators have a finite (and often short) causality window that defines the set of past input tuples that affect its internal state, which allows them to eventually return to a consistent state after a failure. Third, it uses best-guess reconciliation to determine the most accurate answer among possible divergent states of the free-running operators. To do so, it can use value-based reconciliation, state-based reconciliation, or a measure of replica divergence.

HIDDEN GEMS (EXTENDED ABSTRACTS)

Summarized by Geoffrey Lefebvre

■ ***Making Exception Handling Work***

Bruno Cabral and Paulo Marques, University of Coimbra, Portugal

Presented by Bruno Cabrel

Exceptions are the standard mechanism for error handling in modern programming languages. Unfortunately, dealing with exceptions is a tedious process. Programmers often avoid the issue by writing empty handlers to save time. Programmers who take the effort to write proper exception-handling code see their productivity impaired. The authors argue that exception handling should not interfere with normal programming tasks but instead become a system issue. In this scenario, the run-time environment provides a set of generic exception handlers and deals with exceptions automatically. The programmer only has to wrap the code with try blocks at the appropriate locations. Because the system may have to try multiple handlers when an exception occurs, this approach requires that try blocks be resumable and appear atomic.

■ ***Speculations: Providing Fault-tolerance and Recoverability in Distributed Environments***

Cristian Tapus and Jason Hickey, California Institute of Technology

Presented by Cristian Tapus

Cristian began his talk by noting that distributed systems are ubiquitous. It is now mandatory that we build safe and reliable systems. Failures are frequent in highly parallel machines. It is critical that systems expected to run for a long time support fault tolerance. Unfortunately, traditional checkpoint mechanisms are application-specific, their implementation is expensive in terms of man-hours, and they are also error-prone.

To address these issues, the authors present a new programming model based on speculative execution. The approach separates fault recovery code from computation. Fault recovery becomes transparent and automated and the design of distributed systems is simplified. Speculations are implemented as an extension to the Linux kernel. The implementation provides system calls to begin, abort, and commit speculative executions. Outbound messages sent while executing speculatively are marked accordingly. A process automatically switches to speculative execution when it receives a message marked as speculative.

■ ***Discrete Control for Dependable IT Automation***

Yin Wang, University of Michigan; Terence Kelly, Hewlett-Packard Laboratories; Stéphane Lafortune, University of Michigan

Presented by Yin Wang

Workflows are programs written in high-level languages and used increasingly for IT automation. These languages can express concurrency, contingency, composition, etc., making workflow programming difficult and error-prone. The authors present an approach based on discrete control theory which provides safe execution of possibly flawed workflows. Their approach uses finite-state automata to represent all execution states reachable from the initial state.

The safety specifications are represented by forbidden states or as regular expressions. The goal is to ensure that the system reaches satisfactory termination without entering forbidden states. A discrete controller is automatically generated offline. The controller dynamically disables controllable transitions based on the current execution state, avoiding transitions to forbidden states when possible. The approach presented allows workflows to be partially decoupled from the dependability requirements.

■ ***SecondSite: Disaster Protection for the Common Server***

Brendan Cully, University of British Columbia; Andrew Warfield, University of Cambridge

Presented by Brendan Cully

Brendan began his talk by noting that disaster can strike at any time. Whether caused by floods, severed power lines, or dinosaurs, site disasters can and do happen. The typical solution is

to teleport your server to a new location by restoring a backup and redirecting traffic using DNS. The problem is that backups are expensive and do not always work and DNS updates can take hours, even days, to propagate.

The solution to this problem is to think under the box and use virtualization. The favored approach is to constantly replicate a primary site by performing a continuous live migration of virtual machines. Virtual machines are never suspended; their memory is simply marked copy-on-write when a snapshot is taken. Brendan stated that one of the major challenges is dealing with replication overhead. This issue can be addressed partially by using delta compression. Another challenge is how to take consistent snapshots of multiple servers. Snapshots of individual virtual machines are taken independently but coordination is required to maintain causality within the global snapshot.

■ *Debate Panel*

All presenters took questions from the audience.

George Candea asked what applications were most amenable to the SecondSite approach. He hinted that databases have large write sets and deal with disaster naturally by shipping their log. Brendan agreed that databases were probably not the best candidate but that there are many existing server applications without built-in recovery mechanisms.

Someone asked Cristian whether programmers would be able to deal with speculations intuitively. Cristian stated that people in other communities, especially in the scientific computing community, are very excited about the idea of speculations.

Christopher Stewart asked Yin Wang, "By leaving the choice to the program, could a program go down a wrong path?" Yin explained that their approach guarantees that programs do not go down forbidden paths.

John Wilkes asked Bruno Cabrel about the metrics used to evaluate his approach. He answered that exception injection could be used as an evaluation tool.

Someone asked a joint question to Bruno and Cristian, since the work of both deals with removing the need for programmers to deal with errors. Are the techniques mutually exclusive? Bruno answered that their intended targets are different. The exceptions framework aims to be a general platform solution, whereas Speculations targets distributed applications. Cristian added that although they differ, the goal is the same: to have cleaner code that is easier to reason about.

Someone stated that if a system magically handles errors, then you have a system that is slightly wrong. How do you reason about this? Cristian answered that this is a similar problem to compiler-generated errors. People do not suspect that their compiler or their operating system is wrong. John Wilkes seems to disagree on the last point. The goal is to increase the level of confidence in the application.

George Candea asked Bruno whether he learned anything surprising from some of the studies on exception handling he cited. The major surprise for Bruno was that the exception-handling code only accounts for 4 to 8 percent. John Wilkes enquired about the applications that were included in these studies. Bruno answered that the studies looked at 16 professional applications such as JBoss.

Petros Maniatis stated that many generic catch blocks will not be acceptable for certain applications. Is there a clean way to override the generic handlers? Bruno explained that the set of recovery blocks can be defined by the platform or the program.

George Candea enquired about the differences between SecondSite and some of the related work from Stanford, especially the Collective. Andrew Warfield jumped into the conversation and stated that the Collective was more about the transport format than the ability to capture instantaneous snapshots of running virtual machines.

Christopher Stewart said that there exists a subculture in the dependability community that believes that exceptions should simply be logged and not handled. He then asked Bruno and Cristian for their opinion on the matter. Bruno answered that this is similar to checked versus unchecked exceptions. He believes that all exceptions should be handled. Cristian said that not handling exceptions can result in violation of program correctness. If you transparently roll back an application without any notification, then the same problem could resurface later and you could end up in a worse state. He believes it is important to report errors to the application. The best approach is to combine the two: Provide availability and report the error. But in the end, no approach will be perfect.