Vigilant—Out-of-band Detection of Failures in Virtual Machines

Dan Pelleg[1] Muli Ben-Yehuda[1] Rick Harper[2]
Lisa Spainhower[2] Tokunbo Adeshiyan[2]

[1]{dpelleg,muli}@il.ibm.com
[2]{reharper,lisa,tokunbo}@us.ibm.com

ABSTRACT

What do our computer systems do all day? How do we make sure they continue doing it when failures occur? Traditional approaches to answering these questions often involve in-band monitoring agents. However in-band agents suffer from several drawbacks: they need to be written or customized for every workload (operating system and possibly also application), they comprise potential security liabilities, and are themselves affected by adverse conditions in the monitored systems.

Virtualization technology makes it possible to encapsulate an entire operating system or application instance within a virtual object that can then be easily monitored and manipulated without any knowledge of the contents or behavior of that object. This can be done out-of-band, using general purpose agents that do not reside inside the object, and hence are not affected by the behavior of the object.

This paper describes *Vigilant*, a novel way of monitoring virtual machines for problems. Vigilant requires no specialized agents inside a virtual object it is monitoring. Instead, it uses the hypervisor to directly monitor the resource requests and utilization of an object. Machine learning methods are then used to analyze the readings. Our experimental results show that problems can be detected out-of-band with high accuracy. Using Vigilant we demonstrate that out-of-band monitoring using virtualization and machine learning can accurately identify faults in the guest OS, while avoiding the many pitfalls associated with in-band monitoring.

1. INTRODUCTION

What do our computer systems do all day? How do we make sure they continue doing it when failures occur? As Verbowski notes in "The Secret Lives of Computers Exposed: Flight Data Recorder for Windows" [19], these questions are both important and extremely hard to answer. Being able to monitor the system, detect anomalies and—if necessary—respond to them is a requirement for any computer system, be it an embedded, consumer, server or super-computer system.

Traditional approaches for trying to answer these questions can be roughly divided into the *in-band* and *out-of-band* approaches. In-band approaches require installing one or more agents that monitor and—if necessary—respond to the changing conditions of the system. Such agents are typically specific to a given operating system and application, and are of limited utility, since they run as part of the workload they monitor. An operating system under stress may well deprive an agent running inside it from the resources it needs to be effective, such as CPU time or memory. An in-band agent is also susceptible to attacks from within the monitored object.

Out-of-band agents, as exemplified by network-based monitoring solutions such as Network Flight Recorder [15], suffer from reduced visibility into the behavior of the system being monitored and limited response abilities. Since they are running on a different machine than the machine being monitored, they can only treat the monitored machine as a black box, and are extremely limited in their ability to respond. Often times only one response is available: do nothing. Other times the response may be limited to informing an administrator or power-cycling the monitored machine, assuming it is equipped for remote power-cycle.

The widespread adoption of virtualization represents an inflection point in the ability to provide improved availability to most computer system users. Using virtualization, it is possible to achieve this without expensive hardware, complicated setup and configuration, expensive consulting contracts, application-specific coding, or continual maintenance and testing of the high availability functionality. This is because the virtualization layer, often called " hypervisor" or "virtual machine monitor", encapsulates an entire operating system and application instance (or, in the case of container virtualization, application containers) within a virtual object that can then be easily monitored, started, stopped, replicated, checkpointed, and restarted locally or remotely, all without knowledge of the contents or behavior of the virtual object. This can all be done out-of-band, by agents that run outside of the virtual objects and communicate with the hypervisor via defined external interfaces.

Of course, with this new opportunity comes a new set of challenges. Traditional high availability technologies rely on intrusion into the behavior of the now encapsulated operating system and application to determine their health, and application-specific recovery procedures are usually employed. While these techniques can of course still be used in a virtualized environment, many of the attractive capabilities outlined above, such as generality, are lost.

What is needed is the ability to determine the health and status of a virtual object based solely on measures available externally to that object, typically provided by the hypervisor. Unfortunately, the richness of these indicators is typically quite limited, for a number of reasons. Visibility into the virtual object may be limited because of security reasons, or the hypervisor may only collect simple performance metrics such as CPU utilization, I/O activity, and memory utilization. This raises a difficult problem, since it is quite difficult to discriminate based on these measures between a virtual object that is performing properly, and one that is quite ill.

This paper describes one approach to out-of-band monitoring that performs this discrimination based on statistical analysis, as implemented in the "Vigilant"[1] system. Our experimental results show that problems can be detected out-of-band with high accuracy. We demonstrate successful identification of hard-to-detect kernel hangs that saturate CPU resources. Hangs of this type will cause a typical load monitor to allocate more resources to the virtual object, and will eventually result in resource exhaustion and degradation of service. Under our analysis, we automatically choose the preferred approach of shutting the object down. Using Vigilant we demonstrate that out-of-band monitoring using virtualization and statistical analysis can equal and even surpass the diagnostic accuracy of in-band monitoring, while avoiding the many pitfalls associated with in-band monitoring.

The Vigilant system is presented in Section 2. Several experiments which were used to validate both the system and the general approach are described in Section 3. Related work is presented in Section 4, future work in Section 5 and our conclusions in Section 6.

2. ARCHITECTURE AND IMPLEMENTATION

Vigilant is a two-stage system. First, we generate a *classifier*, which is a piece of code that can identify faults. Second, the classifier is applied to data from live virtual objects to determine their current state.

In particular, we first collect data from a variety of virtual machines under various loads. This data is then fed to a machine learning process, which outputs the classifier. The classifier is a decision routine that can label each observation, and do so in a manner that is generally consistent with its training data. In the second stage, the system is running in production, and the observations are collected as before. But this time, each observation is fed through the classifier and a label is predicted. If the label matches an actionable condition (such as an imminent system failure), the system can be configured to take the appropriate action. This process corresponds to a discipline of machine learning known as "supervised learning". The supervision comes in the form of the labels that are attached to the initial observations.

In our case, there are two possible labels: normal and faulty. In particular, the fault we focus on is extremely high CPU utilization in kernel space. This may be the result of either a direct programming error leading to an infinite loop, but also of more subtle scenarios. For example, a run queue being constantly re-filled by its service thread, or a failed piece of hardware continuously generating interrupts.

[1]For "virtual guest inspection, learning, and control".

To inject this kind of fault, we wrote a minimal Linux kernel module. As soon as it is loaded, it enters an infinite loop. Loading of the module was timed to be at the beginning of each respective experiment. In Linux, this kind of bug does not obliterate the system's vital signs. The system will still respond to network pings, and continue to provide network functions like serving web pages. However, the system is extremely slow to respond to interactive input, and its throughput is greatly diminished. This behavior makes such a bug interesting to detect. As a specific example, monitors that simply try to retrieve web pages as a sign for the system's health will be ineffective in this case.

Vigilant monitors virtual machines running on top of the Xen [1] virtual machine monitor. For out-of-band monitoring we used the `xenmon` [7,9] utility. `xenmon` collects information about various events, mostly related to the hypervisor scheduler. Specifically, it maintains counters for the following types of information:

Execution count: The number of times a guest virtual machine was scheduled to run.

CPU usage: Utilization of CPU by the guest virtual machine, once it is scheduled.

Time waiting: Time spent in the "runnable" state.

Time blocked: Time spent waiting for an event (such as completion of I/O, or sleep).

Time allocated: Amount of running time allocated to the guest virtual machine by the scheduler.

I/O count: Xen uses "page-flipping" for I/O, a technique in which rather than copy data to or from the I/O partition (dom0) to or from the virtual machines, the pages holding the data are "flipped" between them. The I/O count counts the number of page exchanges between dom0 and the virtual machine.

We can see some of these are approximate counts only (for example, the number of page exchanges is a rather crude estimate of I/O traffic). This is the downside of out-of-band monitoring. But below we demonstrate how even this inaccurate data is sufficient for guest health monitoring.

The classifier chosen for the task was a decision tree. In essence, this is a nested conditional statement, where each test is a threshold test on a particular attribute (see Figure 1). For example, the root node may test the CPU utilization against a threshold value, and its two children correspond to the outcome of the test. Recursively, each sub-tree corresponds to a refined subset of the data. Leaves may occur at any level (i.e., the tree is not necessarily balanced), and are used to label an example. In our setting, a leaf may be labeled either "normal" or "faulty". Training the classifier entails determining, from data, the decision tree's structure, as well as the tested attribute and threshold value for each decision node. For further information about decision trees and their learning we refer the interested reader to Mitchell's "Machine Learning" [13].

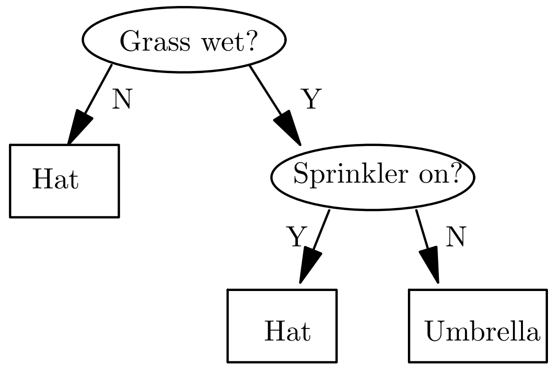

Figure 1: Example of a decision tree.

Once a decision tree is trained, it can be used to classify new, unseen examples. The new observation is tested for the condition in the root node, and depending on the outcome of the test, one of the subtrees is selected. When a leaf node is reached, the label of the leaf is output as this observation's predicted class. In terms of implementation, the procedure is very efficient and easy to code directly from the description of the tree.

It is common practice to prune the tree by recursively removing leaves. The main justification is statistical — a tree that is too tall is likely to model the training data well, but not likely to be good at classifying unseen examples. Pruning is usually implemented as part of the training process. For example, an internal node may have two child nodes. Suppose one child has label "A" and covers 99% of the examples corresponding to th internal node. The other has label "B" and covers 1%. A pruning step may eliminate both leaves, and turn the internal node into a leaf with the unconditional label "A".

Our tree was trained using the standard MATLAB library routine. It was then converted to small utility written in C which runs in the dom0 service virtual machine in real time. The raw readings from `xenmon` are noisy. That is, they might have high variance between two samples, or exhibit occasional spikes and dips. We therefore smooth them by averaging over a moving window of five seconds. In addition, the first five seconds in each experiment were thrown away. To further eliminate false positives, we implemented a "three strike" policy on top of the decision tree. Specifically, the decision tree has to emit three consecutive "faulty" labels before the combined classifier output "faulty". Otherwise, the label "normal" is output.

A decision tree is only one of many possible classifiers. In this space, it has several advantages. First, simplicity in training. Second, the generated tree is easy to interpret by humans. Third, the trained tree is easy to implement and runs efficiently. But it also has problems. In terms of classification power, a decision tree is generally considered as a crude precursor to today's more modern tools (such as support vector machines [6]). Our experience is that for this kind of data, a decision tree is still powerful enough. Therefore we chose to stick with it and enjoy its benefits.

3. EXPERIMENTS

We have experimented with several approaches to the problem of out-of-band detection. In one early experiment, we deployed several virtual Linux instances under the QEMU emulator [2]. Each of them ran a different type of workload (web service, mail service, etc.), and the workloads were timed to begin so as to vary the overall load on the host system. We collected metrics from within the systems using `sar(1)`. We were able to classify, using a simple decision tree, the case where the workloads from different machines strain the host machine's resources — as opposed to the case where only one of the virtual machines is under load. We omit the details of that experiment for brevity, but mention it to show that our approach is applicable in a diversity of settings.

To demonstrate the effectiveness of our approach on a full-fledged virtualized system, we built a test-bench based on Xen. We used a mixture of physical hosts with CPU speeds ranging from 1.4 to 3 Ghz, and memory sizes from 256MB to 2GB. The hypervisors used were Xen versions 3.0.3 and 3.1.0, using para-virtualization and full virtualization, respectively. The guests were running primarily SUSE Linux, as well as proprietary operating systems.

To exert normal workloads, we used the following tools:

iperf: A TCP bandwidth measurement tool [18], which was run in client mode (the server was not monitored).

iozone:: A filesystem benchmark tool [14], configured for the write/rewrite test with file sizes that exceed physical memory.

libMicro: A micro-benchmark library [17], configured to run the "memset" and "forkbomb" tests, which consume large amounts of memory and processes, respectively.

WebSphere: IBM's Application Server, running a benchmark application which simulates a stock trading platform. An unmonitored DB2 server was used on the back end.

Apache: Web-server workload, generated from an unmonitored client running a benchmark tool fetching static pages at high concurrency levels.

idle: Idle workload.

The workload is exerted on a particular guest virtual machine. In about half of the runs, the guest was the only one on its host. In the others, another (either idle or busy) guest was running on the same host.

Each experiment consisted of the following steps:

1. Starting up a guest virtual machine (and potentially starting up any sibling virtual machines).

2. Starting `xenmon`, configured to log all readings about the domain.

3. Exerting the load in question by either starting the benchmark program, starting a benchmark client, or loading the kernel module, as appropriate.
4. Collecting data for about two minutes.

By default, `xenmon` generates readings at 1Hz. This resulted in about 120 readings per experiment. Overall, 106 experiments were conducted, composed of 19 hung systems and 87 normal systems.

To make best use of the available data, we created an ensemble of 50 different trees, each one defined by a random partition of the data. The data was randomly divided into "train" and "test" sets. Given a partition into "train" and "test", a specific tree was created as follows.

Each decision tree was created using only the training set, using the standard MATLAB routine as explained above. Once the tree was grown, each example in the training set was classified by the tree. We repeated the process for each level of pruning, and choose the best level (i.e., the one which yields the highest accuracy on the training set). The tree is then fixed at its optimal pruning level. After the tree is fixed, its performance is measured, this time by classifying (testing) the test set.

Given the full ensemble of 50 trees, we iterated over the samples. For each one, we recorded the fraction of trees which classified it as normal. This is a number in the range $[0, 1]$, referred to as the "prediction value", where values toward zero indicate that more trees classify it as faulty, and the converse for values near one.

Next, we implement an aggregated classifier. It uses the prediction value for each example as above. The prediction is "normal" if and only if the value is above a given threshold. We can now range over the threshold value to get a family of classifiers. A low value will result in many (or all) examples being classified as normal. This translates to many false positives and a high hit rate on the positive examples. Conversely, a high value will classify very few examples as normal, entailing a low false positive rate and a low hit rate. Figure 2 shows, for each possible threshold level, the hit rate versus the false positive rate. This plot is known as the ROC ("Receiver operating characteristic") curve [6][2]. The area under the curve is widely considered to be an indication of a classifier's accuracy. For our ensemble classifier, the ares is 0.94 (out of a maximum of 1), giving a very accurate classifier.

Before (or during) deployment, a particular threshold value has to be chosen. The requirements of the particular setting will determine this value. The ROC curve will help in estimation of the expected accuracy. For example, if system uptime is paramount, even at the cost of burned cycles, high values should be chosen. This will increase the hit rate, at the cost of possible false negatives. Conversely, other constraints may lower the chosen threshold.

[2]ROC curves have a long and varied history, dating back to the attack on Pearl Harbor. See `http://en.wikipedia.org/wiki/Receiver_operating_characteristic` for more information on ROC curves.

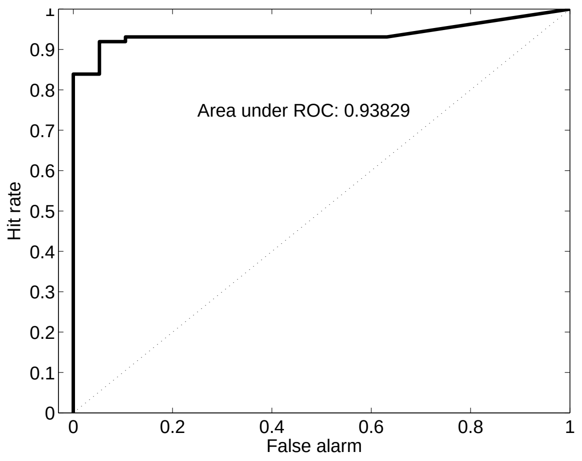

Figure 2: ROC curve for an ensemble of hang detection decision trees.

The lowest threshold value for which there are no false positives is 0.434. If we look at the individual predictions at this value, we find 14 false negatives (normal systems classified as faulty). A closer examination reveals that four of them were running the "forkbomb" workload, which places considerable stress on the system, and the other were running the "memset" workload, which is also a major resource drain. Additionally (and probably less importantly), all were run on multiple-guest systems. Because of the nature of the workloads, we feel flagging them as faulty is still acceptable behavior of the classifier, as the virtual guests would need at least some sort of administrative inspection.

We now give and discuss an excerpt of the final Vigilant decision tree used to detect kernel hangs in virtual machines (Figure 3). First, we check whether `blockedTot` is smaller than some (empirically determined as part of the training process) amount. `blockedTot` is the amount of time the domain spent blocked (sleeping). Intuitively, if the domain spent a lot of time blocked, then it must have been waiting for external resources which it was using to do useful work. Therefore it is probably not hung. Next we check whether the amount of time the domain waited for the CPU (i.e., was runnable but did not run) in the last execution period is bigger than some (empirically determined as part of the training process) amount. Again, intuitively, if the domain is "almost never" waiting for any external events and is "always" ready to run, then it is probably hung. We continue in the same manner down the tree, refining the criteria for "ok" or "hung" in each step.

4. RELATED WORK

Machine learning has been applied to problem determination for complex systems in avionics and energy production and generation. For computer systems machine learning has been applied primarily in the domains of security and performance management. Service management products that aid

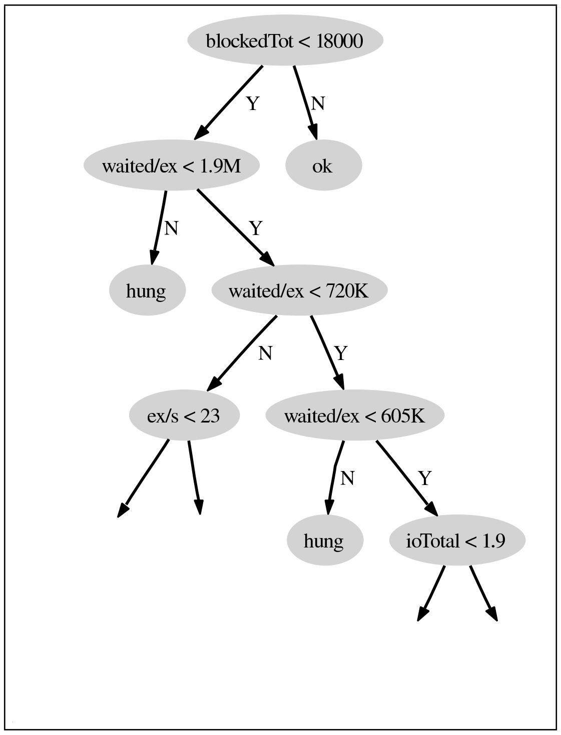

Figure 3: Part of the Vigilant decision tree.

in problem determination, such as IBM's Tivoli [11] or HP's Business Technology Optimization software [10], largely rely on expert systems derived from human input, unlike Vigilant which applies machine learning to computer system problem determination.

Another approach to problem determination, Multivariate State Estimation Technique (MSET), was developed in the early 1990s for proactive detection of online sensor and signal anomalies in space shuttle telemetry data and nuclear power plants. MSET was used by Gross et al. to detect the onset of software aging in commercial computer systems [8] and by Cassidy et al. to identify early signs of shared memory pool latch contention [4]. Similarly to Vigilant, machine learning was used for the detection component of problem determination. Unlike Vigilant, these works were aimed at predicting future critical events thus allowing operational staff to perform actions to decrease or avoid unplanned outage, whereas Vigilant is used to identify existing but hard-to-detect kernel hangs.

In the context of the Recovery Oriented Computing (ROC) project, Chen et al. evaluated automated problem determination and the use of data mining for faulty component identification [5] and Brown et al. used statistical modeling to compute component dependencies to find problem sources [3]. In both of these works the necessary data is provided by in-band rather than out-band monitoring and machine learning is only employed for problem isolation.

Virtualization and machine learning were identified by Kephart et al. as key components of Autonomic Computing [12], but they were not linked or integrated. Vigilant combines virtualization (used to monitor virtual machines out-of-band) with machine learning (used to analyze the monitoring results). The importance of virtualization as a framework for software rejuvenation was explored by Silva et al. [16]. Their work described the detection of software aging in a virtualized environment, and a method for recovery with (usually) no downtime, VM-Rejuv, that employs three virtual machines: active VM, standby VM, and a monitoring VM. Like Vigilant, it operates on VM-level granularity. Unlike Vigilant, VM-Rejuv was aimed at detecting future resource constraints. VM-Rejuv also included an automated recovery scheme, whereas Vigilant is currently limited in its ability to recover from detected faults.

5. FUTURE WORK

This paper has described the use of machine learning to detect that a virtual object has failed, based on externally-visible indicators. This work has focused on detecting a limited set of faults, and should be expanded to being able to detect a much larger class of faults.

This kind of research has been chronically hobbled by the limited availability of labeled data that can be used to train the classifier to detect normal and abnormal operation. One interesting approach to overcoming this lack of data is to automatically detect that a virtual object has failed (essentially, labeling the data) using the hypervisor's indication that the object has crashed, and then performing real-time machine learning on the data obtained prior to the failure to learn how to detect or predict the failure.

It also seems important to enhance the problem detection capability with problem prediction by detecting that the virtual object's parameters are wandering toward a portion of the state space associated with a problem, and perhaps taking proactive measures such as notifying the user that a failure might be imminent, checkpointing the virtual machine in anticipation of the failure, or similar measures. A natural extension is to determine that a virtual machine is having performance problems, and then either adjust resource allocations or outright migrate the problematic virtual object to a physical host that has more resources available.

6. CONCLUSIONS

We have described the Vigilant system, a novel way of monitoring virtual machines for problems. We have shown that our system, which takes an out-of-band approach and knows nothing about the workload running in the virtual machine being monitored, can detect a hard-to-detect failure such as a kernel hang which an in-band agent is likely to fail to detect. Using Vigilant we demonstrate that out-of-band monitoring using virtualization and machine learning can accurately identify faults in the guest OS, while avoiding the many pitfalls associated with in-band monitoring.

7. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] A. Brown and D. Patterson. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Seventh IFIP/IEEE International Symposium on Integrated Network Management, 2001*, 2001.

[4] K. J. Cassidy, K. C. Gross, and A. Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 478–482, Washington, DC, USA, 2002. IEEE Computer Society.

[5] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic, internet services. In *International Conference on Dependable Systems and Networks (IPDS Track)*, 2002.

[6] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, 2000.

[7] R. Gardner, L. Cherkasovah, and D. Gupta. Xen performance monitoring. Xen Summit 2006. `http://xen.xensource.com/files/xs0106_xenmon_brief.pdf`.

[8] K. C. Gross, V. Bhardwaj, and R. Bickford. Proactive detection of software aging mechanisms in performance critical computers. In *SEW '02: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, page 17, Washington, DC, USA, 2002. IEEE Computer Society.

[9] D. Gupta, R. Gardner, and L. Cherkasovah. Xenmon: Qos monitoring and performance profiling tool. Technical Report HPL-2005-187, HP Labs, 2005.

[10] HP. HP Openview. `http://www.hp.com/openview/index.html`.

[11] IBM. Tivoli Business Systems Manager. `http://www.tivoli.com`.

[12] J. O. Kephart. Research challenges of autonomic computing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[13] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[14] W. Norcutt. The iozone filesystem benchmark. `http://www.iozone.org/`.

[15] M. J. Ranum, K. Landfield, M. T. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *LISA '97: Proceedings of the 11th Conference on Systems Administration*, pages 1–8, Berkeley, CA, USA, 1997. USENIX Association.

[16] L. M. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak. Using virtualization to improve software rejuvenation. In *IEEE International Symposium on Network Computing and Applications (IEEE-NCA)*, Cambridge, MA, USA, July 2007.

[17] B. Smaalders and P. Harman. libMicro. `http://www.opensolaris.org/os/project/libmicro/`.

[18] A. Tirumala and J. Ferguson. Iperf. `http://dast.nlanr.net/Projects/Iperf/`.

[19] C. Verbowski. The secret lives of computers exposed: Flight data recorder for windows. *USENIX ;login*, 32(2), April 2007.