In proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), Tokyo Japan, May 12–15, 2003.
http://www.lanl.gov/radiant/pubs/magnet/ccgrid2003.{ps|pdf}

MAGNET: A Tool for Debugging, Analyzing and Adapting Computing Systems*

Mark K. Gardner[†], Wu-chun Feng[†‡], Michael Broxton[†§], Adam Engelhart[†], Gus Hurwitz[†¶]
{mkg, feng, mbroxton, adame, ghurwitz}@lanl.gov

[†] Los Alamos National Laboratory
Los Alamos, NM 87545

[‡] The Ohio State University
Columbus, OH 43210

[§] Massachusetts Institute of Technology
Cambridge, MA 02139

[¶] St. John's College
Santa Fe, NM 87505

Abstract

*As computing systems grow in complexity, the cluster and grid communities require more sophisticated tools to diagnose, debug and analyze such systems. We have developed a toolkit called MAGNET (**M**onitoring **A**pparatus for **G**eneral ker**N**el-**E**vent **T**racing) that provides a detailed look at operating-system kernel events with very low overhead. Using the fine-grained information that MAGNET exports from kernel space, challenging problems become amenable to identification and correction.*

In this paper, we first present the design, implementation and evaluation of MAGNET. Then, we show its use as a diagnostic tool, an online-monitoring tool and a tool for building adaptive applications in clusters and grids.

1 Introduction

The history of high-performance computing has had its share of paradigm shifts driven by the fundamental tradeoff between performance and cost. Monolithic supercomputers, such as the Cray-1, have given way to clusters of symmetric multiprocessor clusters and clusters of PCs in order to leverage commodity or near-commodity components to achieve rapid increases in performance coupled with substantial decreases in relative cost.

Despite this trend, many important computations, e.g., "Grand Challenge Applications" [8], are still beyond reach. Larger clusters yield diminishing returns as absolute cost skyrockets. This dramatically reduces the number of institutions that can afford to build and maintain clusters. Hence, high-performance computing finds itself in the midst of another paradigm shift to *computational grids*.

A computational grid is an aggregate of computing clusters and other resources connected by a wide-area network, such as the Internet. It provides computing power to a diverse community in much the same way that an electrical power grid provides electricity to customers. Middleware, such as the Globus Toolkit [14], provides common abstractions which allow a heterogeneous set of resources to appear as a logical supercomputer. Often the components of a computational grid are acquired and maintained by independent institutions. The institutions participate in order to have more computing power than would otherwise be economically feasible. The TeraGrid Project [15] is an example of the largest computational grid to date.

Each paradigm shift brings new challenges along with new capabilities. Although previous tools can sometimes be adapted, fresh difficulties arise which require that new tools be developed. This is particularly true for tools used to debug, monitor, and analyze computing systems since a paradigm shift often implies that the architecture of the system and its applications have radically changed. In spite of dramatic increases in computational power afforded by the grid architectures, writing, debugging and tuning parallel applications remains a painful task.

A large part of the problem stems from the highly asynchronous nature of distributed computations. Applications written to take advantage of large numbers of CPUs must overlap computation and communication in order to effectively use available resources. Not surprisingly, the causes of performance problems in distributed applications are often distributed, and hence, extremely difficult to identify without global knowledge of the execution history. Furthermore, the causes are often subtle issues of timing that make accurate global histories more important yet more difficult to obtain. A means for monitoring distributed applications and the hosts on which they run is needed.

*This work was supported by the SciDAC Program within the U.S. Dept. of Energy's Office of Science through Los Alamos National Laboratory contract W-7405-ENG-36. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DOE or Los Alamos National Laboratory. Los Alamos Unclassified Report (LA-UR) 02-7170.

Monitoring frameworks for collecting and presenting significant events in the life of a distributed computation are being developed as part of, or in conjunction with, frameworks for writing such applications. For example, Autopilot [11], NetLogger [4] and CODE [12] are monitoring frameworks designed to work with the Globus [14] Toolkit.

One of the challenges faced by monitoring frameworks is the selection of an appropriate level of detail needed to identify a problem. For example, low performance can be caused by messages arriving while their processes are waiting for a time slice. To diagnose this problem, detailed information concerning message arrival times and process scheduling is required. Currently, the *sensors* in the above frameworks are unable to provide this level of detail.

We have developed a tool called MAGNET, **M**onitoring **A**pparatus for **G**eneral ker**N**el-**E**vent **T**racing, which allows the online monitoring of nodes of a cluster or grid. It is a highly efficient mechanism for exporting operating-system (OS) kernel events to user space. Not only can it monitor the OS, but it can also monitor unmodified user applications. The information it provides can be used to develop *adaptive applications*, applications which are aware of the environment in which they execute and can adapt their behavior based on that awareness.

2 Design and Implementation

The primary design goals of MAGNET are transparency to end users in a production environment and high-fidelity, fine-grained monitoring of host events.

MAGNET delivers end-user transparency by incorporating its functionality into the OS kernel where it is available to all applications without modification. As we will show, the overhead of logging an event is so low that MAGNET is also operationally transparent to users.

MAGNET achieves high-fidelity, fine-grained monitoring by allowing any kernel event to be monitored and by timestamping each event with the highest-resolution time source available on most machines, the CPU cycle counter. Optionally, additional information can be exported to give a more detailed look at kernel operation. (For a detailed description of the design and implementation of the previous version of MAGNET, see [1, 5]. See the appendix for a discussion of the differences between the two versions.)

MAGNET is implemented as a patch to the Linux kernel. It creates a circular buffer of event records in kernel memory and provides a function, `magnet_add()`, which inserts an event record into the buffer. A call to `magnet_add()` is placed at each point where kernel information is desired.

In the networking subsystem, we have instrumented the socket, TCP, UDP and IP layers, along with selected Ethernet device drivers. In the task subsystem, we have instrumented task creation, termination and context switching.

To export kernel events to user space for use by applications, we provide `magnet-read`, a program which reads event records from the circular buffer in kernel memory and saves them to disk as a MAGNET trace. We also provide a set of data-analysis tools that translates MAGNET traces into human-readable form for post-processing. These tools provide a simple yet effective way to obtain and utilize kernel information.

	Configuration
1	Linux 2.4.18
2	Linux 2.4.18 w/MAGNET
3	Linux 2.4.18 w/MAGNET, `magnet-read` on sender
4	Linux 2.4.18 w/MAGNET, `magnet-read` on receiver

Table 1. Test Configurations

While `magnet-read` provides a way to read event records from the kernel, it does not allow multiple applications to access the event stream at the same time. In Section 5.1, we describe a more sophisticated tool which facilitates the creation of *adaptive applications* — applications that are aware of their environment and that can adapt to varying resource availability.

3 Performance Analysis

The overhead of running MAGNET is very low. To demonstrate this, we measure the maximum transfer rate between two hosts with and without MAGNET. The extra cycles taken by `magnet_add()`, as well as the overhead of `magnet-read`, show up as a reduced transfer rate.

3.1 Test Setup

A total of four configurations, shown in Table 1, are compared. The baseline configuration consists of two machines with stock Linux 2.4.18 kernels. The second configuration uses the same machines but with MAGNET installed. Although present, MAGNET is inactive, i.e., no event records are read from the circular buffer. The third configuration is the same as the second except `magnet-read` runs on the sender to drain the MAGNET buffer and write the records to a trace file on disk. The fourth configuration is the same as the third, but with `magnet-read` on the receiver instead.

In the results that follow, we configure MAGNET to record socket, TCP, UDP, IP and Ethernet driver (network interface) events. As a result, an application send or receive causes at least four events[1] of 20 bytes each to be recorded. The default 1 MB kernel buffer is used to store event records.

Each of the tests transfers data between two machines containing dual 933 MHz Pentium III processors and 512 MB of RAM. We present transfer rates for both 100 Mbps (Fast) and 1000 Mbps (Gigabit) Ethernet, as well as for uniprocessor and symmetric multiprocessor (SMP) kernels over 1000 Mbps Ethernet.

[1] The precise number of events is $N = 3\left\lceil\frac{\mathrm{Bytes}}{\mathrm{MTU}}\right\rceil + 1$.

	100 Mbps		1000 Mbps	
Conf.	Throughput (Mbps)		Throughput (Mbps)	
1	94.12	± 0.00	615.34	± 0.16
2	94.12	± 0.00	615.00	± 0.84
3	94.12	± 0.00	615.36	± 0.97
4	94.12	± 0.00	613.05	± 0.43

Table 2. Throughput vs. Network Speed

We use netperf [9] on the sender to flood the network in order to measure the achievable bandwidth under worst-case conditions.[2] We minimize the amount of interference in our measurements by eliminating all other network traffic and minimizing the number of processes running on the test machines to netperf and a few essential services. We also conduct enough runs to ensure that the 95% confidence intervals are all less than 5%.

3.2 Performance

Table 2 presents the throughput of the four configurations as a function of the network speed for SMP kernels. Along with the mean, the width of the 95% confidence interval is given. For 100 Mbps Ethernet, the throughput is network limited, as indicated by the high but constant values in all four configurations. (The CPU utilization, as reported by netperf, was below 12%.)

Except when MAGNET is active on the receiver, the results for the 1000 Mbps Ethernet tests are also not statistically significant. Since the amount of processing on the receiver is higher than on the sender, running MAGNET on the receiver shows a greater effect. Even when MAGNET runs on the receiver, the throughput is only reduced by 0.4%. The CPU utilization does not change by a statistically significantly amount in any of the configurations.

Table 3 presents the throughput of uniprocessor and symmetric multiprocessor kernels over 1000 Mbps Ethernet. In general, the cost of mutual exclusion should be lower in the uniprocessor kernel. (Lock contention can still occur due to interrupt handler instrumentation.) Indeed, the throughput of the uniprocessor kernel is slightly higher (+0.3%) than the SMP kernel. On the other hand, the additional load imposed by MAGNET should have a greater effect on a uniprocessor system. The uniprocessor tests 2–4 show a small but statistically significant reduction in throughput compared with test 1, even when MAGNET is inactive. However, the worst-case reduction in throughput (uniprocessor kernel with MAGNET active on the receiver) is still only 3.3%.

In practice, the penalty imposed by MAGNET will be much less since the previous tests attempt to saturate the network to achieve worst-case conditions. To show more typical conditions, we compare the measured throughput on FTP transfers. The server runs wu-ftpd on a MAGNET-

[2]The command is “netperf -n2 -fk -cC -H <host>”.

	Uniprocessor		SMP (dual)	
Conf.	Throughput (Mbps)		Throughput (Mbps)	
1	617.30	± 0.13	615.34	± 0.16
2	614.02	± 0.21	615.00	± 0.84
3	613.33	± 0.46	615.36	± 0.97
4	596.63	± 1.53	613.05	± 0.43

Table 3. Throughput vs. Processors Count

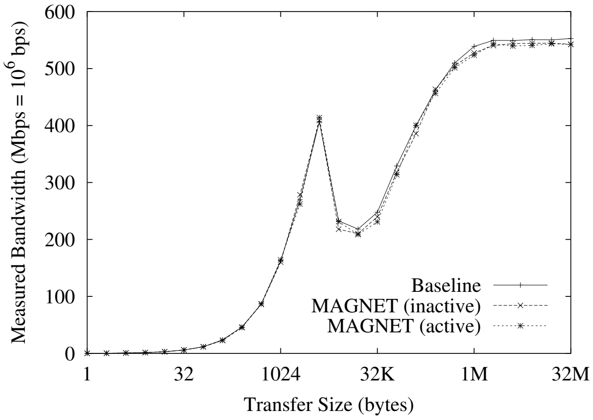

Figure 1. FTP with and without MAGNET

ized uniprocessor kernel. (A uniprocessor kernel accentuates performance differences.) The client machine runs a standard FTP client on a stock kernel.

Figure 1 shows throughput as a function of message size. (The spike at 4 KB is due to the way Linux device drivers allocate memory.) At a message size of 32 MB, the difference in throughput is only 0.8%.

In summary, MAGNET generally imposes a negligible overhead on the maximum achievable throughput. We observe the largest effects on configurations with MAGNET on the receiver or on a MAGNET-ized uniprocessor kernel. In the worst case, which occurred with the MAGNET-ized uniprocessor kernel running on the receiver, the penalty was only 3.3% for a saturated network. In normal usage, however, the overhead is less than 1%.

4 Using MAGNET as a Diagnostic Tool

Due to its low overhead and the fine-grained information it exports, MAGNET makes a very useful diagnostic tool. We give two examples in which MAGNET provides important insights into the operation of the kernel. The first example shows how we uncovered an anomaly in the Linux 2.4 scheduler. The second example shows how we used MAGNET to diagnose a performance problem in a 10-Gigabit Ethernet driver.

4.1 Scheduling Anomaly

We instrumented the process subsystem by adding monitoring points for process creation (fork), scheduling and ter-

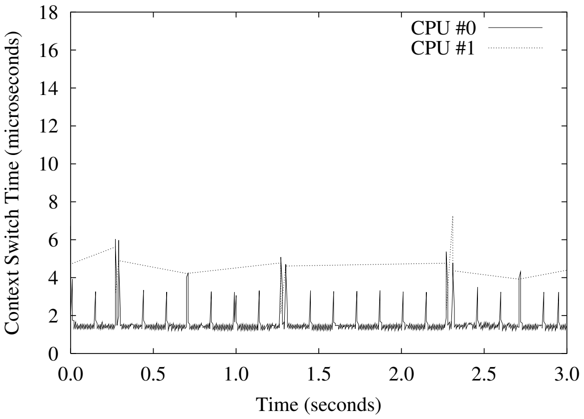

Figure 2. Context Switches "at Idle"

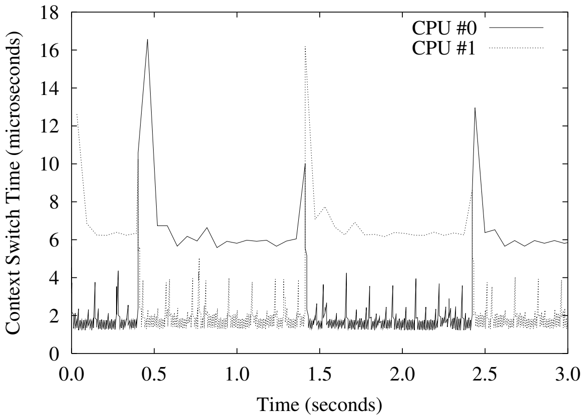

Figure 3. Context Switches "under Load"

mination (exit) events. The modification consisted of 26 lines of code in four files.

The instrumentation generates two events during a reschedule decision. The first event indicates the process being deactivated along with the CPU upon which it was running. The second event indicates the process being activated and gives the CPU upon which it will run. The time difference between the two events is the context-switch time. During that time, the kernel saves the deactivated state, decides which process to execute next and reloads the newly activated state.

To test the instrumentation, we monitored the behavior of the scheduler on a workload containing only the usual system processes (the "idle" test). We also monitored the behavior of the scheduler with a single user process executing a tight infinite loop (CPU-bound test).

Figure 2 shows the context-switch time for the "idle" system test. The average context switch time for CPU #0 is around 2 μs corresponding to the fast path through the scheduler with occasional excursions to approximately 4 μs corresponding to the slow path through the scheduler. The context-switch times for CPU #1, on the other hand, are 4-6 μs likely due to contention on the spin lock protecting the ready queue.

Figure 3 shows the same graph but with the CPU-bound process running. The average context-switch time for CPU #0 now oscillates between 2 μs and 6 μs. The period of the oscillation is around 2 s. Likewise, the average context-switch time for CPU #1 also oscillates but is 180 degrees out of phase.

Unable to explain the behavior of the CPU-bound test, we searched the archives of the Linux kernel mailing list and found that this curious behavior is due to a known problem in the Linux kernel scheduler which causes a process to migrate to idle processors in a round-robin fashion [6]. The problem is being addressed in the experimental 2.5 kernel and will hopefully be fixed by the time the stable 2.6 kernel is released. In this case, the information obtained using MAGNET exposed the scheduling anomaly which would have been very difficult to directly observe otherwise.

4.2 Problem with Ten-Gigabit Ethernet Driver

In this section, we show how we used MAGNET to diagnose and fix a performance problem in a pre-release Linux driver for a 10-Gigabit PCI Ethernet card.

The solid line in Figure 4, labeled "Baseline," shows the measured bandwidth, as a function of the message size, using the pre-release Linux driver. (The results shown are for a standard 1500 byte maximum transfer unit (MTU), although similar results are obtained for various "jumbo frames" up to the largest supported MTU.) Usually the bandwidth increases asymptotically to a maximum value. Indeed, up to a message size of approximately 16 KB, the bandwidth asymptotically increases as expected. However, the bandwidth drops off dramatically after 32 KB and exhibits tremendous variability. The results are repeatable.

We suspected that the problem was in the receiver's protocol stack and inserted a number of `magnet_add()` calls throughout the receiver code. This allowed us to quickly determine that an unexpected amount of time was being spent in the receive handler.

Although not readily visible in Figure 4, we observed periodic drops in performance of varying magnitudes for message sizes that are multiples of 4 KB. Since the minimum unit of network buffer allocation is a 4 KB physical memory page, we suspected a problem in performing buffer allocations during interrupt handling.

High-performance drivers maintain a ring of free buffers so that arriving packets can be moved off the card as quickly as possible. Under heavy load, e.g., when measuring maximum achievable bandwidth, the ring begins to empty. This forces the driver to refill the ring in the interrupt handler rather than wait for a less time-critical moment. We hypothesized that the dramatic performance drop for message sizes over 32 KB occurs because the driver was attempting to refill the ring from within the interrupt handler.

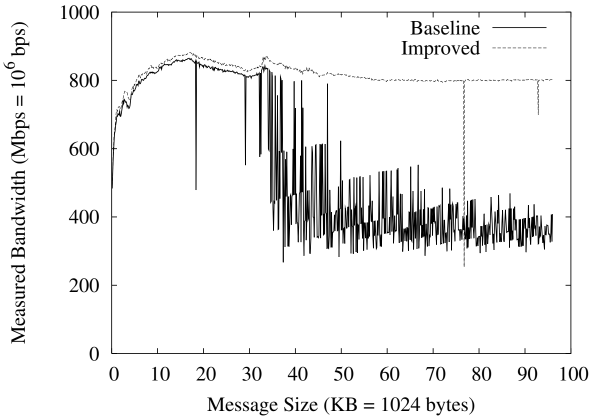

Figure 4. Ten-Gigabit Ethernet Performance

To test this hypothesis we placed `magnet_add()` calls at the beginning and end of the receive handler. We also placed a `magnet_add()` call before the `refill_rx_ring()` function call. At each instrumentation point, we exported the number of free buffers in the parameter field of the MAGNET event record. As suspected, the refill routine was consuming a large amount of time. Furthermore, it was not able to refill the ring completely every time it was called, further suggesting that something was wrong with memory allocation.

The next step was a thorough review of the driver source. We found that the target size of the ring was 1024 buffers. Other high-performance drivers we are familiar with have much smaller ring buffers. We reduced the size of the ring to 64 buffers and ran the test again. The results are given by the dashed line in Figure 4 labeled "Improved".

Not only did the performance improve dramatically for message sizes greater than 32 KB, but the bandwidth also became much more stable. Additionally, the peak bandwidth improved by approximately 2%. Clearly the drop in performance occurred because the driver was having trouble allocating so many buffers. (We are still investigating the reason for the slight falloff in performance after a message size of 32 KB. We are also investigating the cause of the anomalies at message sizes of approximately 77 and 93 KB.) With additional tuning, large MTUs and PCI-X cards, we have seen peak bandwidths of 4.02 Gbps. We are optimistic that the bandwidth can be increased even more by the time the card is ready for production.

To summarize, MAGNET was indispensable in diagnosing the performance problem. Its low overhead did not perturb the behavior we were measuring. It also allowed specific segments of code within a function to be instrumented instead of only providing function-level profiling, as is the case with most profiling tools. Furthermore, it provided a complete event history rather than a statistical summary. Finally, it allowed key kernel data to be exported along with high-resolution timestamps.

5 Online Monitoring Using MAGNET

In this section, we provide two examples of how MAGNET, through a daemon process called `magnetd`, can be used to monitor applications while they are running. In the first example, we show how MAGNET can be used to monitor a distributed application in a computational grid. In the second example, we show how applications can use MAGNET to become resource-aware and adapt to changing conditions. We call these *adaptive applications*.

5.1 The MAGNET Daemon

We have developed a daemon, called `magnetd`, which collects MAGNET events and provides appropriate data to applications. This strategy consolidates all monitoring activity and amortizes the overhead across all applications running on a host. It also facilitates the development of adaptive applications.

While applications can request a copy of the complete event stream from `magnetd`, the amount of information is likely to be overwhelming. `magnetd` has the capability to filter the event stream to just those events an application needs. It can also perform computations on relevant events within the stream, returning results periodically or on demand. As an example, `magnetd` can compute a running average bandwidth for a network connection which an application can query as needed. Alternatively, the application can request `magnetd` to periodically "push" the data without the application's involvement. Finally, we have designed `magnetd` with an extensible architecture so trusted users can augment its behavior without modifying the daemon. (For more details, see [2].)

5.2 Monitoring Distributed Computations

Distributed systems, by nature, are very complex with ample opportunities for subtle bugs and performance problems. The ability to visualize the execution history of a distributed application could potentially save large amounts of time and speed up the development and deployment of such applications. In much the same way as a test engineer uses an oscilloscope to observe the behavior of a complex electronic circuit, MAGNET can be used to observe the behavior of complex distributed computations, such as those found in computational grids.

As a proof of concept, we translate MAGNET event records into the Universal Log Message (ULM) format used by the NetLogger toolkit [4]. This allows us to use the NetLogger Visualization tool (NLV) to view an event stream graphically. The translator establishes a connection with `magnetd`, requests some subset of the event records, translates them to ULM format and appends them to the end of a log file. NLV watches the tail of the file and updates the display as new events appear.

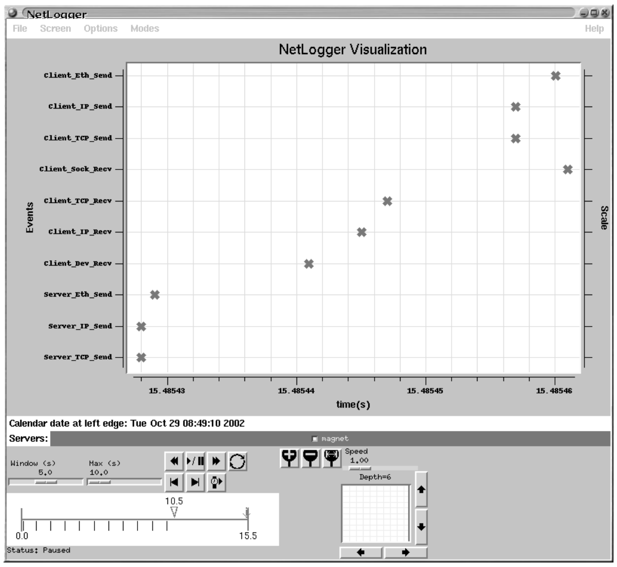

Figure 5. Visualizing FTP Transfer with NLV

We have also developed a translator from MAGNET event records to the format expected by GScope [3], an open-source software oscilloscope library. The GScope library is easily incorporated into applications, which may be more convenient in some circumstances than using a standalone visualization application like NLV.

Figure 5 shows NLV displaying a segment of a FTP transfer with `magnetd` monitoring both the client and the server. The first cluster of three data points shows the server sending a packet through the network stack and out onto the network. The second cluster of three data points shows the same packet traversing the network stack on the client. Three of the remaining four points in the final cluster show an TCP acknowledgment being sent back to the server, while the fourth point shows the client receiving the packet from the socket.

From the figure, we see that the packet traverses the server's stack quickly. (The packet was delayed slightly before being sent by the Ethernet driver.) On the client, however, the packet sat in a buffer for a time until the IP layer was ready for it, after which the IP layer passed the packet on to the TCP layer with less delay. The greatest delay occurred waiting for the client to read the packet from the socket. Visualizing the events of an FTP transfer quickly provides insights into the overall behavior of the FTP transfer, e.g., we identified at least one place, the TCP-socket transfer, where time is potentially wasted.

Although this is a simple example, it serves to illustrate how MAGNET can be used to monitor more complex distributed applications. We are working with the developers of the Autopilot monitoring framework [11] to incorporate MAGNET as a *sensor* (event source). Autopilot is one of several frameworks which can be used to monitor applications executing on top of Globus [14].

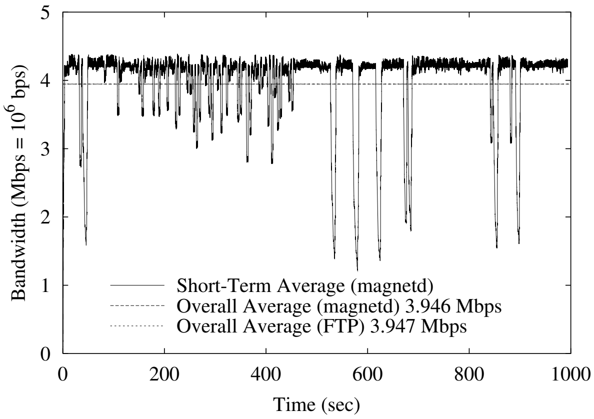

Figure 6. Comparing Transfer Rates

5.3 Adaptive Grid Applications Using MAGNET

Adaptive applications are aware of the environment in which they execute. Because of that awareness, they may choose to adapt their behavior. Within the context of MAGNET, a adaptive application contacts `magnetd` to receive pertinent information about the state of the system. In this section, we discuss a adaptive application which is concerned with the bandwidth that the network can provide.

Consider a distributed visualization application that steers through a large data set. The application consists of a renderer which is co-located with the stored data and a user interface that executes on the scientist's workstation. When the available bandwidth is plentiful, the renderer sends raw frames to the user interface for display. This provides the maximum resolution to the scientist. If the network becomes congested, the renderer reduces the frame rate or compresses the data to provide better response times. The key capability for the application to respond appropriately is awareness of what bandwidth is available from the network. MAGNET, through `magnetd`, provides this capability.

To test the accuracy of `magnetd` bandwidth measurements, we compare the transfer rate reported by `magnetd` to the transfer rate reported by FTP on a 467 MB file transfer from `ftp.debian.org` to our facility. Besides the results for the complete transfer, Figure 6 also shows the short-term average transfer rate, as reported by `magnetd`, during the transfer. (The short-term average transfer rate is computed over a sliding window of 1000 socket receive events on the FTP data connection.)

The overall transfer rate reported by `magnetd` (3.946 Mbps) is nearly identical to that reported by FTP (3.947 Mbps). The slight difference occurs because FTP computes the average over the time interval in which data is being transfered while `magnetd` must wait for the socket to be closed before it knows that no more data will be sent. Hence `magnetd` will always underestimate the transfer rate slightly.

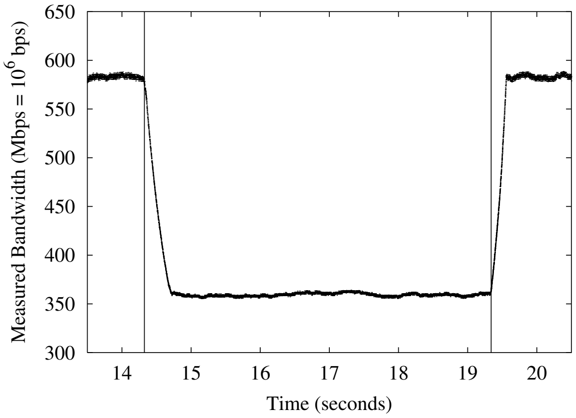

Figure 7. Bandwidth During Congestion

Figure 6 also shows that although the short-term average transfer rate is consistently over 4 Mbps, there are times when the network is obviously congested and the transfer rate drops. While the FTP client does not adapt to fluctuating bandwidth availability, the distributed visualization application would reduce the frame rate or increase the compression ratio in order to maintain responsiveness.

We next test the sensitivity of the bandwidth reported by `magnetd` to abrupt changes in network load. We simulate the transfer of frames from the renderer of the visualization application to the display using `netperf` as the source of network traffic. We simulate the adaptive subsystem by a program which queries `magnetd` periodically and writes the short-term average bandwidth to disk. The test begins with an uncongested network. Another bandwidth-intensive application, also represented by `netperf`, starts 14.3 seconds into the test to cause congestion. The second application finishes 5 seconds later after which the distributed visualization application once again has uncontested access to the network.

Figure 7 shows the short-term average bandwidth, reported by `magnetd`, as a function of time. The bandwidth is initially 583 Mbps without network congestion. At a time 14.3 seconds into the test, the bandwidth-intensive application starts. The bandwidth of the visualization application drops exponentially as the other application goes through slow start. After the bandwidth intensive application terminates 19.3 seconds into the test, the measured bandwidth returns to what it was before.

Based on these encouraging results, we are working with visualization researchers at our facility to modify one of their distributed visualization applications to use `magnetd`. The application will adjust the rate at which it progressively refines images in order to adapt to changing network conditions. When bandwidth is plentiful, full resolution images are transmitted. When bandwidth is scarce, lower resolution images are initially sent, then gradually refined to full resolution.

6 Related Work

Computational grids and other large distributed computing systems provide unique monitoring and tuning challenges. Tools, such as Autopilot [11], CODE [12], Supermon [13] and NetLogger [4] provide the infrastructure to collect, correlate and present information about the state of a computation and the resources being used. Additionally, Autopilot and CODE provide a decision-making infrastructure to modify the behavior of a distributed application based on measurements. Finally, Autopilot (through the Pablo toolkit [10]) and NetLogger (through NLV) provide tools to visualize system behavior.

Rather than providing the infrastructure for collecting, correlating or visualizing information about the behavior of distributed applications, MAGNET provides an infrastructure for *recording* kernel events that occur while an application executes. When coupled with `magnetd`, it is analogous to the *sensors* which the above tools use to obtain the state of a node.

MAGNET provides greater detail about the state of the system than the sensors provided with the above tools. Rather than coarse-grained or aggregate statistics, MAGNET records the occurrence of each event. Its timestamps are also several decimal orders of magnitude more accurate. If desired, the highly detailed event stream can be reduced, using `magnetd`, to aggregate statistics. Thus, MAGNET is complementary to the above tools.

Finally, Paradyn [7] can be used to monitor kernel events. It accomplishes this task by parsing the binary executable and dynamically inserting jumps to code segments called *trampolines* into the running kernel. The trampolines perform the monitoring, as well as execute the instructions overwritten by the jump instruction. In contrast, MAGNET instrumentation is statically compiled into a kernel. Because of this, Paradyn is much more flexible. The cost of the flexibility is the need for more sophistication on the part of the user. Unless the extra flexibility of Paradyn is needed, MAGNET is likely to be of more immediate use to the average grid user.

7 Future Work

MAGNET provides basic infrastructure for monitoring events in the operating system kernel. We are working to get MAGNET accepted into the official Linux source tree. With MAGNET incorporated, we expect that more kernel subsystems will be instrumented. We also intend to instrument other subsystems as we need them and as time permits.

Although we have shown that `magnetd` can be used to create adaptive applications, we do not have sufficient domain-specific knowledge to create more than toy applications in many domains. We welcome experts in all areas to use MAGNET to develop adaptive applications.

We would like to incorporate support for MAGNET, through `magnetd`, into distributed monitoring and decision middleware for computational grids. In particular, we are looking into using `magnetd` as a sensor for the Autopilot and CODE frameworks. We have already implemented a translator which allows MAGNET traces to be used with NetLogger's visualization tool (NLV).

8 Conclusion

Along with increased capabilities, computational grids also give rise to new difficulties in debugging, monitoring and analyzing distributed systems. The MAGNET toolkit facilitates online monitoring of nodes in computing clusters or computational grids with low overhead, without modification to distributed applications.

In the absolute worst case (network saturation), throughput is reduced by less than 4% while monitoring events throughout the network stack. With a more typical workload, throughput is reduced by only 0.8%. Even lower overhead is possible by monitoring fewer events since it is unlikely that grid applications will need to monitor within the network stack.

Because of the variety and quality of the events it exports, MAGNET is a useful diagnostic tool for identifying and correcting subtle intra- and inter-host performance problems. It also enables the development of adaptive applications, e.g., grid applications that adapt to changing conditions such as fluctuating bandwidth.

References

[1] W. Feng, J. R. Hay, and M. K. Gardner. MAGNeT: Monitor for application-generated network traffic. In *Proceedings of the 10th International Conference on Computer Communication and Networking (IC3N'01)*, Oct 2001.

[2] M. K. Gardner, M. Broxton, A. Engelhart, and W. Feng. MUSE: A software oscilloscope for clusters and grids. In *Proceedings of the 17th IEEE Internation Parallel and Distributed Processing Symposium (IPDPS 2003)*, Apr 2003.

[3] A. Goel. Gscope: A software oscilloscope library. `http://gscope.sourceforge.net`.

[4] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A toolkit for distributed system performance analysis. In *Proceedings of the IEEE Mascots 2000 Conference (Mascots 2000)*, Aug 2000.

[5] J. R. Hay, W. Feng, and M. K. Gardner. Capturing network traffic with a MAGNeT. In *Proceedings of the 5th Annual Linux Showcase and Conference (ALS'01)*, Nov 2001.

[6] M. Kravetz. CPU affinity and IPI latency, Jul 2001. `http://www.uwsg.indiana.edu/hypermail/linux/kernel/0107.1/0770.html`.

[7] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. In *IEEE Computer*, pages 37–46. IEEE, Nov 1995.

[8] National HPCC Software Exchange. `http://www.nhse.org/grand_challenge.html`.

[9] Netperf. `http://www.netperf.org`.

[10] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the IEEE Scalable Parallel Libraries Conference*, Oct 1993.

[11] R. L. Ribler, H. Simitci, and D. A. Reed. The Autopilot performance-directed adaptive control system. In *Future Generation Computer Systems, special issue (Performance Data Mining)*, Sep 2001.

[12] W. Smith. A framework for control and observation in distributed environments. Technical Report NAS-01-006, NASA, Jun 2001.

[13] M. Sottile and R. Minnich. Supermon: A high-speed cluster monitoring system. In *Proceedings of IEEE Cluster 2002*, Sep 2002.

[14] The Globus Project. `http://www.globus.org/`.

[15] The TeraGrid Project. `http://www.teragrid.org/`.

A Appendix

Version 1.0 of the toolkit, entitled "**M**onitor for **A**pplication **G**enerated **Ne**twork **T**raffic" (MAGNeT) [1,5], was created to enable the capture of an application's network requests. Because the toolkit can monitor any kernel event, the name has been changed in version 2.0 to "**M**onitoring **A**pparatus for **G**eneral ker**N**el **E**vent **T**racing" (MAGNET) in order to emphasize the general purpose nature of the toolkit.

Version 2.0 is an extensive rewrite. The most important change ensures correct operation from within interrupt contexts and on symmetric multiprocessor (SMP) machines. A SMP-safe spinlock now protects the critical section which adds events to the kernel buffer.

The binary trace format now contains more information needed to make use of the trace. For example, the second (pseudo) event contains the number of seconds since epoch allowing timestamps to be correlated with wall clock time.

More instrumentation points are now available. The UDP layer, generic network device dispatch code and selected Ethernet device drivers are instrumented. Context switches are also instrumented.

Finally, kernel configuration of MAGNET is much easier. All options can now be individually configured at compile time. (For performance reasons, we are resisting the urge to make MAGNET configurable at run time through `sysctl()` or the `/proc` file system.) The configuration options have been relocated from the "Network Options" section to the "Kernel Hacking" section in keeping with MAGNET's ability to monitor any kernel event. The MAGNET source code has also been relocated from `kernel/net/magnet` to `kernel/magnet`.