A Framework for Analyzing Linux System Overheads on HPC Applications*

Sushant Sharma, Patrick G. Bridges, and Arthur B. Maccabe
Department of Computer Science
University of New Mexico

July 18, 2005

Abstract

Linux currently plays an important role in high-end computing systems, but recent work has shown that Linux-related processing costs and variablity in network processing times can limit the scalability of HPC applications. Measuring and understanding these overheads is thus key for future use of Linux in large scale HPC systems. Unfortunately, currently available performance monitoring systems introduce large overheads, performance data is generally not available on-line or from the operating system, and the data collected by such systems is generally coarse-grained. In this paper, we present a low-overhead framework for solving one of these problems: making useful operating system performance data available to the application at runtime. Specifically, we have enhanced Linux Trace Toolkit(LTT) to monitor the performance characteristics of individual system calls and to make per-request performance data available to the application. We demonstrate the ability of this framework to monitor individual network and disk requests, and show that the overhead of our per-request performance monitoring framework is minimal. We also present preliminary measurements of Linux system call overhead on a simple HPC.

1 Introduction

Scientists use high-performance computing (HPC) systems for the performance benefits offered by these systems—they expect these systems to help them in solving large problems,

*This work supported in part by the Los Alamos Computer Science Institute under subcontract R7A82H from Rice University

efficiently and in less time. Unfortunately, operating system interference has recently been reported as limiting application scalability on large-scale machines. For example, ASCI Q at Los Alamos National Laboratory was taking twice the expected time to complete some benchmarks (SAGE [11]) because the operating system was interrupting the benchmark from time to time, resulting in serious performance degradation. Similarly, ASCI White at Lawrence Livermore National Laboratory have shown serious performance problems in the past because of operating system issues; identification of these problems was a difficult task [9, 17]. Researchers spent months finding the exact cause of these performance problems. Other work has shown that applications run on Linux sometimes scale worse than those run on specialized lightweight operating systems [3, 4].

Because of this, understanding and analyzing operating system costs in high-performance environments is imperative to improving system scaling and performance. Only by knowing system performance details is it possible to know the area in which efficiency of system can be improved. Unfortunatley, most existing performance monitoring tools either fail to provide detailed performance information or the price for the information in terms of system overhead is very large. Tools like Ganglia[12] and Supermon[18], for example, provide system-wide performance information but do not provide per-request performance information that would allow for detailed analysis of changing operating system costs. Even more detailed tracing systems like the Linux Trace Toolkit (LTT) [19] also fail to provide per request performance information.

In this paper, we present a framework for monitoring the costs of individual operating system networking and disk system calls from the interrupt announcing data arrival to the delivery of data to the application. We also show that this framework introduces minimal monitoring overhead for HPC applications as well as traditional single-node benchmarks. Finally, we present preliminary information on the variation in performance of Linux system calls when running a simple HPC benchmark.

2 Monitoring Framework

To gather information on HPC-related Linux kernel requests for later analysis, we developed a monitoring framework for gathering the performance of individual system calls. The overall goal is to have each monitored system call return, as part of the system call, metadata describing where time was spent in the system call. In addition, this must be done without large costs in terms of monitoring overhead, overhead, latency or bandwidth.

2.1 Monitoring Architecture

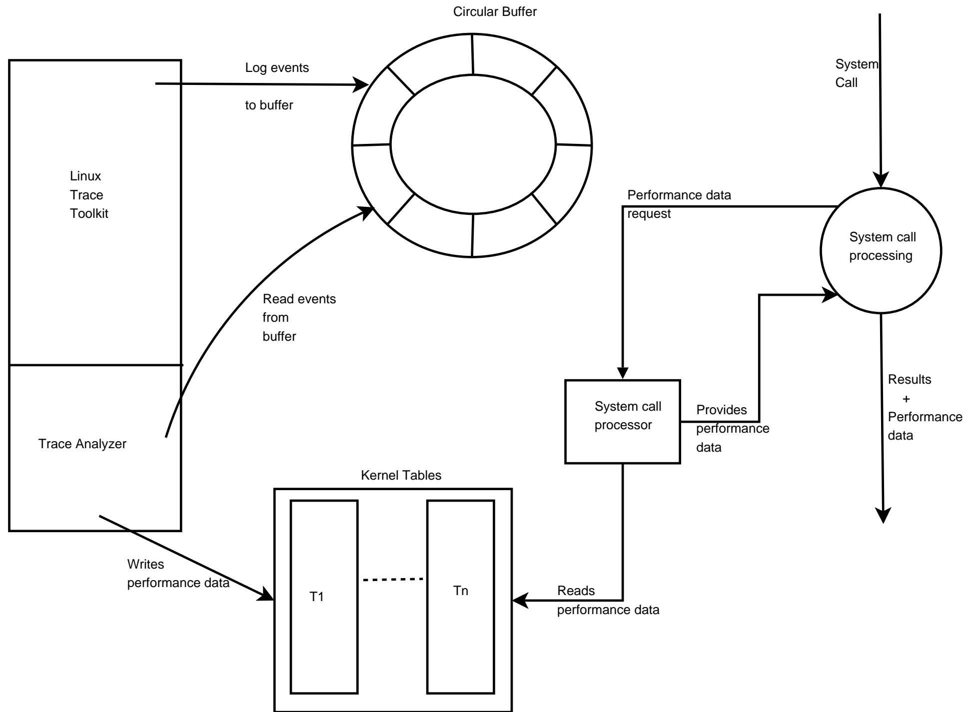

Figure 1: Architecture.

Figure 1 shows our basic architecture which is built around runtime management of operating system trace data gathered using a system such as the Linux Trace Toolkit(LTT)[19]. The basic idea is that this tracing tool will collect various events from the kernel during the processing of a system call and log them to memory. At the completion of a system call, a *system call processor* processes system-call-related performance costs and returns a summary of where time was spent in that system call along with the normal system call results.

To limit the memory overhead of tracing, we assume that, as is the case in system like LTT, traces are written to circular buffers [20]. In such a system, however, events may get overwritten from time to time. To avoid loss of the events in our architecture, a *trace analyzer* is responsible for processing the circular trace buffer as it fills and putting summary performance information about system activities into global kernel tables that are later read by the system call processor. The trace analyzer is also explicitly run by the system call processor to analyze partially filled buffers prior to returning performance information about a system call.

2.2 Monitoring Implementation

The basic approach for implementing this framework uses LTT in flight recorder mode. This section describes implementation details along with the following challenges we faced during implementation.

1. There was not sufficient support in Linux kernel to enable tracing of individual packets arriving from network. It was required to add this support.

2. The LTT was not tracing the events required for generating detailed per-request information. It was required to add more and modify some existing events.

3. There was a synchronization problem faced by the trace analyzer while reading the events from circular buffer.

The following subsection discusses the first two challenges and the third challenge is discussed in next subsection which describes trace analyzer in detail.

Gathering Kernel Trace Information. As mentioned above, our current monitoring framework implementation is built using the Linux Trace Toolkit (LTT) to gather traces for later analysis. LTT continuously logs explicitly recorded events happening in the kernel to a preallocated circular buffer. Our framework uses LTT in *flight recorder mode*, a mode in which LTT continuously writes the events happening in the kernel into the preallocated circular buffer. To facilitate external processing of these buffers, we added a callback mechanism to LTT that causes it to invoke an in-kernel callback whenever the flight recorder buffer becomes full.

While LTT is sufficient for gathering information about many kernel activities, it did not initially track which network packet was being processed during each event, and Linux did not give network packets unique identifiers to allow them to be tracked and associated with specific system calls that later delivered the data in these packets. In addition, LTT did not include sufficient event logging to track the lifetime of packets through the kernel networking stack. To address these problem, we added a unique identifier to each Linux `skbuff` and event logging to the network stack that logged the buffer identifier and associated kernel event logging at packet interrupt, queuing, and delivery.

Trace Analyzer. The trace analyzer is implemented in the kernel as the full buffer callback target for LTT and as an explicitly callable entity for the system call processor. When called, the trace analyzer scans the current LTT buffer, extracts performance data from the buffer and writes it into the appropriate global kernel-table. This allows the system call processor

to later retrieve relevant system call performance information after the reuse of a tracing buffer.

However, while trace analyzer is reading from the circular buffer, a synchronization problem can happen between trace analyzer and LTT. Figure 2 shows a possible scenario which can create a synchronization problem. It shows that an IRQ can preempt the writing of the last event in the circular buffer. When this happens, LTT will try to log an IRQ event in the buffer and due to absence of enough space in the buffer, LTT will invoke trace analyzer. As a result, trace analyzer will try to read from an incomplete buffer where the data is not yet written.

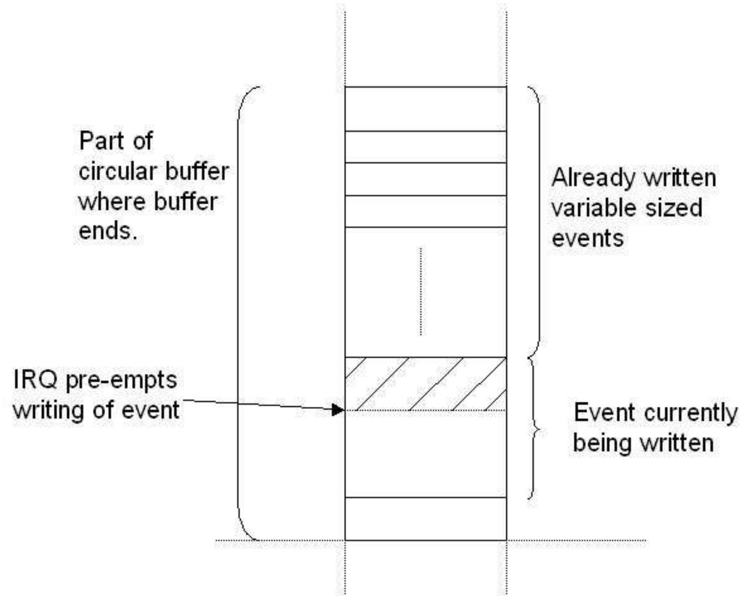

Figure 2: Synchronization Issue.

Figure 3 shows the process of avoiding this synchronization problem within our framework. Because we are currently running in a non-SMP kernel, our current implementation avoids this situation by delaying the invocation of trace analyzer until the receipt of a non-irq event. This non-IRQ event ensures that the last event in the buffer is completely written and it is now safe to read the last event from the buffer. For events received between a buffer filling and another non-IRQ even being logged, a separate extended buffer is used to temporarily held events logged from within an interrupt context. When run, the trace analyzer then not only scans the filled buffer but it also scans the extended buffer.

There are various other possible ways of solving this problem which we discarded. For example, when LTT encounters an IRQ event as the last event, it could invoke the trace

analyzer and have it scan the circular buffer excluding the last event. However, while reading the circular buffer, the trace analyzer has to create performance data and allocate memory for global kernel tables, which is an expensive operation. Because we want to exit interrupt context as soon as possible, we discarded this solution. In the approach which we followed, memory is still allocated, but the allocation is much less and faster. Another approach could be to start scanning the circular buffer before it is completely filled by setting a high water mark. In this approach if we encounter an IRQ event on reaching the water mark, we will still be having the space in the circular buffer to log the IRQ event and then scan the buffer on next non-IRQ event. But this approach alone will fail as we can encounter a series of IRQ's which can fill the remaining circular buffer and then we again have to fall back on our approach to solve the problem.

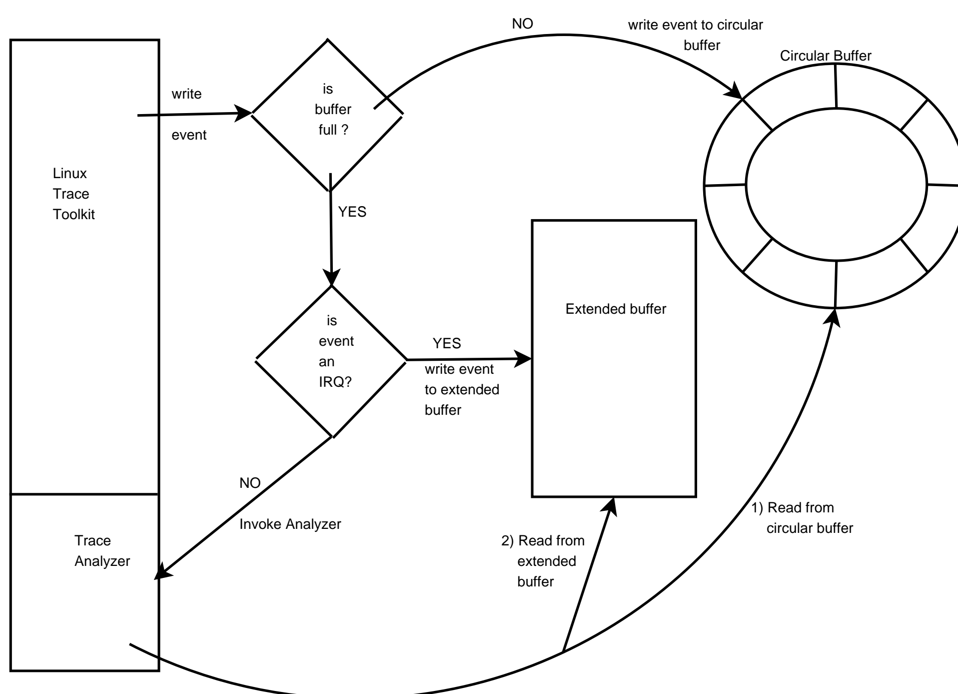

Figure 3: Avoiding incomplete read of last event.

As the trace analyzer processes traces, it records information about asynchronous activities, for example packet processing, in global kernel tables, with one global kernel table for each source of kernel activity, for example network processing or disk activity. The size of these tables grow and reduce dynamically as the information is added or deleted from them. As mentioned above, information is added by the trace analyzer. Stale information is deleted by the system call processor as it handles system calls.

System Call Processor. The system call processor (SCP) is used by modified versions of system calls for getting associated performance data. The SCP knows which kernel-table to query in order to get the required performance data for the system call which is asking for the performance data associated with it. When a system call is about to finish, the SCP invokes the trace analyzer to process any partially filled trace buffers, extracts and removes any system call-relevant information from the appropriate global table, and returns a summary of the performance of the system call to the user. We have currently implemented enhanced versions of two system calls and their associated kernel code, the `recvfrom` and associated networking calls and the disk-relevant portions of the `read` system call.

Status Currently the framework is implemented in Linux kernel version 2.6.3 and will soon be available as a kernel patch. The patch will need to be applied to the kernel already patched with Linux Trace Toolkit.

3 Available Measurements

Using our per-request performance monitoring framework, a wide variety of potential performance-degrading operating system costs can be measured. Tables 1 and 2 show examples of the performance information returned by the modified version of `recvfrom` and `read`.

Type of Event	this process		other processes	
	Count	*Time (usecs)*	*Count*	*Time (usecs)*
Page Fault's	3	17	0	0
IRQ's	14	60	0	0
Process Scheduling	1	15	60	100
Packets Read	16	*N/A*	12	*N/A*

Total time taken by `recvfrom` *= 3819294 usecs*
Time spent in waiting for first packet = 3818893 usecs

Table 1: Example Data Available from `recvfrom`

As can be seen from these tables, our framework is capable of monitoring network device interrupt and scheduling costs; these were the sources of performance degradation and instability on ASCI Q and ASCI White. A tool such as ours would potentially be very useful in such system for discovering such costs.

Type of Event	this process		other processes	
	Count	*Time (usecs)*	*Count*	*Time (usecs)*
Page Fault's	4882	27063	0	0
IRQ's	70	621	1070	8450
Process Scheduling	212	101830	230	4128

Total time taken by `read` in reading 64 MB file = 917069 usecs

Table 2: Example Data Available from `read`

4 Monitoring Overhead Measurements

Because large performance degradations from our framework would limit its usefulness to studying HPC OS performance, we have spent substantial effort quantifying the impact of our framework on the performance of a variety of microbenchmarks, single-processor application benchmarks, and MPI benchmarks. We ran our tests between two 2.2GHz Intel XEON machines with 512MB of memory and Intel EEPRO 1000 Gigabit ethernet cards configured to use MPICH 1.2.6 with the p4 device. We considered 3 different configurations of the Linux kernel to compare.

- Linux kernel version 2.6.3 with no event tracing. (cfg1)
- Linux kernel in which LTT event tracing is enabled but without implementation of our framework. (cfg2)
- Linux kernel in which our framework is implemented along with tracing of events by LTT. (cfg3)

4.1 Simple Benchmarks

To establish lower and upper bounds on the overhead associated with per-request performance monitoring, we ran single-machine computational benchmarks and several kernel-intensive communication benchmarks. To establish a lower bound for overhead, we ran 15 of the SPECINT99 and SPECFP99 benchmarks. On these benchmarks, which are not kernel-intensive, we observed benchmark slowdowns between 0% and 2.75%.

We measured the likely upper bound of monitoring overhead on computation and communication using an MPI ping-pong program and a computational program that does a fixed amount of work while communication is in progress. We then used these programs

to measure the overhead communication imposes on computational processes in the three kernels and the effective MPI bandwidths and latencies achievable by MPICH using these three kernels. The same MPI ping-pong program was also used to calculate overhead in terms of bandwidth and latency.

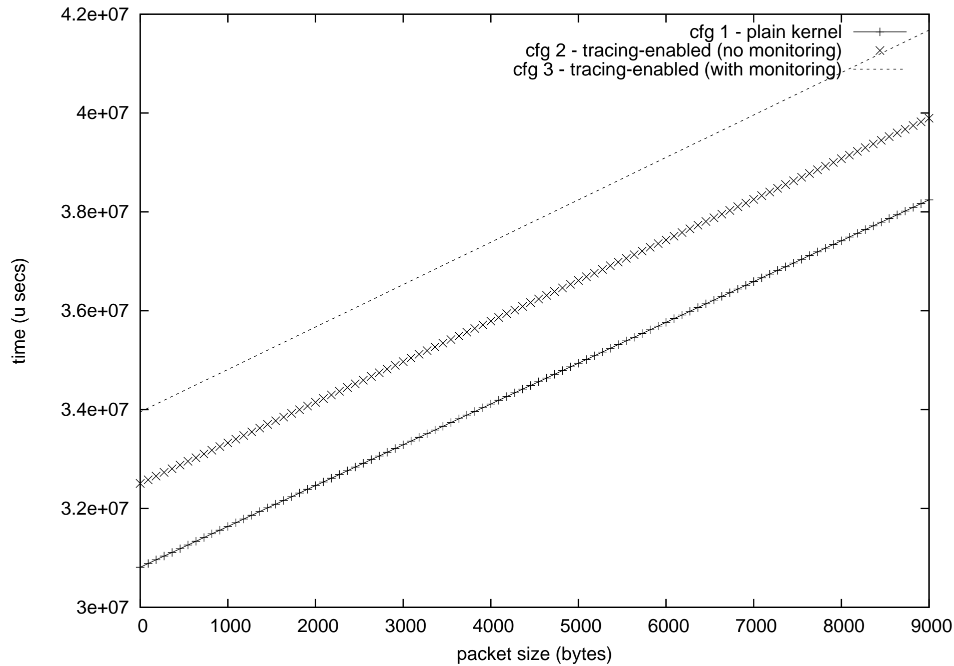

Figure 4: Overhead Measurement

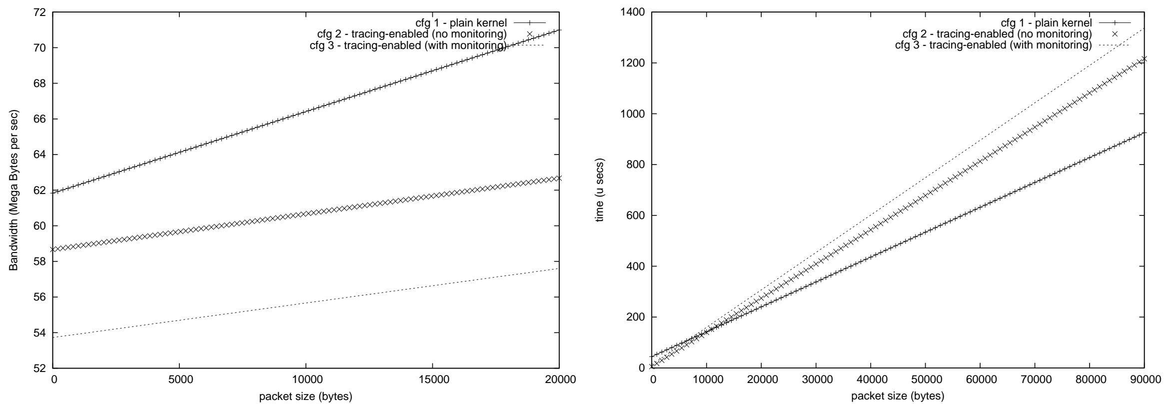

Figure 5: Latency and Bandwidth Measurement

These results show the overhead for doing fixed amount of work to be about (4-5)% above that of LTT and about (9-10)% total. The overhead in termas of latency and bandwidth is approximately the same. Note that this represents worst-case application overhead which

can be incurred by any application as it almost entirely communication-bound. As will be shown in the next section, applications that include a more substantial computational component in addition to network communication, for example standard HPC applications, do not incur such a substantial overhead.

4.2 NAS Parallel Benchmarks.

To measure the overhead our monitoring framework imposes on MPI applications, we ran several NAS parallel benchmarks [6] between a pair of machines modified to include our above-described per-request monitoring framework. Because these benchmarks involve less network communication than ping-pong tests and more communication than the SPEC Benchmarks, they should represent the average-case slowdown of our monitoring framework. Table 3 show that the overhead incurred by these benchmarks is less than 4%, with the large majority of the overhead coming from the system tracing framework. On larger systems with more substantial communication requirements, these numbers will be higher, but should generally be less than the communicatio-intensive ping-pong test described in the previous section.

Name of Benchmark	Class	Time(secs) in different configuration types		
		Linux (secs)	*Linux+LTT (secs/%slowdown)*	*Linux+LTT +Per-request (secs/%slowdown)*
CG	B	196.18	202.65/3.3%	203.68/3.8%
LU	B	701.08	705.94/0.7%	710.51/1.3%
EP	B	239.94	240.69/0.3%	240.59/0.3%
MG	B	28.63	29.68/3.7%	29.70/3.7%

Table 3: Runtime and Percent Slowdown of selected NAS Parallel Benchmarks with and without per-request monitoring

5 System Overheads on a Simple HPC Benchmark

Because UNIX operating system overheads have recently been seen as a major factor effecting application scalability and performance, we have run a preliminary test of UNIX operating system overheads. We changed the mpich 1.2.6 p4 device to use our modified `recvfrom` system call and to save and record the amount of time spent in `recvfrom()` system call processing. Figure 6 shows the distribution of `recvfrom` system call times over the course of

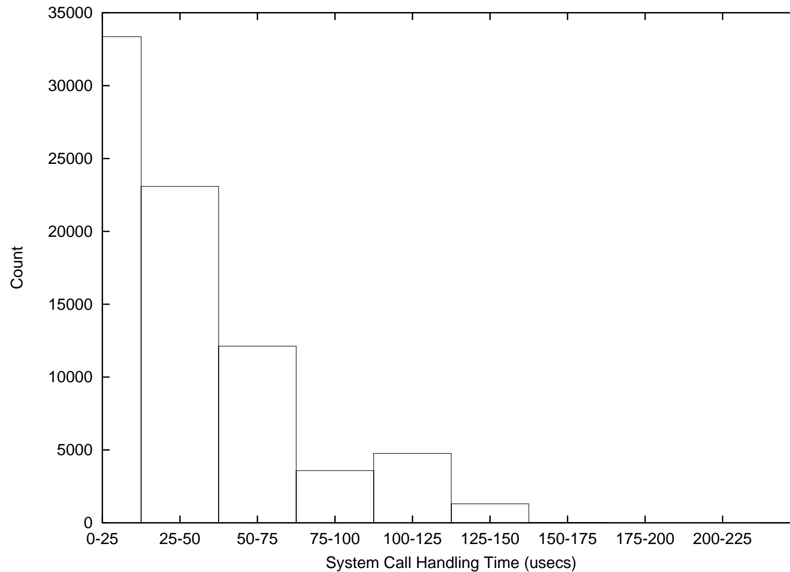

Figure 6: Distribution of Linux Networking Times, CG-B Benchmark

running the class B NAS CG benchmark. The large majority of system calls take between 0 and 25 microseconds, with a rapid falloff out through 125 microseconds. Most of the receives in the 0 to 25 microsecond range processed no kernel skbuffs and therefore had no kernel overhead; we hypothesize that this is because other receives that took longer than 25 microseconds copied data out of skbuffs into local kernel buffers. For these receives, our framework does measure substantial interrupt processing, scheduling, and data copy overheads, though the majority of this time is not surprisingly, spent waiting for data to arrive at the device. We are currently further studying this behavior to quantify the breakdown of kernel processing times for monitored packets and to determine how to account for inter-packet kernel buffering behaviors. We are also planning data collection runs on larger systems and more complex applications to further study UNIX system overheads using our newly developed framework.

6 Related Work

This section describes various tools currently available for the purpose of performance monitoring and various services these tools provide. A tool which is meant to be used for performance monitoring should be able to provide performance information which is related to individual OS requests as opposed to general system wide profiling information that most

of the currently available tools provide. Moreover the information generated will be more helpful if it is detailed enough to be of some use and available to the applications while they are running instead of after they are finished processing. A great deal of prior work has been done in the area of kernel performance monitoring, though little focused on per-request kernel monitoring [5, 7, 8, 16, 18]. The tools and techniques available for kernel performance monitoring can be broadly divided into three main categories depending upon the approach taken.

1. Trace Based Tools The most widely used are trace-based tools. These tools monitor kernel or user-space code by logging various events taking place into a log either on disk or in memory. They also include various tools that read these log files and provide the user with various different presentations of the trace. It is then the job of user to interpret any anomaly in system behavior by looking at the trace. Some examples of these tools are Linux Trace Toolkit, tracing infrastructure in K42 [10] and Magpie [1].

2. Statistical Sampling-Based Tools These tools depend upon the presence of hardware performance counters in the machine architecture the tools are running on. These tools periodically read the values in hardware counters instead of tracing events. As the number of hardware counters is fixed, these tools are not helpful when in-depth performance information is required. These tools also require exclusive control of these counters and hence cannot be used when other similar other tools are running on a system. Some examples of these tools are Digital Continuous Profiling Infrastructure (`dcpi`) [7], Oprofile [16] and Morph [21].

3. Other tools There are some tools such as Paradyn [13], Crosswalk [14], IPS-2[8], Ganglia [12] and Supermon [18] which does not fall into any of the above two categories. Some of these use a technique called dynamic instrumentation to efficiently obtain performance profiles of unmodified executables and provide performance data down to the procedure and statement level.

Figure 7 lists various tools currently available. The figure summarizes various services that we want to be present in a tool meant for providing performance information, and what among these services currently available monitoring tools provide. It is evident that none of the tools provide all the services that our framework seeks to provide.

	Services / Tools	Per-Request Information	Online Information	Detailed Information	Low Overhead
Trace Based Tools	Magpie			✓	
	Tracing in K42			✓	✓
	LTT			✓	✓
Sampling Based Tools	OProfile				✓
	Continuous Profiling Infrastructure				
	Morph			✓	
Others	Paradyn			✓	
	IPS			✓	
	Crosswalk	✓	✓		✓

Figure 7: Currently available performance monitoring tools.

7 Future Work

A variety of future work could be done to extend and utilize our framework. Such future work could include its use within other projects such as message-centric monitoring or anomaly detection. It could also include improving our framework for use on multiprocessor systems as well as with preemptable kernels. Finally, extensions to the framework are needed for supporting a more diverse collection of system calls. Potential future work will also include running the various benchmarks over large cluster of machines. This will require dedicated machines each running our modified Linux kernel and exclusive access to a big cluster is not yet available to us.

Usage of the framework within other projects Now that an initial implementation is complete, the resulting framework can be used to understand the local impacts of operating system performance on a variety of different HPC applications. In particular, it can be used to monitor individual system call performance on large-scale systems and longer-running applications, and to add instrumentation for further system calls and support for kernel-bypass communication mechanisms such as the Myrinet GM messaging layer [15]. For broader system monitoring support, integrating the work with recent work on message-centric monitoring [2] is also an interesting possibility. Finally, integrating our framework with automatic anomaly detection software might allow for automated detection of operating system-induced performance anomalies.

Improving LTT to enable usage of the framework on multiprocessors Multiprocessor systems result in several different issues that will have to be addressed in order to implement this framework on those systems, one of which is synchronization issue as described in section 2.2. Other issues could be migration of processes to other processors. A preemptable kernel could also result in more complexity to the framework because any event can be responsible for incomplete events in the tracing buffer. Since LTT uses a single buffer to log events from all the processors, it would then become very difficult to extract performance information from that single buffer as there could be several incomplete events in the circular buffer.

One way to address these issues would be to disable preemption in the kernel while LTT is writing events to a buffer. Disabling preemption will also solve the thread migration problem. Thread migration can result in execution of a single system call on multiple processors, due to which performance data related to a particular system call will be associated with more than one processor. It is also possible to adopt the approach taken by the K42 tracing infrastructure which assigns one individual buffer for every processor. This one buffer per processor approach will work fine if there is no thread migration in the kernel. We can then apply our synchronization solution on each individual buffer.

8 Conclusions

In this paper, we presented a new framework for fine-grained analysis of operating system overheads in HPC applications. This framework, built by modifying and extending tools such as the Linux Trace Toolkit, adds minimal overhead to the runtime of HPC applications and returns detailed information on OS overheads in system calls. We also presented preliminary numbers on the behavior of Linux networking system calls on simple HPC benchmarks and showed that, at least for these applications, a carefully constructed Linux cluster setup has relatively predictable system call performance. In this figure, this framework will serve as a foundation for work on optimizing operating systems and network protocols to better serve the needs of high-performance applications, as well as detecting anomalous operating system behavior in running systems.

References

[1] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, Lihei, Hawaii, 2003.

[2] Patrick G. Bridges and Arthur B. Maccabe. IMPuLSE: Integrated monitoring and profiling for large-scale environments. In *Proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Oct. 2004.

[3] Ron Brightwell, Arthur B. Maccabe, and Rolf Riesen. On the appropriateness of commodity operating systems for large-scale, balanced computing systems. In *2003 International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003.

[4] Ron Brightwell, Rolf Riesen, Keith Underwood, Patrick G. Bridges, Arthur B. Maccabe, and Trammel Hudson. A performance comparison of Linux and a lightweight kernel. In *IEEE International Conference on Cluster Computing*, December 2003.

[5] Michael Dagenais, Richard Moore, Robert W. Wisniewski, Karim Yaghmour, and Tom Zanussi. Efficient and accurate tracing of events in Linux clusters. In *Proceedings of the Conference on High Performance Computing Systems (HPCS)*, 2003.

[6] Rob F. Van der Wijngaart. Nas parallel benchmarks version 2.4. NAS Technical Report NAS-02-007, Computer Science Corporation, NASA Advanced Supercomputing(NAS) Division, NASA Ames Research Center, 2002.

[7] S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth Symposium on Operating System Principles*, 1997.

[8] J. K. Hollingsworth, R. B. Irvin, and B. P. Miller. The integration of application and system based metrics in a parallel program performance tool. In *Principles and Practice of Parallel Programming*, pages 189–200, 1991.

[9] Terry Jones, William Tuel, Larry Brenner, Jeffery Fier, Patrick Caffrey, Sean Dawson, Rob Neely, Robert Blackmore, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of SC'03*, 2003.

[10] The K42 Team. *K42 Performance Monitoring and Tracing*, 2001.

[11] D. J. Kerbyson, H. J. Alme, Adolfy Hoisie, Fabrizio Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–48, Denver, CO, 2001. ACM Press.

[12] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, May 2004.

[13] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffery K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[14] Alexander V. Mirgorodskiy and Barton P. Miller. Crosswalk: A tool for performance profiling across the user-kernel boundary. In *International Conference on Parallel Computing (ParCo)*, Dresden, Germany, September 2003.

[15] Myricom, Inc. The GM message passing system. Technical report, Myricom, Inc., 1997.

[16] The OProfile home page. http://oprofile.sourceforge.net.

[17] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of SC'03*, 2003.

[18] Matthew J. Sottile and Ronald G. Minnich. Supermon: A high-speed cluster monitoring system. In *IEEE Conference on Cluster Computing*, September 2002.

[19] Karim Yaghmour and Michael R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[20] Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel R. Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Proceeding of the Linux Symposium*, July 2003.

[21] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, 1997.