Early Experiences with KTAU on the IBM BG/L

Aroon Nataraj, Allen D. Malony, Alan Morris, and Sameer Shende

Performance Research Laboratory, Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{anataraj, malony, amorris, sameer}@cs.uoregon.edu

Abstract. The influences of OS and system-specific effects on application performance are increasingly important in high performance computing. In this regard, OS kernel measurement is necessary to understand the interrelationship of system and application behavior. This can be viewed from two perspectives: *kernel-wide* and *process-centric*. An integrated methodology and framework to observe both views in HPC systems using OS kernel measurement has remained elusive. We demonstrate a new tool called KTAU (Kernel TAU) that aims to provide parallel kernel performance measurement from both perspectives. KTAU extends the TAU performance system with kernel-level monitoring, while leveraging TAU's measurement and analysis capabilities. As part of the ZeptoOS scalable operating systems project, we report early experiences using KTAU in ZeptoOS on the IBM BG/L system.

Keywords: Kernel, performance, measurement, analysis.

1 Introduction

As High Performance Computing (HPC) moves towards ever larger parallel environments, the influence of OS and system-specific effects on application performance are increasingly important to include in a comprehensive performance view. These effects have already been demonstrated ([10], [16]) to be potential bottlenecks, but an integrated methodology and framework to observe their influence relative to application activities and performance has yet to be fully developed. Such an approach will require OS performance monitoring. OS performance can be observed from two different perspectives. One way is to view the entire kernel operation as a whole, aggregating performance data from all active processes in the system and including the activities of the OS when servicing system-calls made by applications as well as activities not directly related to applications (e.g. servicing interrupts or keeping time). We will refer to this as the *kernel-wide* perspective, which is helpful to understand OS behavior and to identify and remove kernel hot spots. Unfortunately, this view does not provide insight into what parts of an application spend time inside the OS and why.

Another way to view OS performance is within the context of an application's execution. Application performance is affected by the interaction of user-space

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 99–110, 2006.
© Springer-Verlag Berlin Heidelberg 2006

behavior with the OS, as well as what is going on in the rest of the system. By looking at how the OS behaves in the context of individual processes (instead of aggregate performance) we can provide a view that details interactions between programs, daemons, and system services. This *process-centric* perspective is helpful in tuning the OS for a specific workload, tuning the application to better conform to the OS configuration, and in exposing the source of performance problems (in the OS or the application).

Both these perspectives are important in OS performance measurement and analysis on HPC. The challenge is how to support both while providing a monitoring infrastructure that provides detailed visibility of kernel actions and that is easy to use by different tools. For example, the interactions between applications and the OS mainly occur through five different mechanisms: system-calls, exceptions, interrupts, scheduling, and signals. It is important to understand all forms of interactions as the application performance is influenced by each. However, some are easier to observe than others.

Our approach to the challenges above is the development of a new Linux Kernel performance measurement facility called KTAU, which extends the TAU [4] performance system with kernel-level monitoring. KTAU allows both a *kernel-wide* perspective of the OS performance as well as a *process-centric* perspective which merges kernel-space measurements with user-space performance data measured by TAU. KTAU is part of the ZeptoOS [5] research project to study operating systems for petascale systems and is included in the ZeptoOS distribution for the IBM BG/L I/O nodes. Below we describe the design of KTAU in Section 2, the BG/L in Section 3, KTAU's integration in the ZeptoOS and our early experiences on the IBM BG/L platform in Section 4, Related work is given in Section 5. Section 6 offers final remarks and future directions.

2 KTAU Design and Implementation

Kernel Tuning and Analysis Utilities (KTAU) is a toolkit for profiling and tracing the Linux Kernel. The toolkit is unique in its ability to produce both a kernel-specific and process-specific view of performance. Its main strength is in analyzing program behavior within the context of the kernel. KTAU generates performance data compatible with the TAU performance system [4], allowing TAU's analysis tools to be used.

As shown in Figure 1, KTAU consists of five distinct parts, namely,

- the Kernel Instrumentation
- the Kernel Infrastructure
- the KTAU proc filesystem
- the libKtau user-space library
- and clients of KTAU including the integrated TAU framework and daemons

2.1 Kernel Source Instrumentation

The kernel instrumentation is composed of easy to use C macros and functions. The instrumentation allows KTAU to intercept the kernel execution path and

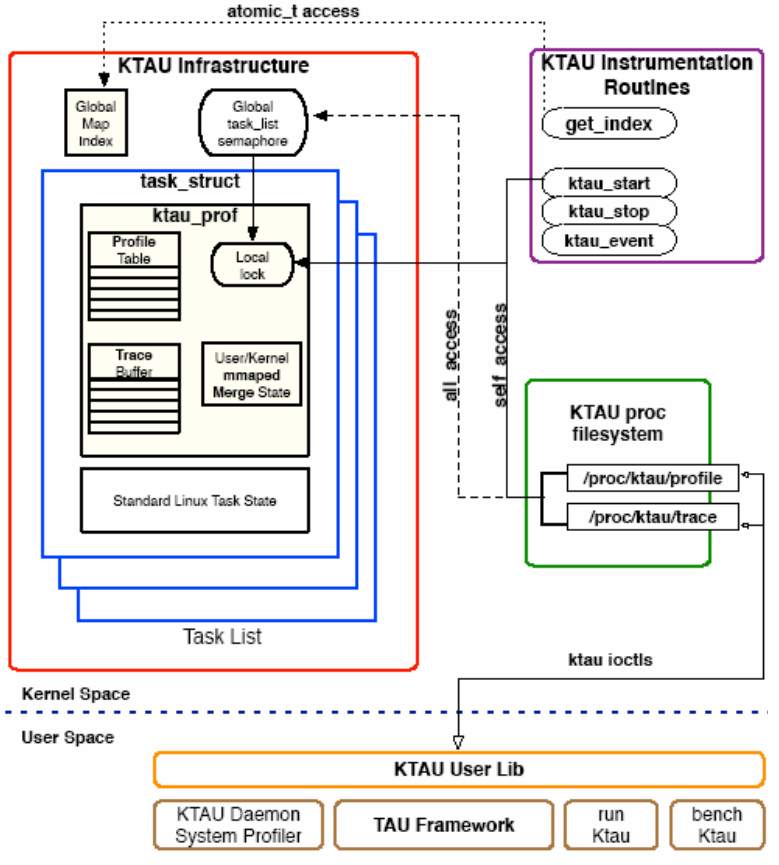

Fig. 1. KTAU Architecture

at each point measurement data is recorded. Profiling and tracing share the same instrumentation points. Instrumentation points are coarsely grouped based on various aspects of the kernel such as the subsystem in which they occur (e.g. scheduling, networking) or in what contexts they occur (e.g. system calls, interrupt, bottom-half handling). Compile-time configuration options and boot-time parameters control which groups of instrumentation points are turned on.

Three different types of macros are provided, namely *mapping*, *timer* and *event* macros. The *mapping* macro is necessary in every function that contains the other two types of instrumentations. It performs the function of providing identities to instrumentation points and mapping performance data to the instrumentation points. The next macro is a start-stop *timer* that calculates time elapsed between an entry and exit point. To obtain high-granularity timing resolution, low-level hardware timers are used. Lastly the *event* macro is used for events that do not conform to the *entry/exit* semantics or for events with non-monotonically increasing values.

2.2 Kernel KTAU Infrastructure

The infrastructure consists of the main in-kernel component that collects performance data and manages the lifecycle of profile/trace data structures. These structures are managed on a per-process basis and are initialized at the process creation time. KTAU adds a member of type *struct ktau_prof* $*$ to the process' *task_struct* which is the process control block in Linux. Small changes are also made to process creation and termination subroutines.

struct ktau_prof comprises of a profile table of configurable size, a fixed-size circular trace buffer and various other state variables for synchronizing access, inclusive/exclusive time calculation and merging of user and kernel profile data.

2.3 KTAU *proc* Filesystem

The *proc filesystem* component interfaces user-space clients with the kernel infrastructure component. It exposes two entries under */proc/ktau* called *profile* and *trace*. User-space clients perform IOCTLs on these files to access kernel performance data.

To accommodate frequent use clients, such as daemons that repeatedly retrieve data at small intervals, a memory-mapped buffer strategy (using *get_user_pages* and *vmap()* kernel routines in Linux) is being experimented with. This removes the need to repeatedly copy to/from userspace.

2.4 libKtau – User Library and API

The User API to KTAU provides a small set of easy to use functions that hide the details of the *proc* filesystem protocol, shielding applications from changes to KTAU kernel components. It provides control (merging, overhead calculation), data retrieval, data conversion (ASCII/binary) and formatted output.

All requests from user-space are grouped into three accessing schemes namely 'self', 'others' and 'all', referring to the set of processes of interest. For 'self' requests, kernel-mode performance data of the same process as the user-space client making the request is accessed. 'other' requests are for a set of processes (one or more) explicitly named in the request. And 'all', just an extension of 'other', is a convenience sparing clients from having to find the pids of all the processes on the system. The reason for having different modes has to do with reducing or removing the need for locking(a process accessing its own kernel-level profile does not need synchronization as it cannot race against itself).

2.5 KTAU Clients

KTAUD - KTAU Daemon. KTAUD periodically extracts performance data from the kernel. It can be configured to extract information for all or a subset of processes. Although it supports extracting profile data, its periodic nature suits it to dumping trace data, as trace buffers within the kernel can become full.

Integrated TAU Framework. The TAU measurement framework has been integrated with KTAU and is a client of libKtau. Applications instrumented with TAU automatically have access to kernel performance data of their own process (i.e. they will be self-profiling clients, using the 'self' mode of libKtau). When enabled through configuration, TAU will generate merged user/kernel profile information. This is supported only on KTAU-patched Linux OSes.

runKtau. Another type of client is similar to how the 'time' command under UNIX works. 'time' spawns a child process, executes the required job within that process, and then gathers rudimentary performance data by doing a waitpid on the child's pid. Similarly 'runktau' extracts the process' detailed KTAU profile.

3 The BG/L and Its Performance Observation

3.1 Brief Description of the Blue Gene/L

The Blue Gene L(BG/L) is a recent massively parallel supercomputer system from IBM, developed in collaboration with LLNL, that scales to 65,536 compute nodes [13]. Its architecture includes dual-processor Compute Nodes, I/O Nodes, front-end nodes and a Service node. Five different interconnect networks provide file I/O, control, debugging and interprocessor communication. Our focus is currently toward the compute and I/O nodes. We provide only a brief description of those aspects relevant to the current work as other work describes all aspects of the BG/L architecture and system software in detail([11],[6]).

The dual-processor (700 Mhz PPC 440) compute nodes, running a proprietary IBM operating system called the Compute Node Kernel (CNK) act as the main computation engines. The CNK is a small, light-weight OS, written in C++, without multitasking or virtual memory support allowing as many cycles as possible to application processing. File I/O is not directly implemented by the CNK and instead is call-forwarded to dedicated I/O nodes.

The I/O nodes serve two purposes. They participate in control of compute nodes including initialization, program launch and termination. They also perform all File I/O processing on behalf of the compute nodes. Multiple Compute nodes share a single I/O node. The ratio of I/O to compute nodes is configurable during system setup with values ranging from 1:8 to 1:128. The I/O node along with the compute nodes connected to it, forms a partition set (or pset).

The IO node OS is a modified Linux kernel (called the IO Node Kernel or INK). Modifications include patches to change interrupt/exception handling, add device drivers and floating point unit support. Due to the nodes being disk-less, a ramdisk containing shells and utilities is loaded into memory during bootup and forms the root filesystem. After bootup remote filesystems can be mounted onto the I/O Node based on the configuration of the ramdisk.

The Control and Input/Output daemon (CIOD) running on the I/O Node manages the control and file I/O of compute nodes in its pset. It listens for and accepts requests for processing of forwarded I/O system calls from the compute

node applications. The file-I/O is blocking on the application side. The CIOD reissues the system-calls through the VFS (Virtual File system) on the I/O Node which in turn is implemented by the file-system in use (e.g. NFS or PVFS2).

A tree network connects the Compute Nodes in a pset to their corresponding I/O node. The tree is used for collective operations as well as file I/O.

3.2 The Performance Measurement Problem

The BG/L system involves different nodes and multiple interacting software components within the nodes. Our approach is to place a unified measurement framework, based on TAU and KTAU, that can observe the applications, the system software and the operating systems and can correlate the performance data from the disparate sources.

The compute and I/O nodes tend to influence application performance the most. The I/O node being a shared resource within a pset, it is important to understand aspects of sharing such as fairness and utilization. Key questions regarding configuration of CIOD and the kernel, kernel version, choice of filesystem (NFS, PVFS2, Lustre etc) need to be answered with reference to performance data. The problem can be further split as follows:

1. Measurement of the Compute Node applications and the CNK and correlation of both performance results across compute nodes. This is (partially) tackled by using applications instrumented with TAU. TAU instrumentation allows tracing/profiling the applications and libraries such as MPI (using the MPI profiling interface). The CNK is closed and proprietary (akin to a black-box) and hence cannot be profiled or traced. But it is considered to be light-weight and built to stay out of the way of the application. We do not go into the details of using the TAU system on the compute nodes any further in this paper.

2. Measurement and correlation of performance of the CIOD, system daemons and IO Node Kernel. This is the focus of out current work including the integration with the ZeptoOS IO Node kernel on the Blue Gene/L at Argonne National Labs and our early experiences and results in using KTAU on the IO Nodes. The black-box nature of the CIOD is one of the challenges in measuring I/O node performance. The next section describes our approach.

3. Correlation of performance between the compute and I/O nodes within a pset and across multiple psets. This faces the problem of two black-boxes (CNK and CIOD) obscuring the call flow between the Compute and I/O-nodes, making correlation of events challenging. This is the target of future work.

4 KTAU On BG/L

Figure 2 shows a simplified view of the architecture of the BG/L's I/O and Compute Nodes within a pset and the main software components on those nodes. The ZeptoOS Linux kernel on the IO Node has been patched with KTAU support and KTAUD has been added to the IO Node utilities. On the compute node side, the applications can be instrumented with TAU to generate profiles or traces.

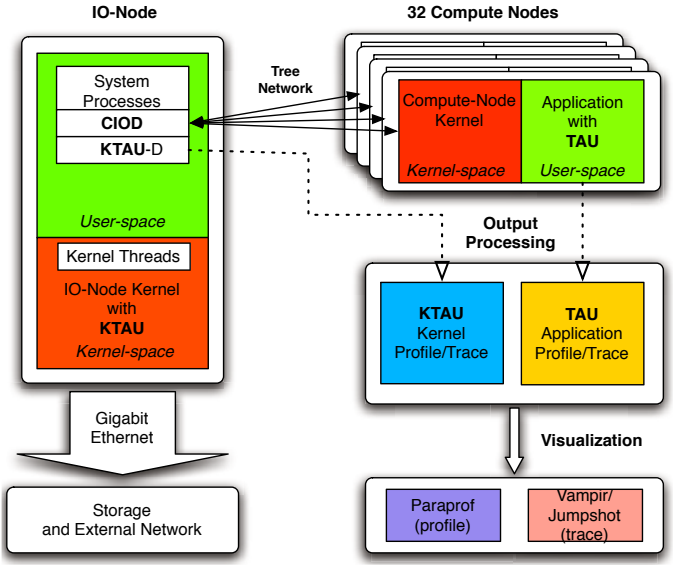

Fig. 2. KTAU on IBM BG/L

KTAU's process-centric kernel level profiling and tracing support is used to observe the CIOD despite it being closed-source. Its interactions at the kernel-level are the only way to peer into it. While utilities such as 'strace' are being used for this purpose, the perturbation caused by strace's trap-and-signal mechanism can cause significant differences between observed and 'real' behavior.

The integration with ZeptoOS (see [1] for details) includes patching and configuration changes to the Kernel source and the ramdisk image.

4.1 Performance Observation Capabilities Demonstrated

The following experiments aim to show KTAU's fine-grained performance observation capabilities on the BG/L IO Nodes. The experimental setup consists of a single pset (a single I/O node with 1 to 32 compute nodes). The compute nodes run a MPI I/O benchmark called *iotest* (used at ANL) which produces aggregate bandwidth numbers over varying block-sizes, number of processors and iterations. While the benchmark is run on the compute nodes, the CIOD services the read and write calls on the I/O node. On the I/O node while KTAU captures the kernel interactions, KTAUD periodically queries and saves the KTAU trace or profile information. Paraprof and Vampir [17] are used for visualization.

1. Fine Grained CIOD Tracing: We first show a trace fragment of the CIOD along with a zoomed in view in Figure 3. The groups of kernel functions shown in the upper image include TCP/IP, Socket calls, Interrupts, bottom half handling and scheduling. The lower zoomed-in image shows a typical CIOD kernel interaction when servicing a write system call from the compute node. The CIOD

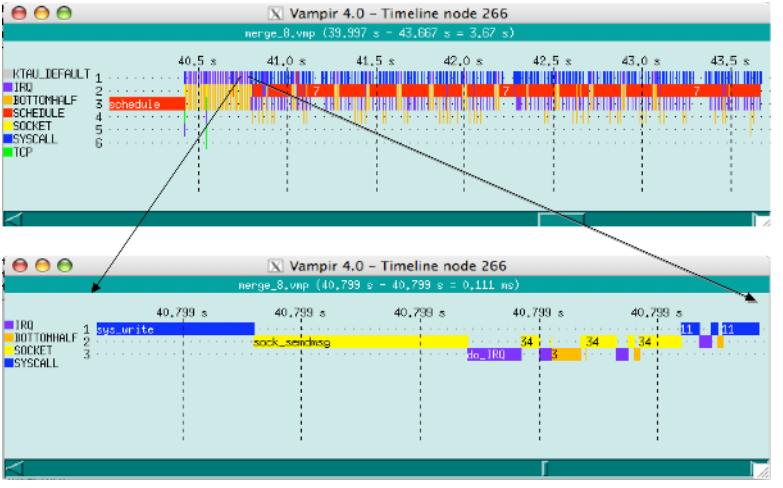

Fig. 3. KTAU Trace of CIOD – Bottom is Zoomed-In view of Top

calls sys_write on the IO node which in turn gets translated into socket and bottom half handling. UDP, instead of TCP, is used due to the filesystem being NFS (but the UDP activity is not shown as those instrumentation points were not enabled). The fine grained detail also shows interrupts occurring in between (do_IRQ). The 'node 266' in the title refers to process-id 266 (of the CIOD).

2. Effect of Increasing Compute Jobs on CIOD: As the I/O Node is a shared resource among the compute nodes in the pset, the load on the CIOD will no-doubt increase as the iotest benchmark is run in parallel over multiple compute nodes. Using KTAU we are able to capture the effect on the CIOD through its increasing interactions with the INK as it tries to service all of the compute nodes. Figure 4 shows five runs each with varying number of compute jobs.

3. Correlating behavior of Daemons and Kernel threads: It is possible to loosely correlate activity between different interacting daemons and kernel threads using traces collected from them. While no actual causal relationships can be directly deduced from the traces, intelligent guesses can be made based on timestamps and functionality of the processes. Figure 5 shows trace fragments from two processes on the IO node namely the CIOD and the RPCIOD (RPC I/O daemon). CIOD can be seen to repeatedly make sys_write calls and then be scheduled-out. At the same time, RPCIOD can be seen to be scheduled-in and perform sock_sendmsg followed by bottom-half handling. This behavior can be explained by the fact that with the underlying filesystem being NFS, the CIOD's sys_write calls are being handled by the RPCIOD.

4. Effect of Filesystem choice on CIOD: The backend filesystem mounted by the IO Node after bootup can be varied. By default on the ANL BG/L this is NFS, but a PVFS2 based configuration is also supported. The choice and configuration parameters of the backend filesystem can significantly influence I/O performance. To demonstrate KTAU's characterization of this effect two

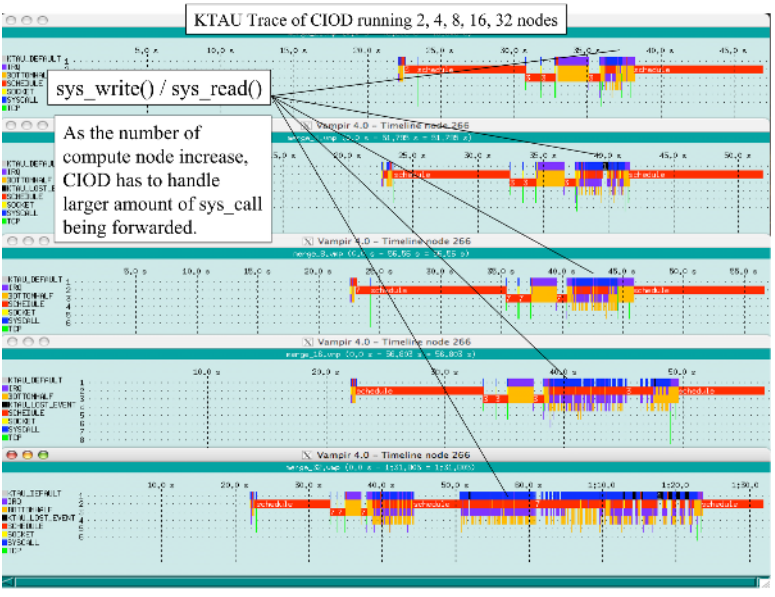

Fig. 4. KTAU Trace of CIOD - Effect of Increasing Compute Nodes from 2 to 32

runs of the benchmark were conducted, one under NFS and the under PVFS2. The traces[1] show clear differences in behavior (including obvious ones such as the use of TCP in the PVFS client versus the use of UDP in the RPCIOD).

5. Profiling support on the IO Node: Profiling support is important to provide as it can quickly help locate performance bottlenecks. While trace information can be post-processed to provide profiles, direct online profiling has much lower overheads. KTAU can collect profiling data, in addition to above mentioned traces, that can be visualized in Paraprof. The profiles[1] show inclusive and exclusive times taken by various kernel routines in the context of the CIOD.

5 Related Work

It is interesting to compare KTAU to other kernel instrumentation and measurement projects. We discuss below a few of the tools presented in Table 1.

KTAU is clearly distinguished from tools that use dynamic instrumentation rather than modify the kernel source. In **KernInst** [8] and **DTrace** [9] kernel routines are instrumented by *splicing* in measurement code dynamically at runtime. While kernel measurement can be modified during execution, the overhead of changing instrumentation can be greater than direct code instrumentation. KernInst, by itself, does not support merging user and kernel related performance information. Dtrace's user-level instrumentations also trap into the kernel, making it a costly choice for measurement of parallel HPC codes.

[1] Not shown here due to lack of space.

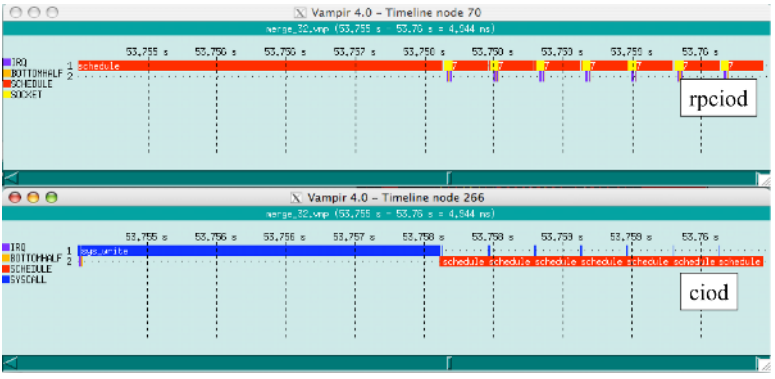

Fig. 5. Correlating CIOD Activity with RPC-IOD

In contrast to KernInst and DTrace, the **Linux Trace Toolkit (LTT)** [12] is based on source instrumentation. The actual source code of the Linux kernel is modified to include LTT macros in specified functions and LTT based data structures for holding the trace information. Other tools that fall into this category are K42 [14] and KLogger [18]. All of these only provide tracing.

5.1 Measured and Statistical Profiling Tools

SGI's KernProf [3] (under call-graph modes) is an example of measured-profiling tools. It uses compiler (gcc -pg option) generated profiling support. Every function is instrumented at compile-time with code to track a call-count and which functions called it and which functions were called from it. This is used to generate a call-graph of the kernel.

Oprofile [2], a statistical profiler, is meant to be a type of *continuous* profiler for Linux meaning always turned on. It performs both user-mode and kernel-mode profiling across the system providing merged user/kernel information. Its shortcomings include its inability to provide online information and the costly requirement of a daemon. SGI KernProf's flat-profile mode also uses sampling.

5.2 Merged User-Kernel Performance Analysis Tools

These tools explicitly provide support to merge the performance information between the application and the kernel. This enables understanding program-OS interaction and being able to pin-point bottleneck location in overall program/OS stack. It may also allow identifying intrusive effects such as excessive scheduling or interrupts that can steal cycles from applications.

CrossWalk [7] is a tool that *walks* a merged call-graph across the user-kernel boundary in search of the real source of performance bottlenecks. Using a specified performance threshold, it tries to find routines that take longer than the threshold starting its search from the *main()* function and examining the entire user/kernel control flow. **DeBox** [19] and [15] are also merged performance

measurement tools. All three suffer from the fundamental problem of not providing merged support, unlike KTAU, for interrupts, exceptions and scheduling.

5.3 Discussion

The tools mentioned above are unable to produce valuable merged information for all aspects of program-OS interaction. In addition, online OS performance information and ability to function without a daemon is not widely available. Most of the tools do not provide explicit support for collecting, analyzing and visualizing parallel performance data. KTAU aims to explicitly support online merged user/kernel performance analysis for all program-OS interactions in parallel HPC execution environments while using existing visualization tools.

Table 1. Related Work (N/E stands for 'No Explicit support')

Tool	Instr.	Measurement	User+Kernel	Parallel	SMP	OS
KernInst	Runtime	Flexible	N/E	N/E	Yes	Sun
DTrace	Runtime	Flexible	Trap into OS	N/E	Yes	Sun
KLogger	Static Src.	Trace	N/E	N/E	Yes	Linux
LTT	Static Src.	Trace	N/E	N/E	Yes	Linux
OProfile	Not App.	Stat. Prof.	Partial	N/E	Yes	Linux
KernProf(Flat)	Not App.	Stat. Prof.	N/E	N/E	Yes	Linux
KernProf(C Path)	gcc -pg	Call Path	N/E	N/E	Yes	Linux
LACSI'05	Static Src.	Trace	Syscall Only	N/E	No	Linux
CrossWalk	Runtime	Flexible	Syscall Only	N/E	Yes	Sun
DeBox	Static Src.	Meas. Prof., Trace	Syscall Only	N/E	Yes	Linux
KTAU+TAU	Static Src.	Meas. Prof., Trace	Full	Explicit	Yes	Linux

6 Conclusions and Future Work

The desire for a kernel monitoring infrastructure that can provide both a *kernel-wide* and *process-centric* performance perspective led us to the design and development of KTAU. KTAU is unique in its ability to measure the complete program-OS interaction, its support for joint daemon and program access to kernel performance data, and its integration with a robust application performance measurement and analysis system, TAU. In this paper, we described using KTAU as part of a performance measurement framework on Argonne's IBM BG/L system within the scope of the ZeptoOS project. Our early experiences indicate that KTAU along with TAU can be used to perform fine-grained performance measurement across the system. We demonstrated KTAU's measurement capabilities showing tracing/profiling of the I/O Node processes.

The I/O Node and the BG/L system as a whole provide many interesting performance questions that KTAU can be used to answer. We intend to deepen these early efforts through full-fledged experiments to study BG/L I/O performance and scaling under different backend filesystems, application loads and IO

Node (and CIOD) configurations. Another area we intend to explore is correlating performance observations between the Compute and I/O nodes. As the ZeptoOS project matures, KTAU will be also be used to provide kernel performance measurement and analysis for dynamically adaptive kernel configuration.

References

1. KTAU / ZeptoOS Integration. http://www.cs.uoregon.edu/research/ktau/docs.php.
2. Oprofile. http://sourceforge.net/projects/oprofile/.
3. Sgi kernprof. http://oss.sgi.com/projects/kernprof/.
4. TAU: Tuning and Analysis Utilities. http://www.cs.uoregon.edu/research/paracomp/tau/.
5. ZeptoOS: The Small Linux for Big Computers. http://www.mcs.anl.gov/zeptoos/.
6. A. Gara et. al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.
7. A. Mirgorodskiy et. al. Crosswalk: A tool for performance profiling across the user-kernel boundary.
8. A. Tamches et. al. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, 1999.
9. B. M. Cantrill et. al. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, USA.
10. F. Petrini et. al. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2003.
11. J. E. Moreira et. al. Blue Gene/L programming and operating environment. *IBM Journal of Research and Development*, 49(2/3):367–376, 2005.
12. K. Yaghmour et. al. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX '00: Proceedings of the 2000 USENIX Annual Technical Conference*, Boston, MA, USA, 2000.
13. N.R. Adiga et. al. An overview of the Blue Gene/L supercomputer. In *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.
14. R. W. Wisniewski et. al. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems.
15. S. Sharma et. al. A Framework for Analyzing Linux System Overheads on HPC Applications. In *LACSI '05: Proceedings of the 2005 Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, USA, 2005.
16. T. Jones et. al. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2003.
17. W. E. Nagel et. al. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
18. Y. Etsion et. al. Fine Grained Kernel Logging with KLogger: Experience and Insights.
19. Y. Ruan et. al. Making the "Box" Transparent: System Call Performance as a First-class Result. In *USENIX '04: Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, USA, 2004.