Kernel-Level Measurement for Integrated Parallel Performance Views: the KTAU Project

Aroon Nataraj Allen D. Malony Sameer Shende Alan Morris

Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
E-mail: {anataraj,malony,sameer,amorris}@cs.uoregon.edu

Abstract

The effect of the operating system on application performance is an increasingly important consideration in high performance computing. OS kernel measurement is key to understanding the performance influences and the interrelationship of system and user-level performance factors. The KTAU (Kernel TAU) methodology and Linux-based framework provides parallel kernel performance measurement from both a kernel-wide *and* process-centric *perspective. The first characterizes overall aggregate kernel performance for the entire system. The second characterizes kernel performance when it runs in the context of a particular process. KTAU extends the TAU performance system with kernel-level monitoring, while leveraging TAU's measurement and analysis capabilities. We explain the rational and motivations behind our approach, describe the KTAU design and implementation, and show working examples on multiple platforms demonstrating the versatility of KTAU in integrated system / application monitoring.*

1. Introduction

The performance of parallel applications on high-performance computing (HPC) systems is a consequence of the user-level execution of the application code and system-level (kernel) operations that occur while the application is running. As HPC systems evolve towards ever larger and more integrated parallel environments, the ability to observe all performance factors, their relative contributions and interrelationship, will become important to comprehensive performance understanding. Unfortunately, most parallel performance tools operate only at the user-level, leaving the kernel-level performance artifacts obscure. OS factors causing application performance bottlenecks, such as those demonstrated in [12, 21], are difficult to assess by user-level measurements alone. An integrated methodology and framework to observe OS actions relative to application activities and performance has yet to be fully developed. Minimally, such an approach will require OS kernel performance monitoring.

The OS influences application performance both directly and indirectly. Actions the OS takes independent of the application's execution have indirect effects on its performance. In contrast, we say that the OS directly influences application performance when its actions are a result of or in support of application execution. This motivates two different monitoring perspectives of OS performance in order to understand the effects. One perspective 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 (such as servicing hardware interrupts or keeping time). We will refer to this as the *kernel-wide* perspective, which is helpful in understanding OS behavior and in identifying and removing kernel hot spots. However, this view does not provide comprehensive insight into what parts of a particular application spend time inside the kernel and why.

Another way to view OS performance is within the context of a specific application's execution. Application performance is affected by the interaction of user-space 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 (versus aggregate performance) we can provide a detailed view of the 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 the *kernel-wide* and *process-centric* perspectives are important in OS performance measurement and analysis on HPC systems. The challenge is how to support both perspectives while providing a monitoring infrastructure that gives detailed visibility of kernel actions and is easy to use

1-4244-0328-6/06/$20.00 ©2006 IEEE.

by different tools. For example, the interactions between applications and the OS mainly occur through five different mechanisms: system-calls, exceptions, interrupts (hard and soft), scheduling, and signals. It is important to understand all forms of interactions as the application performance is influenced by each. Of these, system calls are the easiest to observe as they are synchronous with respect to the application. Also, importantly, system calls are serviced inside the kernel relative to the context of the calling process, allowing greater flexibility in how these actions are monitored. On the other hand, interrupts, scheduling, and exceptions occur asynchronously and usually in the interrupt context, where the kernel-level facilities and state constrain monitoring options and make access to user-space performance data difficult. One problem is observing all the different program-OS interactions and correlating performance across the user / OS boundary. Another is observing these interactions across a parallel environment and collectively gathering and analyzing the performance data. Support for this needs to be explicit, especially for post-processing, analysis, and visualization of parallel data.

Our approach is the development of a new Linux Kernel performance measurement facility called ***KTAU***, which extends the TAU [3] performance system with kernel-level monitoring. KTAU allows both a *kernel-wide* and *process-centric* perspective of OS performance to be obtained to show indirect and direct influences on application performance. KTAU provides a lightweight profiling facility and a tracing capability that can generate a detailed event log. We describe KTAU's design and architecture in Section 4. We demonstrate KTAU's features and versatility through working examples on different platforms in Section 5. We begin with a discussion of related work in Section 2 as background to the problem. The paper concludes with final remarks and future directions of KTAU in Section 6.

2. Background Related Work

To better understand the design and functionality of KTAU, we first review related research and technology in OS monitoring. Tools operating at the kernel level only and and those that attempt to combine user/kernel performance data are most closely related to KTAU's objectives. The different instrumentation and measurement techniques used are the primary distinguishing factors between the existing approaches. For lack of space, we show only a few candidate tools in each category. Table 1 compares related work on several dimensions.

Dynamic Instrumentation: Dynamic (runtime) instrumentation [13] modifies the executable code directly, thereby saving the cost of recompilation and relinking. **KernInst** [8] and **DTrace** [9] use dynamic instrumentation to insert measurement code into kernel routines. While KernInst incurs overhead only for dynamically instrumented events, trampolining adds delay and there is a cost in altering instrumentation during execution. KernInst, by itself, does not support merging user and kernel performance data. In Dtrace, application-level instrumentation also traps into the kernel where the kernel-logging facility logs both application and kernel performance data. However, because trapping into the kernel can be costly, using DTrace may not be a scalable option for combined user/kernel performance measurement of HPC codes.

Static Source Instrumentation: In contrast to the dynamic instrumentation in KernInst and DTrace, source instrumentation tools operate on static source code. **Linux Trace Toolkit (LTT)** [14], based on kernel source instrumentation, is a **Kernel Tracing** tool. While LTT can provide rich information about kernel function, its use of heavy-weight timing (*gettimeofday*) has high overhead and low precision, and its use of a single trace buffer requires global locks. Some of the limitations are being addressed [11]. Source-instrumented kernels also support tools to measure performance in the form of profile statistics. **SGI's KernProf** [2] (under callgraph modes) is an example of a **Kernel Profiling** tool.. By compiling the Linux Kernel with the *-pg* option to *gcc*, every function is instrumented with code to track call counts and parent/child relationships. As every function is being profiled the overhead at runtime can be significant. Neither tool provides extensive support for online merging of user and kernel data.

Statistical Sampling Tools: In contrast to instrumentation that modifies the kernel code (source or executable), statistical sampling tools periodically record Program Counter (PC) values to build histograms of how performance is distributed. This approach can observe both user-mode and kernel-mode operation across the system including libraries, applications, kernel-image and kernel-modules on a per processor basis. For example, **Oprofile** [1] is meant to be a *continuous* profiler for Linux, meaning it is always turned on. Its shortcomings include an inability to provide online information (as it performs a type of partial tracing) and the requirement of a daemon. Other issues stem from the inaccuracy of sampling based profiles.

Merged User-Kernel Performance Analysis: To better understand program-OS interaction, localize bottlenecks in the program/OS execution stack and identify intrusive interrupt/scheduling effects, it is important to associate kernel actions with specific application processes. **DeBox** [24] makes system-call performance a first-class result by providing kernel-mode details (time in kernel, resource contention, and time spent blocked) as in-band data on a per-syscall basis. An implementation-level shortcoming is its inability to maintain performance data across multiple system-calls (the performance state is re-initialized at the start of each syscall). Therefore DeBox requires access to source code of any libraries used by the application. **CrossWalk** [6] is a tool that *walks* a merged call-graph across the user-kernel boundary in search of the real source of performance bottlenecks. Both tools' requirement of knowing call-site locations allows them to bridge only the user-kernel gap across system-calls. DeBox and CrossWalk do not provide any means of collecting interrupt, exception, or scheduler data outside of system calls.

Discussion: Table 1 summarizes the tools discussed above and others we studied. While many of the tools mentioned can perform kernel-only performance analysis, they are unable to produce valuable merged information for all aspects of program-OS interaction. Merged information is absent, in particular, for indirect influences such as interrupts and scheduling that can have a significant performance impact on parallel workloads. In addition, online OS performance information and the ability to function without a daemon is not widely available. A daemon-based model causes extra perturbation and makes online merged user/kernel performance monitoring more expensive. KTAU resorts to a daemon only when proprietary/closed-source applications disallow source instrumentation. Most of the tools do not provide explicit support for collecting, analyzing, and visualizing parallel performance data. KTAU aims to provide explicit support for online merged user/kernel performance analysis for all program-OS interactions in parallel HPC execution environments.

3. Objectives

The background discussion above provides a rich research context for the KTAU project. In relation to this work, KTAU's distinguishing high-level objectives are:

- Support low-overhead OS performance measurement at multiple levels of function and detail.
- Provide a kernel-wide perspective of OS performance.
- Provide a process-centric perspective of application performance within the kernel.
- Merge user-level and kernel-level performance information across all program-OS interactions.
- Provide online information and the ability to function without a daemon where possible.
- Support both profiling and tracing for kernel-wide and process-centric views in parallel systems.
- Leverage existing parallel performance analysis tools.
- Deliver a robust OS measurement system that can be used in scalable production environments.

While none of the tools studied meet these objectives fully, their design and implementation highlight important concerns and provide useful contrasts to help understand KTAU's approach.

The measurements made in the kernel will be driven by the observation needs of the performance evaluation problems anticipated. Our goal is to provide an integrated kernel measurement system that allows observation of multiple functional components of the OS and choice of level of performance measurement granularity of these components. Like LTT, KTAU targets key parts of the Linux Kernel, including interrupt handlers, the scheduling subsystem, system calls, and the network subsystem. Structured OS observation is also seen in K42, Dtrace, and KernProf, whereas other tools provide limited kernel views (e.g., only system calls). KTAU provides two types of measurement of kernel performance: profiling and tracing. Most of the other tools provide only one type. The choice allows measurement detail to be matched to observation requirements.

Any measurement of the OS raises concerns of overhead and influence on application (and kernel) execution. The frequency and detail of measurement are direct factors, but how the OS is instrumented is an important determinant as well. The above contrasts of dynamic versus source (compiled) versus interrupt-driven instrumentation to identify variant strategies (plus technologies) which result in different overhead effects. While each approach has its devotees, for any one, it is more important to assess overhead impact as best as possible. Instrumentation in KTAU is compiled into the kernel. As a result, there are issues about how to control (turn on/off) measurements and questions of execution perturbation. We present experiment data later to address these concerns.

Tools such as LTT, KernProf, and Oprofile provide a kernel-wide view. DeBox, DTrace, and CrossWalk provide capabilities for generating a process-centric view, with possible restrictions on the scope of kernel observation. In contrast, KTAU desires to support both perspectives. KTAU enables process-centric observation through runtime association of kernel performance with process context to a degree that is not apparent in other tools.

The performance data produced by KTAU is intentionally compatible with that produced by the TAU performance system [3]. By integrating with TAU and its post-processing tools, KTAU inherits powerful profile and trace analysis and visualization tools, such as Paraprof [17] for profiling and Vampir [22] and Jumpshot [16] for trace visualization.

The KTAU project is part of the ZeptoOS [4] research project at Argonne National Laboratory funded by the DOE Office of Science "Extreme Performance Scalable Operating Systems" program. Thus, KTAU is intended to be used in the performance evaluation of large-scale parallel systems and in performance monitoring and analysis at runtime

Classification of Related Work						
Tool	Instrumentation	Measurement	**Combined User/Kernel**	**Parallel**	SMP	OS
KernInst [8]	dynamic	flexible	not explicit	not explicit	yes	Solaris
DTrace [9]	dynamic	flexible	trap into OS	not explicit	yes	Solaris
LTT [14]	source	trace	not explicit	not explicit	yes	Linux
K42 [18]	source	trace	partial	not explicit	yes	K42
KLogger [23]	source	trace	not explicit	not explicit	yes	Linux
OProfile [1]	N/A	flat profile	partial	not explicit	yes	Linux
KernProf [2]	gcc (callgraph)	flat/callgraph profile	not explicit	not explicit	yes	Linux
[20]	source	trace	syscall only	not explicit	no	Linux
CrossWalk [6]	dynamic	flexible	syscall only	not explicit	yes	Solaris
DeBox [24]	source	profile/trace	syscall only	not explicit	yes	Linux
KTAU+TAU	source	profile/trace	***full***	***explicit***	yes	Linux

Table 1. Kernel-Only and Combined User/Kernel Performance Analysis Tools

for kernel module adaption in ZeptoOS.

4. Architecture

The KTAU system was designed for performance measurement of OS kernels, namely the Linux Kernel. As depicted in Figure 1, the KTAU architecture is organized into five distinct components:

- Kernel instrumentation
- Kernel measurement system
- KTAU proc filesystem
- libKtau user-space library
- clients of KTAU including the integrated TAU framework and daemons

While KTAU is specialized for Linux, the architecture could generally apply to other Unix-style OSes. In the following sections we discuss the KTAU architecture components and their implementation in more detail.

4.1. Kernel Instrumentation

Our instrumentation approach in KTAU inserts measurement code in the Linux Kernel source. The kernel instrumentation uses C macros and functions that allow KTAU to intercept the kernel execution path and record measured performance data. The instrumentation points are used to provide profiling data, tracing data, or both through the standard Linux Kernel configuration mechanism (e.g., *make menuconfig*). Instrumentation points are grouped based on various aspects of the kernel's operation, such as in which subsystem they occur (e.g. scheduling, networking) or in what contexts they arise (e.g., system calls, interrupt,

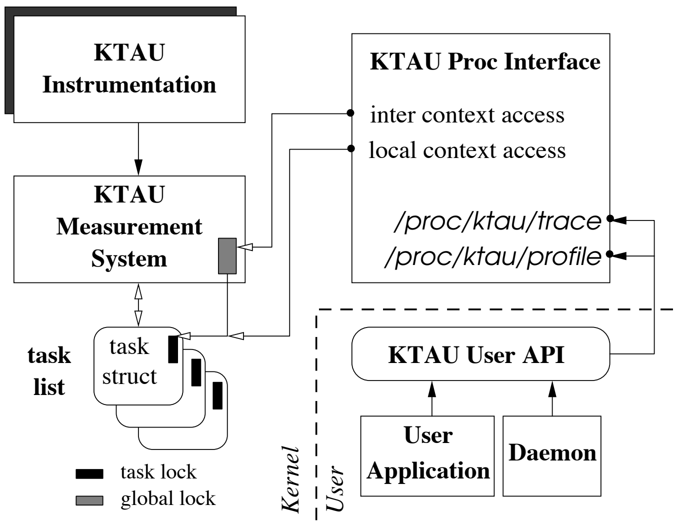

Figure 1. KTAU Architecture

bottom-half handling). To control the scope and degree of instrumentation, compile-time configuration options determine which groups of instrumentation points are enabled. Boot-time kernel options can also be used to enable or disable instrumentation groups.

Three types of instrumentation macros are provided in KTAU: *entry/exit event*, *atomic event*, and *event mapping*. The *entry/exit event* macro performs measurements between an entry and exit point. Currently, KTAU supports high-resolution timing based on low-level hardware timers (known as the *Time Stamp Counter* on the Intel architecture and the *Time Base* on the PowerPC). The KTAU entry/exit event instrumentation keeps track of the event activation (i.e., instrumentation) stack depth and uses it to calculate inclusive and exclusive performance data.

Not all instrumentation points of interest conform to *en-*

try/exit semantics or consist of monotonically increasing values. The *atomic event* macro allows KTAU to instrument events that occur stand-alone and to measure values specific to kernel operation, such as the sizes of network packets sent and received in the network subsystem. The *entry/exit* and *atomic* event macros are patterned on the instrumentation API used in TAU.

The performance data generated by the two instrumentation event macros must be stored with respect to the events defined. If we were only interested in supporting a kernel performance view, we could statically allocate structures for this purpose. However, one of KTAU's main strengths is in tracking process context within the kernel to produce process-centric performance data. The *event mapping* macro solves the problem of how to associate kernel events to some context, such as the current process context. The macro is used to create a unique identity for each instrumentation point, serving to map the measured performance data to dynamically allocated event performance structures. A global mapping index is incremented for the first invocation of every instrumented event. A static instrumentation ID variable created within the event code accepts the current global index value and binds that to the event. The instrumentation ID can then be used as an index into a table where event performance data is stored. The same mechanisms are used when mapping to specific process contexts.

4.2 KTAU Measurement System

The role of the KTAU measurement system is to collect kernel performance data and manage the life-cycle of per-process profile/trace data structures. The measurement system runs within the Linux Kernel and is engaged whenever a process is created or dies, kernel events occurs, or performance data is accessed. Upon process creation, KTAU add a measurement structure to the process's task structure in the Linux process control block. The size of this data structure is appropriately configured for profiling or tracing. When tracing is used, a fixed size circular trace buffer (of configurable length) is created for each process. Using this scheme, trace data may be lost if the buffer is not read fast enough by user-space applications or daemons. The KTAU measurement structure also contains state and synchronization variables.

4.3. KTAU *proc* filesystem

The *proc filesystem* scheme was chosen as a standard mechanism for interfacing user-space clients with the KTAU measurement system. As shown in Figure 1, KTAU exposes two entries under */proc/ktau* called *profile* and *trace*. User-space clients gain access to the entries via *libKtau* user library (see §4.4). We designed the interface to be session-less (where a session is described as persisting across multiple invocations from user-space). In this scheme a profile read operation, for instance, requires first a call to determine profile size and another call to retrieve the actual data into an allocated buffer. The interface does not maintain any saved state between calls (even though the size of profile data may change between calls). This design choice was made to avoid possible resource leaks due to misbehaving clients and the complexity required to handle such cases.

4.4. KTAU User API and libKtau Library

The KTAU User API provides access to a small set of easy-to-use functions that hide the details of the KTAU *proc* filesystem protocol. The libKtau library exports the API to shield applications using KTAU from changes to kernel-mode components. libKtau provides functions for kernel control (for merging, overhead calculation), kernel data retrieval, data conversion (ASCII to/from binary), and formatted stream output. Internally, libKtau performs IOCTLs on the *proc/ktau* files to access the kernel performance data.

4.5. KTAU Clients

Different types of clients can use KTAU. These include daemons that perform system-wide profiling/tracing, self-profiling clients (interested only in their own performance information), Unix-like command-line clients for timing programs, and internal KTAU timing/overhead query utilities. We briefly describe the KTAU clients currently implemented and used in our experiments.

KTAUD – the KTAU Daemon: KTAUD periodically extracts profile and trace data from the kernel. It can be configured to gather information for all processes or a subset of processes. Thus, KTAUD uses the 'other' and 'all' modes of libKtau. KTAUD is required primarily to monitor closed-source applications that cannot be instrumented.

TAU: The TAU performance system is used for application-level performance measurement and analysis. We have updated TAU to function as a client of KTAU, gathering kernel profile and trace data through the libKtau API. It is in this manner that we are able to merge application and kernel performance data.

runKtau: We created the *runKtau* client in a manner similar to the Unix *time* command. *time* spawns a child process, executes the required job within that process, and then gathers rudimentary performance data after the child process completes. *runktau* does the same, except it extracts the process's detailed KTAU profile.

5. KTAU Experiments

We conducted a series of experiments to demonstrate the features of KTAU on different Linux installations. The methodology followed for the first set of experiments sought to exercise the kernel in a well understood, controlled fashion and then use KTAU performance data to validate that the views produced are correct, revealing, and useful. The second set of experiments investigated KTAU's use on a large cluster environment under different instrumentation configurations to explore perturbation effects and highlight the benefit of kernel performance insight for interpreting application behavior. Unfortunately, not all of the performance studies we have done can be reported here, given the space provided. Thus, we present only a subset of those using the LU application from the NAS Parallel Benchmarks (NPB) (version 2.3) [10] and the ASCI Sweep3D [5]. In addition to other NPB applications, we have also experimented with the LMBENCH micro-benchmark for Linux [15]. Finally, we have conducted experiments with KTAU as integrated in ZeptoOS distribution and run on the IBM BG/L machine (see [7]).

5.1. Controlled Experiments

Simple benchmarks (LU, LMBENCH) were run on two platforms to show how the performance perspectives supported by KTAU appear in practice. Our testbeds included:

- *neutron*: 4-CPU Intel P3 Xeon 550 MHz, 1GB RAM, Linux 2.6.14.3 kernel with KTAU
- *neuronic*: 16-node 2-CPU Intel P4 Xeon 2.8GHz, 2GB RAM/node, Redhat Linux 2.4 kernel with KTAU

We use the LU benchmark to demonstrate several things about using KTAU and its benefits to integrated performance analysis. First, the experiments show KTAU's ability to gather kernel performance data in parallel environments. This data is used to detect and identify causes of performance inefficiencies in node operation. Second, analysis of kernel-level performance is able to reuse TAU tools for profile and trace presentation. Third, we see how KTAU provides merged user-kernel views, allowing application performance to be further explained by aspects of kernel behavior. Again, TAU tools can be applied to the merged performance information.

To test whether KTAU can detect performance artifacts, we introduced a performance anomaly in our experiments in the form of an artificially induced system workload. Periodically, while the LU application is running, an "overhead" process wakes up (after sleeping 10 seconds) and performs a CPU-intensive busy loop for a duration of 3 secs. This occurs only on one compute node, with the result of disrupting the LU computation. We expect to see the effects of this anomalous behavior as a performance artifact in the kernel and user profile and trace data.

Figure 2-A[1] presents a ParaProf display of kernel-level activity on 8 physical nodes of a 16-processor LU run (numbers represent host ids), aggregated across all processes running on each node. In the performance bargraph of node 8, we clearly see greater time in scheduling events introduce by the extra "overhead" process running on the node. To further understand what processes contributed to Host 8's kernel-wide performance view, the node profile in Figure 2-B shows kernel activity for each specific process (numbers represent process ids (*pids*)). It is clear that apart from the two LU processes (PID:28744 and PID:28746), there are many processes on the system. Also PID:28649 (the extra overhead process) is seen to be the most active and is the cause of the difference in the kernel-wide view. The views can help pin-point causes and locations of significant perturbation in the system.

Figure 2-D demonstrates the benefit of KTAU's integrated user/kernel profile to see performance influences not observed by user-level measurement alone. This view compares TAU's application-only profile with KTAU's, from the same LU run. A single node's profile is chosen. Each pair of bars represents a single routine's exclusive time in seconds, the upper (blue) bar represents the integrated view, and the lower (red) bar represents the standard TAU view. There are two key differences between the two. Kernel-level routines (such as schedule, system calls and interrupts) are additional in the KTAU view. Also the exclusive times of user-functions have been reduced to not include time spent in the kernel and represent the "true" exclusive time in the combined user/kernel call-stack. This comparison can identify compute-intensive routines with unusual kernel-times and also where in the kernel the time is spent.

Processes can be rescheduled voluntarily because they are waiting for some event (such as I/O completion or message arrival from another process). They are involuntarily rescheduled due to time-slice expiry. It is useful to differentiate the two cases in performance measurement. To do so we add, within *schedule()*, a new instrumentation point called *schedule_vol()* which is called when the *need_resched* process flag is unset (signaling time slice has not expired). We ran the NPB LU application on our 4-processor SMP host with this instrumentation. Due to weak CPU affinity, the four LU processes mostly stay on their respective processors. In addition we also run a daemon, pinned to CPU-0, that periodically performs a busy loop stealing cycles away from the LU application. Figure 2-C shows the result of voluntary versus involuntary scheduling on the four processes. LU-0 (top bar) is seen to be significantly affected by involuntary scheduling; its voluntary scheduling time is small. The reverse is true of the three other LU processes as

[1]The authors recommend that Figure 2 be viewed online.

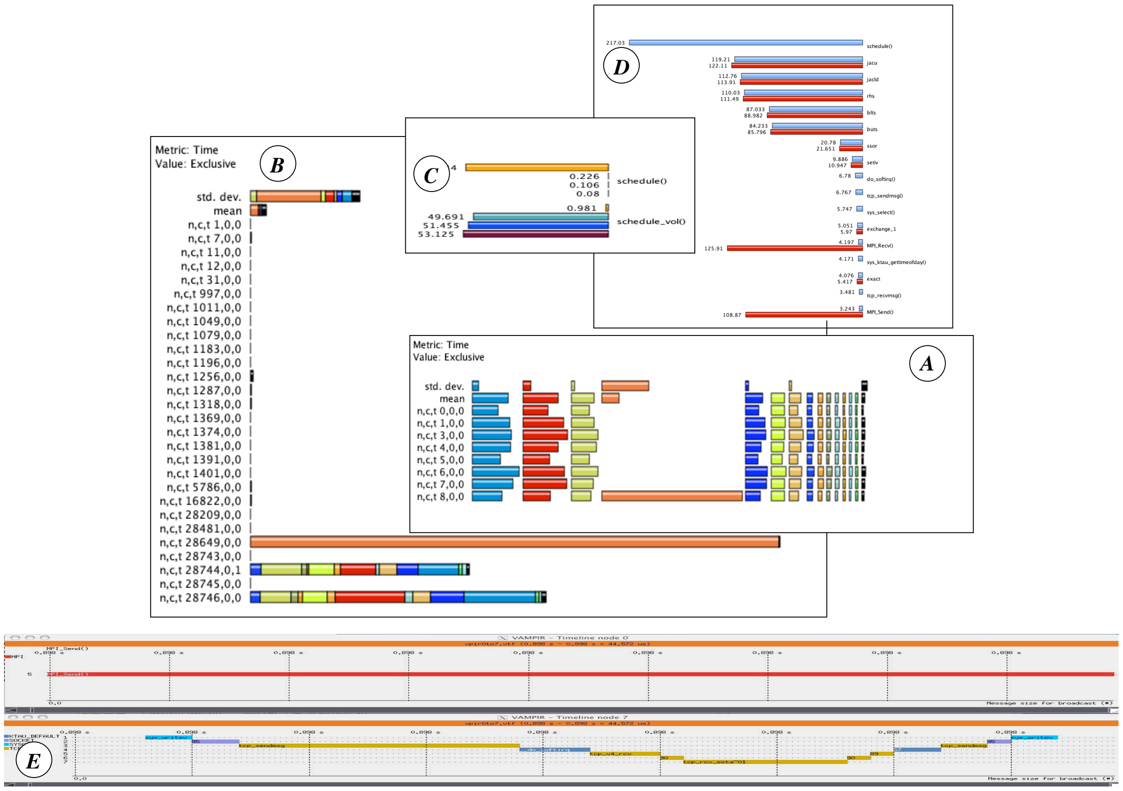

Figure 2. Controlled Experiments with LU Benchmark

they were waiting (voluntarily) for LU0 to catch-up. This example demonstrates how the source of dilation of and imbalance in system-loads can be better isolated.

Tracing can capture the spatial and temporal aspects of performance using a log of events. TAU application traces using the hardware timers can be correlated with KTAU kernel-level traces. Figure 2-E shows kernel-level activity within a user-space *MPI_Send()* call by the LU benchmark. Two snapshots from Vampir of TAU application and KTAU kernel traces have been merged. *MPI_Send()* is implemented by *sys_writev()*, *sock_sendmsg()* and *tcp_sendmsg()*. The bottom-half handling (*do_softirq()*) and tcp receive routines are not directly related to the send operation and instead occur when a bottom-half lock is released and pending softirqs are automatically checked.

5.2. Chiba Experiments

The second set of experiments we report target a larger-scale cluster environment. Argonne National Lab generously allocated a 128-node slice of the Chiba-City cluster for use with KTAU. Each node is a dual-processor Pentium III running at 450MHz with 512MB RAM. All nodes are connected by Ethernet. The OS is a KTAU-patched Linux 2.6.14.2 kernel. With this experimental cluster, we again used the LU benchmark, this time to assess the performance overhead of KTAU for a scalable application in different instrumentation modes. Serendipitously, we also needed to use KTAU to track down and resolve a performance issue related to the system configuration.

Initially, just to check out our experimental Chiba cluster, we configured KTAU with all instrumentation points enabled and ran the LU (Class C) application on 128 processes, first with 128 nodes (128x1) and then with 2 threads per node over 64 nodes (64x2). Our measurement involved merged user and OS level profiling using KTAU and TAU. The mean total execution time of LU over five runs for the 128x1 was 295.6 seconds and for the 64x2 configuration was 504.9 seconds. This was very surprising. We thought using 128 separate nodes may have advantages such as less contention for the single Ethernets interface, more bandwidth per thread, and less interference from daemons. But could these be sufficient to account for the 73.2% slowdown?

We used KTAU to help explain this performance bug and hopefully make optimizations to bring the 64x2 execution performance more in line with the 128x1 runs. On examining just the user-space profile produced by TAU we found several interesting features. The largest contributors to the profile were *MPI_Recv* and *rhs* routines. There was consid-

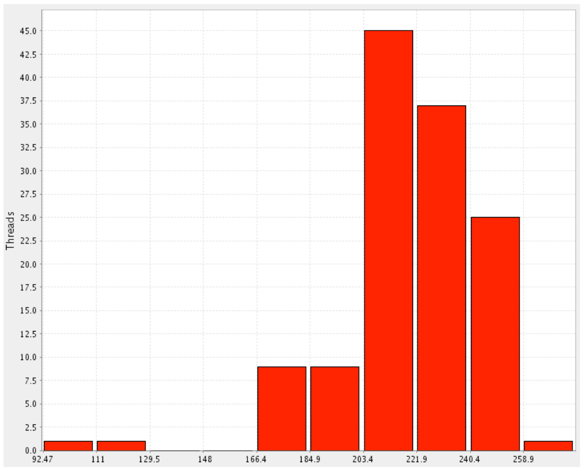

Figure 3. MPI_Recv: Excl. Time (seconds)

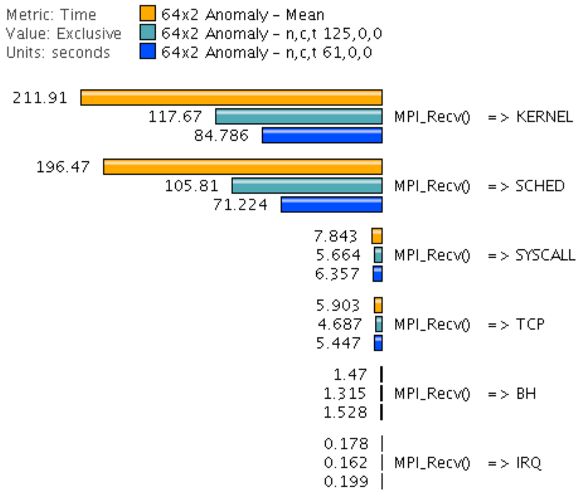

Figure 4. MPI_Recv OS Interactions

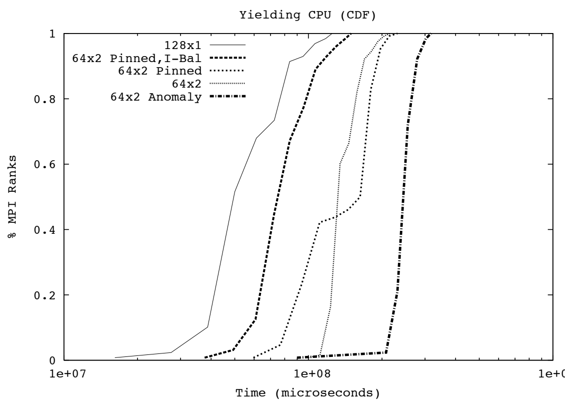

Figure 5. Voluntary Scheduling (NPB LU)

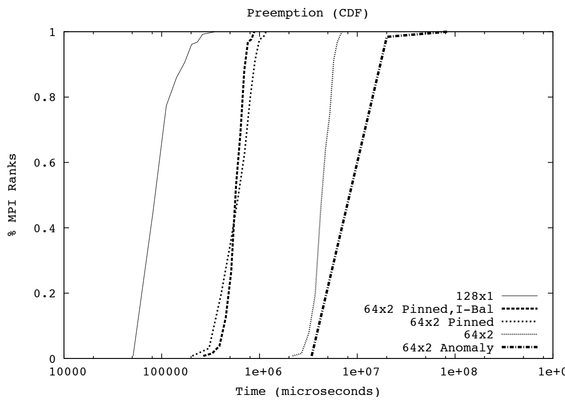

Figure 6. Involuntary Scheduling (NPB LU)

erable time spent in MPI_Recv across the ranks, except for two of the ranks which showed far lower MPI_Recv times. Figure 3 shows a histogram of the MPI_Recv routine's exclusive times. The left-most two outliers (ranks 61 and 125) also showed relatively larger times for the *rhs* routine. The user-level profile alone was unable to shed more light on the reasons behind these observations.

On examining the merged user/OS profile from KTAU additional observations were made. We first examined the MPI_Recv to see what types of OS interactions it had and how long they took. Figure 4 shows MPI_Recv's kernel call groups (i.e, those kernel routines active during MPI_Recv execution). The top bar is the mean across all the ranks and the next two bars are MPI ranks 125 and 61 respectively. On average most of MPI_Recv was spent inside scheduling, but comparatively lesser for ranks 125 and 61.

Scheduling is of two types: *voluntary*, where processes yield the processor waiting for completion of certain events, and *involuntary* where the process is pre-empted due to processor contention. Figures 5,6 were generated from the KTAU profile and show the cumulative distribution function (CDF) of voluntary and involuntary scheduling activity experienced by the LU application threads. The X-axis is logscale to accentuate the curve's shape. In Figure 5, the bottom of the curve labeled *64x2 Anomaly* shows a small proportion of threads experiencing relatively very low voluntary scheduling activity. This trend is reversed for *64x2 Anomaly* in Figure 6. Again two ranks (61, 125) differ significantly from others with large involuntary and small voluntary scheduling. These are the same two ranks that also showed higher *rsh* compute times in the user-level profile.

Surprisingly, we found that both of these ranks run on the same node (ccn10). Our first intuition was that OS-level interference and/or daemon activity on ccn10 was causing the problem. Low interrupt and bottom-half times suggested OS-interference was not the major cause. The involuntary scheduling instead suggested daemon activity causing preemption of the LU tasks on ccn10. Again, the views from

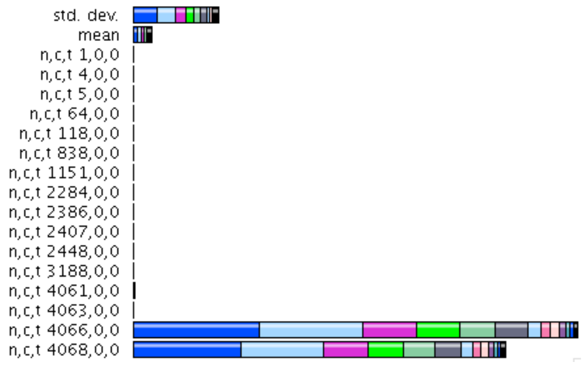

Figure 7. Node ccn10 OS Activity (NPB LU)

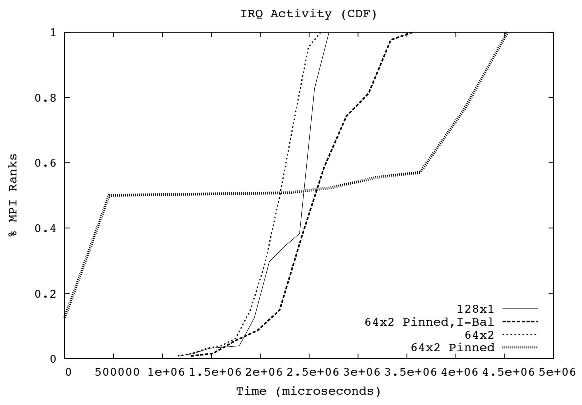

Figure 8. Interrupt Activity (NPB LU)

KTAU allowed easy testing of our hypothesis. Figure 7 shows the activity of all processes (including daemons, kernel threads, and applications) on ccn10. Each bar represents a single process that was active while the LU application was running on ccn10. The bottom two bars represent the LU tasks themselves. It is clear that no significant daemon activity occurred during the LU run as all other processes have minuscule execution times compared to the LU execution duration. This invalidated the daemon hypothesis.

The involuntary scheduling on ccn10 of the LU tasks could only be explained if the two LU tasks themselves pre-empted each other But why would this occur? On examination of the node itself (kernel logs and /proc/cpuinfo) we found that the OS had erroneously detected only a single processor (the reason is still being investigated). This matched exactly with KTAU measurements as the LU tasks were contending for ccn10's single available processor.

As a result, the other MPI ranks experienced high voluntary scheduling as they waited for the slow node (ccn10) in MPI_Recv. Remote influences will be seen as voluntary scheduling, where processes are made to wait for other processes. And local slow-down due to pre-emption is seen as involuntary scheduling. By removing the faulty node from the MPI ring and re-running our LU experiment, the benchmark reported an average total execution time of 396.7 seconds. This was an improvement of 21.4 percent, which is still 36.1 percent slower than the 128-node case.

We then examined the profiles of the runs without the faulty node. The curve *64x2* in Figures 5,6 shows lower scheduling activity than *64x2 Anomaly* as expected. But there still remains pre-emptive scheduling (Figure 6) of 2.5 to 7 seconds duration across the ranks. With only a few hundred milliseconds worth of daemon activity accounted for, the LU tasks were still pre-empting each other. We decided to pin the tasks (using cpu affinity) one per processor so as to avoid unnecessary pre-emption between the LU threads.

The pinned 64x2 run provided only a meager improvement of 3.2% (or 13.1 seconds) over the non-pinned runs. The curve labeled *64x2 Pinned* in Figure 6 shows much reduced pre-emptive scheduling (0.2 to 1.1 seconds). Surprisingly, KTAU reported a substantial increase in voluntary scheduling times. Figure 5 shows the increase in voluntary scheduling of *64x2 Pinned* over *64x2* and also a nearly bi-modal distribution. To understand why the *64x2 Pinned* run was experiencing greater imbalance and idle-waits, we examined another 'usual suspect' of system interference, namely interrupts. Figure 8 shows *64x2 Pinned* has a prominent bi-modal distribution with approximately half of the threads experiencing most of the interrupts. This was clearly the major cause of the imbalance leading to the high idle-wait times. It suggested that interrupt-balancing (or irq-balancing) was not enabled. Therefore all interrupts were being serviced by CPU0 which meant LU threads pinned to CPU0 were being delayed and threads pinned to CPU1 were waiting idly.

By re-running the experiment with both pinning and irq-balancing enabled a total execution time of 335.96 seconds. This resulted in a 16.5% improvement over the configuration without pinning and irq-balancing. Using KTAU's performance reporting capabilities the performance gap between the 64x2 and 128x1 configuration, runs had been reduced from 73.2% to 13.6%. This would have been impossible to do with user-level profile data alone. We also experimented with the Sweep3D ASCI benchmark [5] in this study. The gap there was reduced from 72.8% to 9.4%. These results are summarized in Table 2.

However, there was still a difference in performance between the 64x2 pinned, interrupt-balanced run and the 128x1 run (13.6% in NPB LU and 9.4% in Sweep3D). An advantage of correlating user-level and OS measurement is that the merged profiles can provide insight regarding the existence and causes of imbalance in the system. Examining kernel interactions of purely compute-bound segments of user-code is one way of investigating imbalance. In a synchronized parallel system with a perfectly load-balanced and synchronous workload, compute and communication

NPB LU and ASCI Sweep3d Experiments				
	NPB LU		ASCI Sweep3D	
Config	**Exec. Time**	**%Diff. from 128x1**	**Exec. Time**	**%Diff. from 128x1**
128x1	295.6	0	369.9	0
64x2 Anomaly	512.2	73.2%	639.3	72.8%
64x2	402.53	36.1%	428.96	15.9%
64x2 Pinned	389.4	31.7%	427.9	15.6%
64x2 Pin,I-Bal	335.96	13.6%	404.6	9.4%

Table 2. Exec. Time (secs) and % Slowdown from 128x1 Configuration

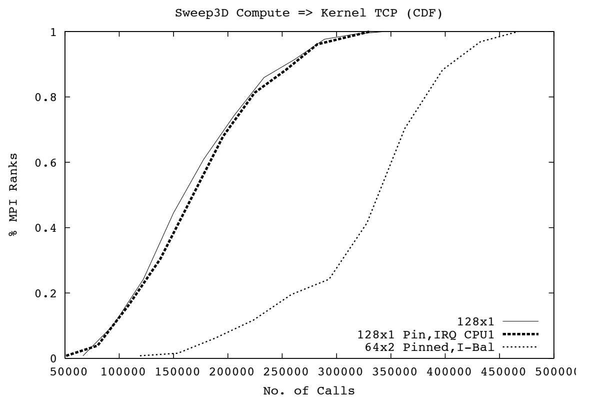

Figure 9. OS TCP in Compute (Sweep3D)

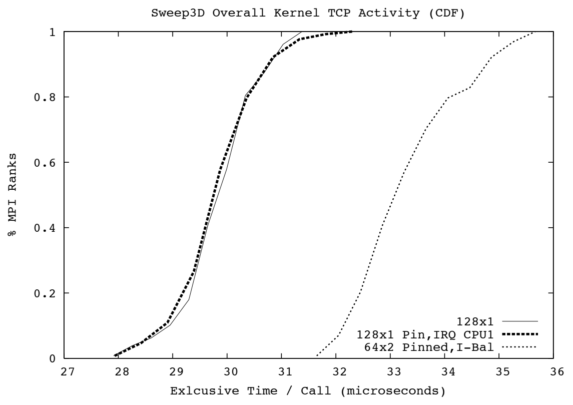

Figure 10. Time / Kernel TCP Call (Sweep3D)

phases happen in all processing elements (PE) at generally the same time. But assume some local slowdown causes a PE X's compute-phase to be longer than another PE Y's. The compute-bound segment of user-code at PE X will be interrupted by kernel-level network activity initiated by PE Y at the start of its I/O phase. This will further dilate X's compute phase causing even more imbalance effects.

Consider Sweep3D. Figure 9 shows the number of calls of kernel-level TCP routines that occurred within a compute bound phase (free of any communication) of Sweep3D inside the *sweep()* routine. Larger call numbers indicate a greater mixing of computation and communication work, suggesting greater imbalance due to increased background TCP activity. The *64x2 Pinned, I-Bal* curve shows a significantly larger amount of TCP calls than the *128x1* curve, implying that imbalance is reduced in the 128x1 configuration. Note, the total number of kernel-level TCP calls throughout the Sweep3D run did not differ significantly between the 64x2 and 128x1 cases. This suggests that the "missing" TCP routine calls in the 128x1 run (Figure 9) occur elsewhere in the I/O phase (where they should occur).

But what causes the imbalance to decrease in the 128x1 case? Is the extra processor absorbing some of the TCP activity? The curve labeled *128x1 Pin,IRQ CPU1* denotes a modified 128x1 configuration where both the Sweep3D process and all interrupts were pinned to the second cpu (CPU1). This curve closely follows '128x1'. Also, by default, irq-balancing was not enabled for the 128x1. Thus, the extra free processor was not causing the performance improvement.

Figure 10 shows the CDF of exclusive time of a single kernel-level TCP operation across all kernel TCP operations. There is a marked difference of approximately 11.5% over the entire range between the 64x2 and 128x1 configurations. Hence, TCP activity was more expensive when two processors are used for computation. Interrupt-balancing blindly distributes interrupts (and the bottom-halves that follow) to both processors. Data destined for a thread running on CPU0 may be received by the kernel on CPU1 causing cache related slowdowns. Therefore, the dilation in TCP processing times seen in 64x2 run is very likely cache related ([19] also found TCP/IP cache problems on SMP).

5.3. Perturbation Experiments

The above exercises clearly demonstrate KTAU's utility in performance measurement and analysis of parallel applications, especially when system-level effects cause performance problems. However, a source-based approach to kernel instrumentation, coupled with direct measurement, raises concerns of perturbation. The following experiments investigated the effects of KTAU measurement overhead on perturbation with respect to the level of instrumentation.

With a correctly configured Chiba cluster, we ran the LU benchmark again on 16 nodes and measured the overall application slowdown under five different configurations:

- *Base*: A vanilla Linux kernel and uninstrumented LU.
- *Ktau Off*: The kernel is patched with KTAU and all instrumentation points are compiled-in. Boot-time control turns off all OS instrumentation by setting flags that are checked at runtime.
- *ProfAll*: The same as KtauOff, but with all OS instrumentation points turned on.
- *ProfSched*: Only the instrumentation points of the scheduler subsystem are turned on.
- *ProfAll+Tau*: ProfAll, but with user-level Tau instrumentation points enabled in all LU routines.

Table 3 shows the percentage slowdown as calculated from minimum and average execution times (over five experiments) with respect to the baseline experiment. In some cases, the instrumented times ran faster (which was surprising), and we report this as a 0% slowdown.

Operation	Mean	Std.Dev	Min
Start	244.4	236.3	160
Stop	295.3	268.8	214

Table 4. Direct Overheads (cycles)

These results are remarkable for two reasons. First, with *Ktau Off* instrumentation (complete kernel instrumentation inserted, but disabled) we see no statistically significant slowdown for LU. While it is certainly the case that different workloads will result in different kernel behavior, these results suggest that KTAU instrumentation can be compiled into the kernel and disabled with no effects. Second, the slowdown for fully enabled KTAU instrumentation is small and increases only slightly with TAU instrumentation on average. If only performance data for kernel scheduling was of interest, KTAU's overhead is very small. It is interesting to note that the average execution time for the Sweep3D *Base* experiment was 368.25 seconds and 369.9 for the fully instrumented *ProfAll+Tau* experiment. This produces an average slowdown of 0.49%. For completeness, Table 4 reports the direct KTAU overhead introduced by a single measurement operation (start or stop).

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 demonstrated various aspects of KTAU's functionality and how KTAU is used for different kernel performance monitoring objectives. The present release of KTAU supports the Linux 2.4 and 2.6 kernels for 32-bit x86 and PowerPC platforms. We are currently porting KTAU to 64-bit IA-64 and PPC architectures and will be installing KTAU on a 16-processor SGI Prism Itanium-2 Linux system and a 16-processor IBM p690 Linux system in our lab.

There are several important KTAU objectives to pursue in the future. Because KTAU relies on direct instrumentation, we will always be concerned with the degree of measurement overhead and KTAU efficiency. Additional perturbation experiments will be conducted to assess KTAU's impact and to improve KTAU's operation. However, based on our current perturbation results, we believe a viable option for kernel monitoring is to instrument the kernel source directly, leave the instrumentation compiled in, and to implement dynamic measurement control to enable/disable kernel-level events at runtime. This is in contrast to other kernel monitoring tools that use dynamic instrumentation, such as KernInst. Presently, the instrumentation in KTAU is static. Our next step will be to develop mechanisms to dynamically disable/enable instrumentation points without requiring rebooting or recompilation. Finally, we will continue to improve the performance data sources that KTAU can access and improve integration with TAU's user-space capabilities to provide better correlation between user and kernel performance information. This work includes performance counter access to KTAU, better support for merged user-kernel call-graph profiles, phase-based profiling, and merged user-kernel traces. We will also continue to port KTAU to other HPC architectures.

The scope of KTAU application is quite broad. We have ported KTAU to the IBM BG/L machine [7] as part of our work on the ZeptoOS project [4, 7] (KTAU is a part of the ZeptoOS distribution). We will be evaluating I/O node performance of the BG/L system, and once a ZeptoOS port is complete, KTAU will be also be used to provide kernel performance measurement and analysis for dynamically adaptive kernel configuration. I/O performance characterization and kernel configuration evaluation are equally of interest on any cluster platform running Linux.

NPB LU Class C (16 Nodes)						ASCI Sweep3D (128 Nodes)	
Metric	**Base**	**Ktau Off**	**ProfAll**	**ProfSched**	**ProfAll+Tau**	**Base**	**ProfAll+Tau**
Min	468.36	463.6	477.13	461.66	475.8	-	-
% Min Slow	0	0	1.87	0	1.58	-	-
Avg	470.812	470.86	481.748	471.164	484.12	368.25	369.9
% Avg Slow	0	0.01	2.32	0.07	2.82	0	0.49

Table 3. Perturbation: Total Exec. Time (secs)

7. Acknowledgment

The KTAU research was supported by the Department of Energy's Office of Science (contract no. DE-FG02-05ER25663) and the National Science Foundation (grant no. NSF CCF 0444475). The authors thank Suravee Suthikulpanit for his contribution to KTAU's implementation and Rick Bradshaw for his assistance with the Chiba-City cluster.

References

[1] OProfile. http://sourceforge.net/projects/oprofile/.
[2] SGI KernProf. http://oss.sgi.com/projects/kernprof/.
[3] TAU: Tuning and Analysis Utilities. http://www.cs.uoregon.edu/research/paracomp/tau/.
[4] ZeptoOS: The small linux for big computers. http://www.mcs.anl.gov/zeptoos/.
[5] A. Hoisie et. al. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *ICPP'00: Proceedings of the 29th International Conference on Parallel Processing*, Toronto, Canada, 2000.
[6] A. Mirgorodskiy et. al. CrossWalk: A Tool for Performance Profiling Across the User-Kernel Boundary. In *ParCo'03: Proceedings of the International Conference on Parallel Computing*, Dresden, Germany, 2003.
[7] A. Nataraj et. al. Early Experiences with KTAU on the IBM BG/L. In *Europar'06: Proceedings of the European Conference on Parallel Processing*, Dresden, Germany, 2006.
[8] A.Tamches et. al. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *OSDI'99: Proceedings of Operating Systems Design and Implementation*, New Orleans, LA, USA, 1999.
[9] B.M. Cantrill et. al. Dynamic Instrumentation of Production Systems. In *USENIX '04: Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, USA, 2004.
[10] D. H. Bailey et. al. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
[11] M. D. R. et al. Efficient and Accurate Tracing of Events in Linux Clusters. In *HPCS'03: Proceedings of Symposium on High Performance Computing Systems and Applications*, Quebec, Canada, 2003.
[12] 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.
[13] J. K. Hollingsworth et. al. Dynamic Program Instrumentation for Scalable Performance Tools. Technical Report CS-TR-1994-1207, 1994.
[14] K. Yaghmour et. al. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *USENIX '00: Proceedings of the USENIX Annual Technical Conference*, Boston, MA, USA, 2000.
[15] L. W. McVoy et. al. lmbench: Portable Tools for Performance Analysis. In *USENIX'96: Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, USA, 1996.
[16] O. Zaki et. al. Toward Scalable Performance Visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.
[17] R. Bell et. al. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. *Lecture Notes in Computer Science*, 2790:17–26, 2003.
[18] R. W. Wisniewski et. al. Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems, 2003.
[19] S. Bhattacharya et. al. A Measurement Study of the Linux TCP/IP Stack Performance and Scalability on SMP systems. In *Proceedings of 1st International Conference on COMmunication Systems softWAre and middlewaRE (COMSWARE)*, Bangalore, India, 2006.
[20] 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.
[21] 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.
[22] W. E. Nagel et. al. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
[23] Y. Etsion et. al. Fine Grained Kernel Logging with KLogger: Experience and Insights. Technical Report Technical Report 2005-35. School of Computer Science and Engineering. The Hebrew University of Jerusalem, 2005.
[24] 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.