The Ghost in the Machine: Observing the Effects of Kernel Operation on Parallel Application Performance

Aroon Nataraj
Department of Computer &
Information Science
University of Oregon
Eugene, OR
anataraj@cs.uoregon.edu

Alan Morris
Department of Computer &
Information Science
University of Oregon
Eugene, OR
amorris@cs.uoregon.edu

Allen D. Malony
Department of Computer &
Information Science
University of Oregon
Eugene, OR
malony@cs.uoregon.edu

Matthew Sottile
CCS Division
Los Alamos National Lab
matt@lanl.gov

Pete Beckman
MCS Division Argonne
National Lab
beckman@mcs.anl.gov

ABSTRACT

The performance of a parallel application on a scalable HPC system is determined by user-level execution of the application code and system-level (OS kernel) operations. To understand the influences of system-level factors on application performance, the measurement of OS kernel activities is key. We describe a technology to observe kernel actions and make this information available to application-level performance measurement tools. The benefits of merged application and OS performance information and its use in parallel performance analysis are demonstrated, both for profiling and tracing methodologies. In particular, we focus on the problem of kernel noise assessment as a stress test of the approach. We show new results for characterizing noise and introduce new techniques for evaluating noise interference and its effects on application execution. Our kernel measurement and noise analysis technologies are being developed as part of Linux OS environments for scalable parallel systems.

1. INTRODUCTION

The performance of a parallel application is the consequence of user-level execution of the application code, OS-level operations occurring during program execution, and the interactions between the two. To understand fully the performance achieved on a HPC machine, it would be ideal to have an integrated set of user-level and OS-level measurements, plus additional information to identify the interactions and quantify their performance effects. Unfortunately, current parallel performance tools operate primarily at the level of the application code, producing a perspective on performance that, at best, lacks information on OS contributions, and, at worst, misallocates, misattributes, and misinterprets performance to application-only factors. It is untenable to argue that parallel application performance is dualistic in nature, and if only the OS were not present, the performance tool would produce correct results. To avoid being haunted by the performance anomalies and inconsistencies caused by *ghosts in the machine*, next-generation HPC performance engineering environments should include new techniques for understanding OS effects on application performance. Indeed, the ability to dynamically adapt the machine to the application demands such advances.

Research in the past five years has drawn attention to the role of the operating system in achieving scalable performance on massively parallel machines [5, 17]. An important focus has been on OS interference (a.k.a. noise) and its effects on application progress and synchronicity. The outcome of this work has been a better characterization of the interference phenomena and the relationship to performance behavior. In general, understanding how operating systems interact with applications and how interrupts, process scheduling, and I/O processing affect performance at increasing scales is key to petascale systems research [14]. This is true not only for configuring the OS and developing parallel applications to be more interference resilient, but also for adapting the kernel online for dynamic performance optimization.

Given a parallel application and parallel system, how is an integrated performance view obtained, such that OS-application interactions can be observed and analyzed? Past OS noise research has been based almost exclusively on modeling [17], specialized noise benchmarks [20], and message-driven simulation with noise-injection [5]. These methods do not translate to a general-purpose tool framework for investigating integrated performance in real parallel environments. What is required is an ability to measure OS activity directly and to associate OS measurements with application-level performance characteristics. Our approach is to instrument the OS kernel and measure system operations during application execution. The association of kernel and application measurements is supported by providing the parallel application with low-overhead access to the kernel performance data. The approach extends the KTAU [16] kernel-level measurement system with a capability to create metrics from kernel performance data that can be read by

SC07 November 10-16, 2007, Reno, Nevada, USA
Copyright 2007 ACM 978-1-59593-764-3/07/0011 ...$5.00.

the TAU application-level performance tool.

Section §2 describes the KTAU extensions for application-level access to Linux kernel performance data. These allow the TAU performance system to profile and trace kernel contributors to application performance. The section also reports the overheads for kernel data access using KTAU. In many respects, noise evaluation is a stress case for tools supporting integrated performance views. Section §3 describes analysis methods developed for the noise study presented in the paper. The work builds on our research in evaluation of operation system interference as well as measurement overhead analysis and compensation. Experimental results are discussed in Section §4 and show clearly the power of the methodology and the integrated tools. A review of related research is given in Section §5. Section §6 summarizes the research conclusions and outlines plans for future work.

2. APPLICATION ACCESS TO KERNEL METRICS

The KTAU [15] instrumentation and measurement fabric was developed for parallel Linux systems to observe the performance effects of OS and runtime (OS/R) components during application execution. KTAU instruments the Linux kernel to intercept the kernel control path and make measurements of certain components including interrupt handlers, the scheduling subsystem, system calls, and the network subsystem. Measurements are made *with respect to the running process*, and are captured in both profile- and trace-based forms. KTAU places the performance data for each process in kernel space and makes it accessible through the Linux *proc* file system mechanisms. KTAU also provides a API that can be used by an application process to obtain its own kernel performance information directly. An application instrumented with TAU in user-space automatically gets access to its kernel performance state due to the integration between TAU and KTAU. This also obviates the need for a daemon-based interface to kernel performance state as every parallel process/thread can directly access its own kernel state and that of all other processes if required. Unfortunately not all applications are open-source and instrumentable. For such cases, a daemon ($KTAUD$) is implemented to periodically access the kernel performance data of all or a subset of processes running on the host system, such as kernel threads, daemons, and user-level applications. Our noise analysis work in this paper did not use the daemon as all our parallel workloads were instrumentable and also since requiring a daemon can itself introduce noise. Both KTAUD and the API use the KTAU library (*libktau*) to access the kernel data in */proc/ktau/profile* and */proc/ktau/trace* for profiles and traces, respectively.

In [16], Nataraj et al. demonstrated the robustness of KTAU to elucidate kernel performance, with respect to both a process-centric and system-wide view. However, missing in KTAU was a capability to directly observe kernel performance at the time of application-level events. Such a capability is important for performance analyses that require kernel operations to be correlated with respect to application context (e.g., in the case of noise analysis). Although the KTAU API could be used to sample kernel performance information for each measured application event, the overhead of the *libktau proc* interface (requiring a Linux system call) limits measurement fidelity. Nataraj et al. enhanced KTAU in [15] to provide application context attribution when a kernel measurement was made (by looking back into TAU's runtime event callstack). However, this does not allow for application-triggered observation. To give a parallel application both efficient access to kernel performance state and runtime observation control, a more tightly-coupled mechanism is required.

2.1 KTAU Extensions

Figure 2.1 contrasts the new KTAU capability for kernel performance data access with the *libktau proc* support. The key idea is to make available a portion of application process memory (called a metric *container*) for use by KTAU to write certain kernel performance metrics. The application can then read the kernel metrics directly from memory, at the time an application event is triggered. The access overhead is reduced because only a subset of the kernel performance data gets promoted as metrics, avoiding the *proc* system call. The metrics themselves are derived from kernel performance data according to functions specified by the application. Metric computation occurs within KTAU and updates are made to the container at the time the kernel returns to the application process.

2.2 Overheads

Our objective is to make KTAU kernel metrics available to application-level performance measurement systems, such as the TAU performance system [2]. This requires low-overhead setup and access, preferably on the order of other sources of performance data. We measured the overhead of the new KTAU access mechanism relative the `gettimeofday()` (gtod) system call, PAPI counters [1], and a "no-op" *proc* call (no data is transferred) implemented in *libktau*. Table 1 shows the configuration and per-call overhead for different numbers of metrics. The tests were performed on a 2.6 GHz x86_64 Xeon (Clovertown) system running a KTAU-enabled Linux. Configuration overhead includes container setup and metric counter installation, which increases with the number of metrics. Typically, configuration would occur only once at the beginning of the application. Metric access overhead is the cost of reading the metrics. The "no-op" *proc* call and gtod are measured for each KTAU test case. All values are in cycles.

As observed, using KTAU metrics has a significantly lower configuration cost compared to the use of PAPI. Overheads to access metrics are on par to both PAPI counter access and gtod. They are superior to using the earlier *proc* interface to access kernel information, even when no data is transferred.

An alternative method to evaluate overhead is use a real application benchmark and compare the cost of TAU performance measurement without and with KTAU metrics. Again, providing KTAU metric access at the application level allows a performance system such as TAU to measure kernel performance with application events. TAU supports this in its ability to read multiple counters with each measured event. For this overhead test, we compared the total execution time of the LU application from the NAS Parallel Benchmarks (NPB) (version 2.3) [4] running on a 16-processor Linux cluster under different measurement setups: *base* (a vanilla Linux kernel and uninstrumented LU), *ktau-only* (KTAU kernel with four instrumentation points), *ktau-proc* (*ktau-only* with *proc* access at end), *ktau-tau* (*ktau-proc* with TAU instrumentation (profile and trace) of MPI

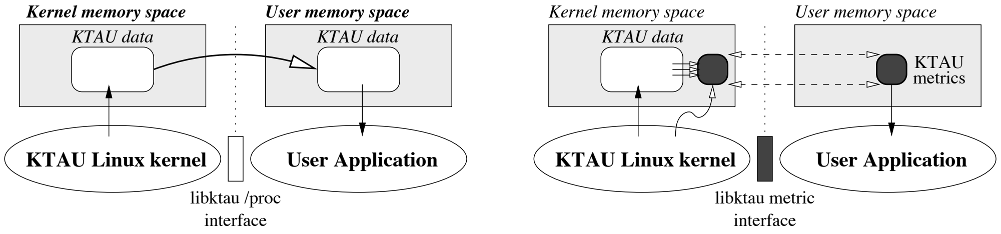


Figure 1: KTAU support for kernel performance data access. The left figure depicts the KTAU *proc* interface to KTAU data. Here a Linux system call transfers the data from kernel to user space. The right figure shows the new support for promoting KTAU-computed kernel metrics to user-space by mapping application memory to KTAU kernel memory.

# Metrics	KTAU config	Metric access	gtod	PAPI config	PAPI access	"no-op" proc
1	18520+17408	192	238	1830064	248	1150
2	18504+19112	272	-	1836720	304	1148
4	18864+25112	288	-	—	—	1215
8	18856+33776	400	-	—	—	1148

Table 1: KTAU metric access overhead relative to `gettimeofday()` (gtod), PAPI, and a "no-op" *proc* call. All values are in cycles.

events), and *ktau-tau-metrics* (*ktau-tau* with metric counters access). Table 2 shows the minimum execution time (over five experiments) with the percentage slowdown calculated with respect to the baseline experiment.

The LU overhead results are remarkable in several respects. First, a KTAU-instrumented Linux kernel, with four kernel actions being measured, showed indistinguishable LU performance compared to a vanilla Linux kernel. Second, with kernel performance data being measured, access to KTAU metrics introduces minuscule overhead compared to standard TAU performance measurements. Last, enabling profiling and tracing of MPI events introduces less than 1% execution time delay.

The ability to access kernel performance data as KTAU metrics at the application level allows for finer correlation of application performance with kernel operation. It places kernel actions in the context of application logic and state. This is important for the analysis of event-relative kernel behavior, such as occurs in noise cases.

3. NOISE MEASUREMENT AND ANALYSIS

Time spent in the operating system during an application's execution influences its performance. When the OS is working on behalf of the application, it is reasonable to regard the OS work as a component of application execution. Characterizing OS activities could certainly aid performance understanding. When the OS is not working on behalf of the application, it introduces performance artifacts, what has come to be referred to as "noise." The question of whether noise poses a significant slowdown in very large scale environments has been investigated by several previous works [17, 9]. Excellent research has been done to uncover the nature of OS noise and to relate it to scalable application performance [5, 21]. However, little of this work has tried to quantify the OS noise directly, in vivo with application-level measurements. Our objective was to examine the problem of noise measurement and charecterization, provide measurement and analysis methodologies and noise quantification metrics that provide new insights into how applications react to the different noise sources in an OS.

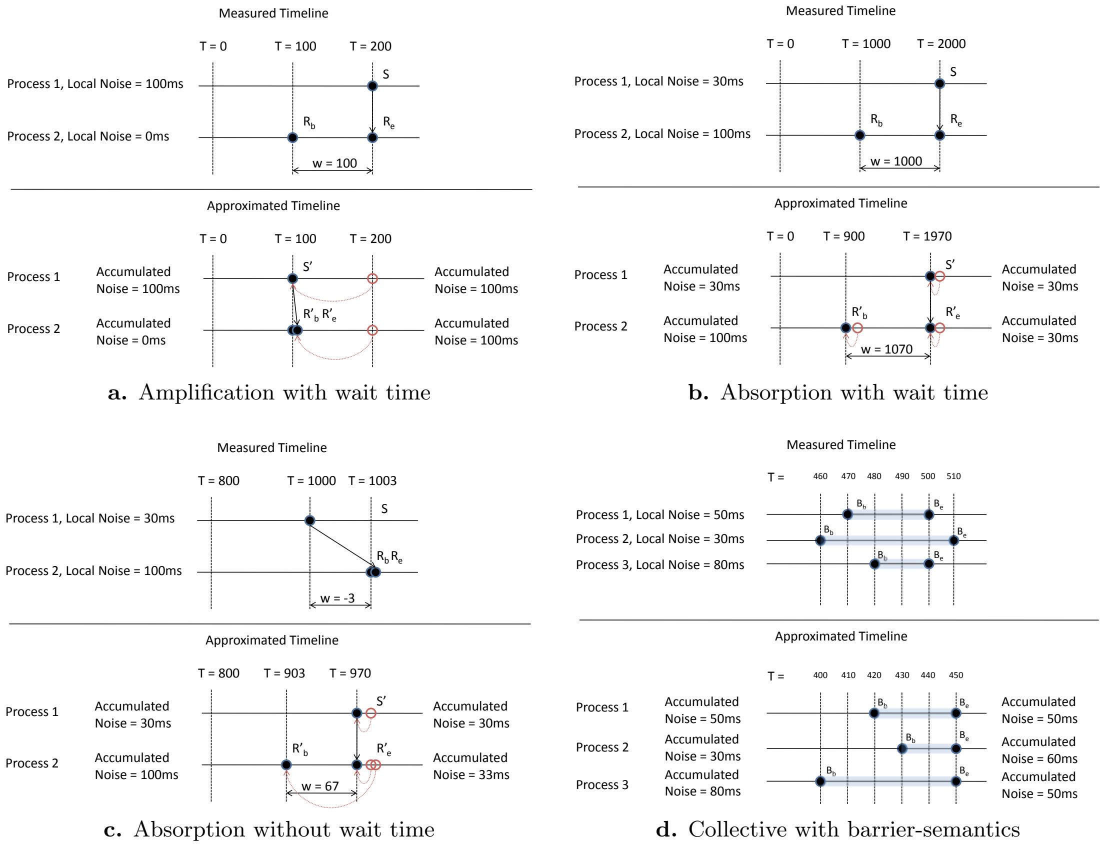
The KTAU project, and several others, have shown that the Linux kernel can be instrumented and measured. The question we pose here is whether OS noise can be measured and analyzed to understand its performance effects. The noise measurements require identifying Linux kernel operations that occur independent of application involvement. In the following experiments, we limit ourselves to known sources of noise that come from timer and scheduling related OS activity. These include:

- *timer_interrupt*: This is a periodic hardware interrupt (triggered by a clock source) to keep time in the system and manage timers. The interval is defined at Linux kernel compile time and has common values of 10, 4, and 1 msec. The *timer_interrupt* is a "global" timer interrupt – it is delivered to only one processor/core per system.
- *smp_apic_timer_interrupt*: This is also a periodic interrupt that has the same frequency as *timer_interrupt*. In contrast, this is a "local" interrupt – it is delivered to every processor/core. The main purpose is to update process times for scheduling.
- *schedule*: This represents "pre-emptive" scheduling only, in contrast to yielding the processor voluntarily. The value is the total time that a process spends not executing as a result of being pre-empted by another runnable process. Process quantums and process *pinning* can affect it.

In the noise experiments we report below, KTAU was configured to measure the above three noise metrics and provide access to the TAU performance system. Note that KTAU is not limited to tracking and promoting these events alone – any OS-event can be tracked in the same manner. TAU

NPB LU Class C on 16 Nodes - Total Exec. Time (sec)					
Metric	*base*	*ktau-only*	*ktau-proc*	*ktau-tau*	*ktau-tau-metrics*
Minimum	475.04	475.53	476.73	479.34	479.66
% Min Slowdown	—	0.10	0.36	0.91	0.97

Table 2: LU overhead results comparing KTAU-instrumented kernel and metric access under different measurement scenarios.



a. Amplification with wait time

b. Absorption with wait time

c. Absorption without wait time

d. Collective with barrier-semantics

Figure 2: Noise Estimation Illustrated

makes performance measurements of instrumented application level events during execution. If KTAU metrics are available for an experiment, those metrics will be measured and stored with the TAU performance data. TAU can generate parallel performance profiles or traces.

Parallel profile analysis in TAU reports statistical performance information for each thread of execution. Depending on what events were instrumented and the level of event callpaths measured, TAU profiling can provide a rich accounting of performance per event. Because KTAU metrics provide TAU with an additional performance data source, kernel noise statistics and their distribution will be shown for all parallel threads and events.

The interesting question is whether parallel profiles of kernel noise are enough to understand how noise affects performance. TAU computes both inclusive and exclusive data, which together with callpath- and phase-profiling [12], affords attribution to application components at any level of event detail. While profiles may be enough to uncover noise determinants to application performance, they do not capture noise dynamics and will lack explanative power.

On the other hand, TAU tracing will record a timestamp record for every event occurrence for every process during execution. With KTAU enabled, each event record will included the three noise metrics. Noise research has shown that performance effects are determined by the application context when and where noise occurs, and different applications with show different noise sensitivities (both complimentary and adversarial). Noise can accumulate and amplify execution time. Noise can also be absorbed and have little consequences. Trace analysis is important to explain these differences.

The trace analysis algorithms we developed are extensions of prior work on parallel measurement overhead compensation [11, 23]. The basic idea is to construct an *approximated* timeline without the influence of noise. Local noise is accumulated as we process it in the trace. At every event entry or exit, the local noise values are stored in the trace. We maintain an accumulated noise value for each process. This value will accumulate all local noise during non-communication events. At communication events, the accumulated noise can go up (amplification) or down (absorption) based on the analysis.

To illustrate, we will look at four examples. Figure 2(a) shows the simplest case of noise amplification. Before the communication events, Process 1 has accumulated 100ms of noise, and Process 2 has accumulated no noise. We construct the *approximated* timeline and reason that Process 1's send event (S) would have occurred 100ms previous in the absence of noise. The start of Process 2's receive event (Rb) would not have changed, since Process 2 had no local noise. However, we can now reason that the end of the receive event (Re) would have occurred earlier because it is no longer delayed by Process 1 (which now sends earlier). Having established the *approximated* timeline, we determine the accumulated noise for each process simply by subtracting the *approximated* exit time from the *measured* exit time. For a send event, no accumulation or absorption is possible, so Process 1 will maintain its 100ms of accumulated noise. Process 2 would have finished this receive 100ms earlier, so it's accumulated noise is now 100ms. In this case, we track 100ms of noise amplification on Process 2.

Figure 2(b) shows a case of noise absorption. As before, we construct the *approximated* timeline by moving the start events back by the current accumulated noise. We see in this case that Process 2 has a *measured* exit of 2000ms and an *approximated* exit of 1970ms, indicating that it has an accumulated noise of 30ms. Here, 70ms of the 100ms of local noise on this process has been absorbed. This is possible due to the 1000ms of *measured* wait time.

Figure 2(c) shows a case where there is no wait time. The send occurs before the receive starts. Again, we move the start times back by the current accumulated times. Then we reason about where the receive exit can occur. It can occur no earlier than the send. Since the send will now occur at time 970, Process 2 could exit as early as then. Since the *measured* exit was 1003, we have 33ms of accumulated noise after this receive event. So Process 2 has absorbed 67ms of it's 100ms of noise.

Figure 2(d) shows a collective operation with barrier semantics. The *measured* barrier begin (Bb) and end (Be) times are shown for each process. The algorithm for collectives is much the same. We construct an *approximated* timeline by moving the entry points back by the current accumulated noise values. Next, we examine the *measured* trace to determine the *sync time*. As an approximation, we compute the *sync time* to be the difference between the last *measured* entry and the last *measured* exit. After determining the *approximated* entry times, we approximate the exit times by adding the *sync time* to the last approximated entry.

The trace analysis maintains the amount of *accumulated* noise and the amount of local noise amplified and absorbed for each process for every event. At the end of the trace, the accumulated noise represents the amount of the total execution time due to noise. The analysis can be conducted for the entire noise (sum of the noise metrics) or for each noise metric separately, allowing an estimate of performance change if only that metric were removed.

4. EXPERIMENTAL RESULTS

A series of experiments were performed to demonstrate and evaluate the KTAU-based noise measurement and analysis methods. We begin with the validation of the noise tracking methodology at scale, using a synthetic bulk synchronous parallel program with noise injection on a BG/L platform. Second, the LU and CG benchmarks from the NAS NPB [4] were tested to demonstrate the extent of noise information that can be obtained with just profiling and the visualization of this information with existing tools in the TAU performance system. Finally, noise-tracking experiments were conducted with the Sweep3D benchmark to highlight the ability of our noise trace analysis and the efficacy of our metrics.

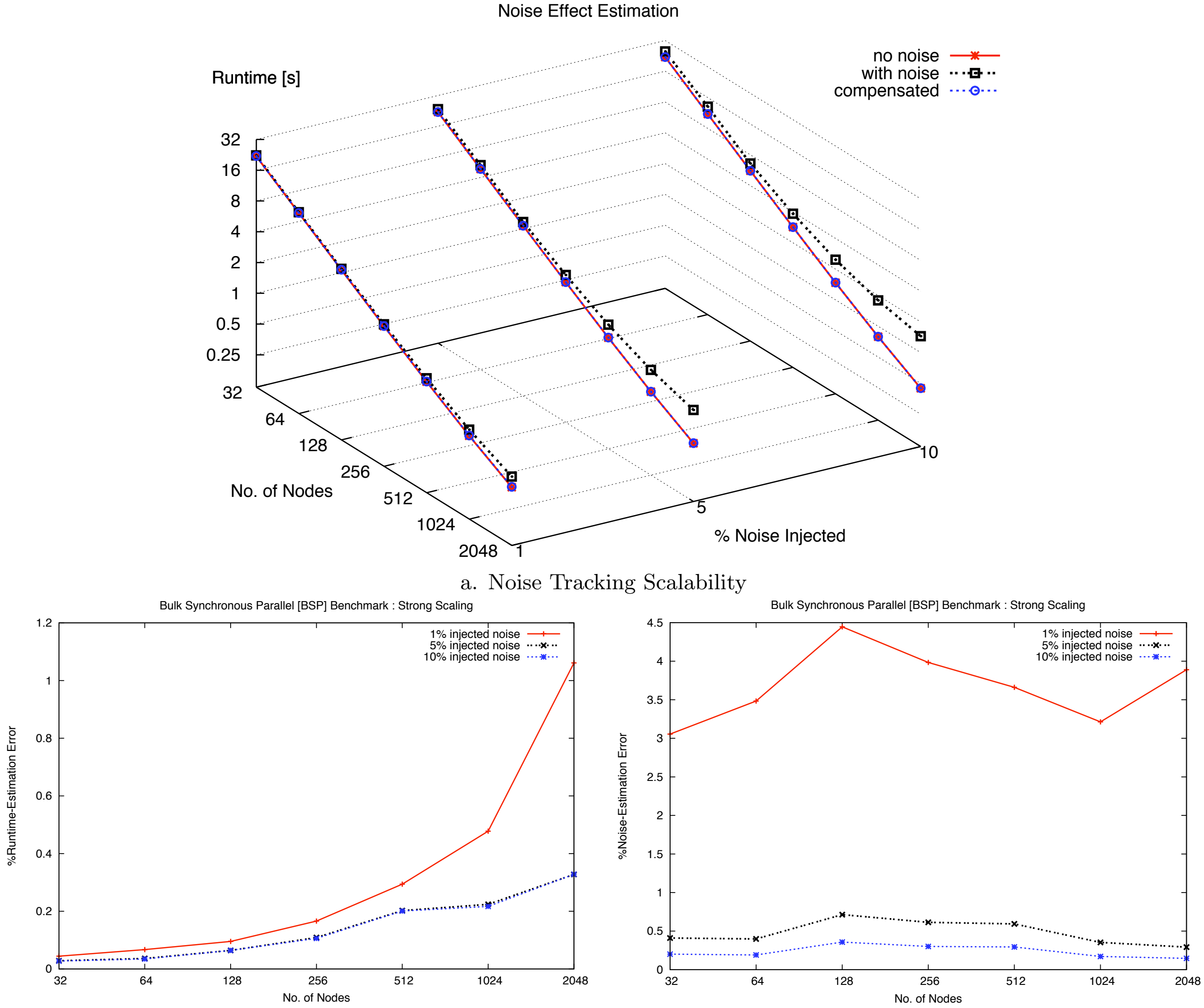
4.1 Empirical Validation of Noise-Tracking

A fundamental problem in validating a methodology that seeks to estimate slowdown due to noise is that in any intrinsically noisy environment it is not possible to remove all the existing noise. After all, that is the original problem this measurement tool and analysis methodology are seeking to aid in solving. One approach is to take an existing noise-less environment, introduce noise into it, and then compare the runtime of the parallel job with and without noise. This is the approach followed by Beckman et. al. [5] in their study of the effect of noise on collectives on the IBM BG/L machine. We adapt their methodology and Selfish noise-injection suite [5] to validate our noise analysis techniques at a larger scale.

We modified Selfish to export a global counter that identifies exactly how much noise has been injected (akin to how KTAU measures Linux kernel noise and publishes noise metrics). This Selfish counter is read and stored by TAU at application events (mainly, MPI communication events). The trace analyzer processes the traces containing Selfish-injected noise information, delivering an *accumulated noise* at the end for each process. We can use this value to predict execution time in the absence of Selfish-injected noise. This compensation estimate can be compared against the noise-less run to determine trace analysis accuracy.

We use a bulk-synchronous parallel (BSP) benchmark as this category of codes suffers significant noise-related slowdowns. This BSP program has two main phases, *computation* and *collective*, and repeatedly iterates over them. It allows control of the computational grain and can be run in either strong or weak scaling modes. In strong scaling mode the total work is split across the processors, which results in the computational grain being halved for every doubling of node-count. The computational-grain remains the same in weak-scaling mode (essentially increasing the total work with the node-count). We repeat the runs with the following configurations:

- *no noise*: It is run without any external noise injection.
- *with 1% noise*: Noise is injected at a frequency of 1000HZ with every noise event costing 10 microseconds. This translates to a total noise of 1%.
- *with 5% noise*: Similar to the 1% noise case, except that every noise event costs 50 microseconds.
- *with 10% noise*: Similar to the 1% noise case, except that every noise event costs 100 microseconds.



a. Noise Tracking Scalability

b. *%Runtime-Estimation Error(%RE)*

c. *%Noise-Estimation Error(%NE)*

Figure 3: Validation of Noise Tracking with Injected Noise

Experiments are run under the strong-scaling mode for 10000 iterations with total work set to take a duration of 64000 microseconds per computational phase over 32, 64 upto 2048 processors. This results in a computational grain of 2 milliseconds over 32 processors and 31.25 microseconds over 2048 processors respectively. The values were chosen so as to exacerbate the noise-problem as scale increases. Figure 3(a) shows the results of the runs for each node-count and amount of injected noise. The *no noise* and *with noise* curves are already explained. The points on *compensated* curves are a result of subtracting the *accumulated noise* (as reported by the trace-analysis) from the runtime of the corresponding *with noise* experiment. As can be seen, for every *noise %*, the *compensated* curves are almost entirely overlaid over the *no noise* curves.

We define two metrics, to further quantify the accuracy of the noise-tracking. The *%Runtime-Estimation Error (%RE)* is defined as:

$$\%RE = \frac{no\ noise\ runtime - compensated\ runtime}{no\ noise\ runtime} * 100$$

and *%Noise-Estimation Error (%NE)* is defined as:

$$\%NE = \frac{dilation\ due\ no\ noise - accumulated\ noise}{dilation\ due\ to\ noise} * 100$$

Figure 3(b) shows *%RE* to be low throughout (with the highest value being 1.06 for 1% injected noise). But it shows an increasing trend toward larger scales. In contrast the *%NE* (Figure 3(c)) is stable across the node-counts. This suggests that the error is a function of the total noise in the system and the increasing *%RE* is an artifact of the exponential increase in noise-related slowdown as compared to the *no noise* value. The accuracy is also shown to improve with greater injected noise (for the noise distribution tested). It is noteworthy that the analysis under-estimated noise in all experiments.

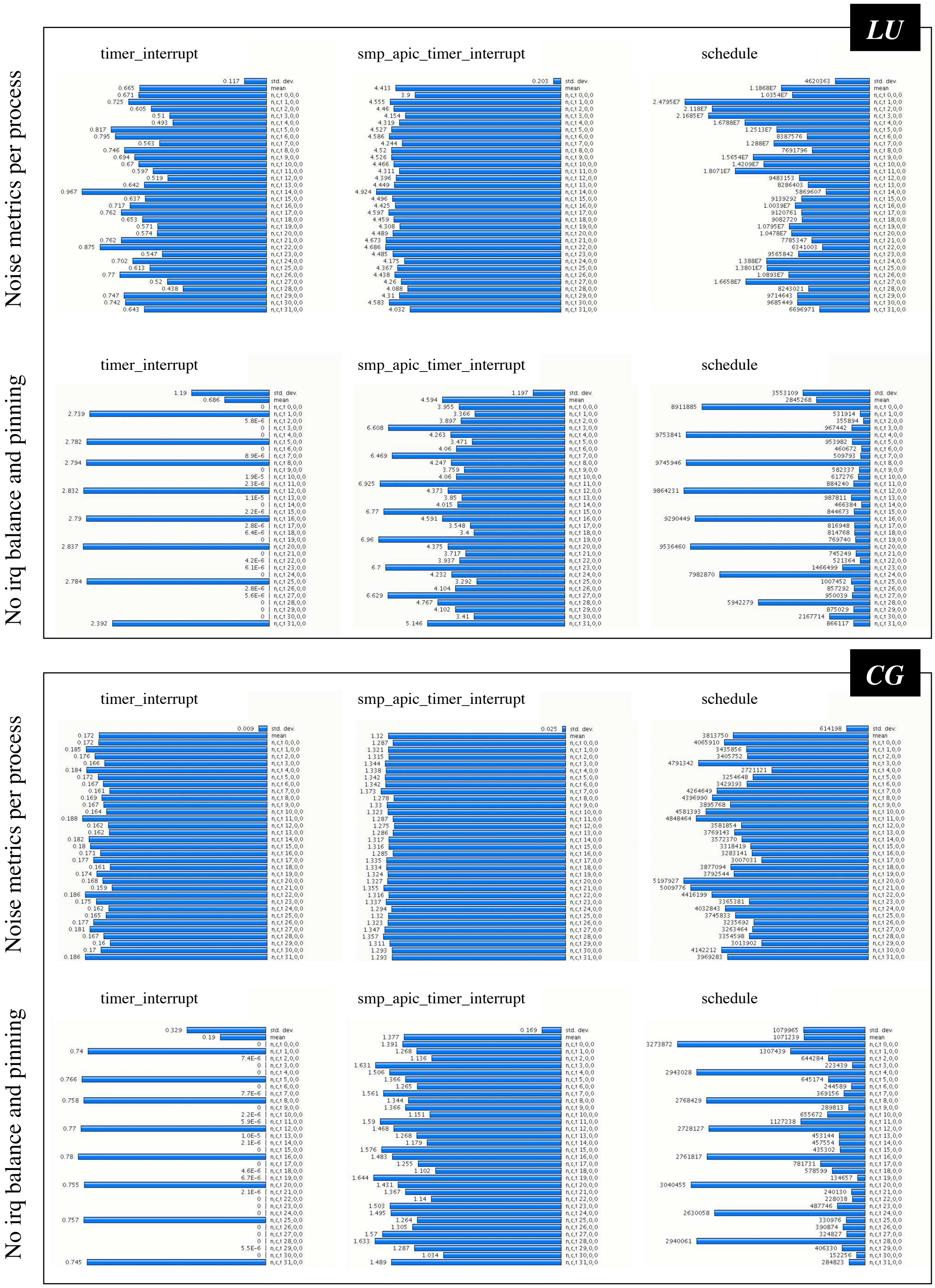


Figure 4: Kernel Noise Profiles of NPB LU and CG

4.2 LU and CG Noise Testing

We ran the NAS Parallel LU and CG benchmarks and performed noise-component studies. We use those runs to report on the integrated profiling capabilities. The top-half of Figure 4 shows NPB LU running on 32 processors under two different configurations, baseline and pinned/non-irqbalanced. The figures are screenshots of the ParaProf [6] parallel performance visualization tool. Each bar represents the total amount of a single type of noise-metric experienced by one application MPI rank. The leftmost metric is the global *timer_interrupt*, the center is the local *smp_apic_timer_interrupt* and the right one denotes the pre-emptive *schedule*. All units are seconds, except *schedule* which is reported as microseconds.

The baseline configuration view for LU shows how certain noise-sources (*smp_apic_timer_interrupt*) are evenly spread across the ranks, while others (*timer_interrupt*, *schedule*) are not. It is important to keep in mind that the high-level profile, in contrast to the trace, only provides a view of the overall application run - so it may not be able to see if distributions of these noise-components were uniform within any particular phase, but only overall. Other types of profiling such as Phase-based profiling, which partitions the measurement data across the phases and hence preserves that information may be used. The pinned/non-irqbalanced configuration shows several repeating features. One in four ranks has high *timer_interrupt* overhead, one in four has higher *smp_apic_timer_interrupt* and *schedule* as well. But the ranks which experience a high value of one metric (e.g. *schedule*) are different from those that experience another (say *timer_interrupt*). Why? Tasks stick to their cores due to pinning. In addition, due to lack of irqbalancing, the global timer is experienced by only one in four ranks (as there are 4 cores per node). This results in the peculiar pattern and serves to show how these interactions can be revealed with integrated performance profile views. Small changes to system/OS parameters can have significant effects on the parallel application. Similar features are seen in the case of CG in the bottom of the figure. Next, we turn to our trace-based methodology for greater insight into noise dynamics.

4.3 Sweep3D Noise-tracking Experiments

The ASCI Sweep3D benchmark [10] is an interesting test case for noise analysis because of its alternative phases of computation and communication. We set up the experiment to run Sweep3D on a problem of size 650x650x650 for 15 iterations with MPI-based communication. The San Diego Supercomputing Center provided access to a test cluster consisting of 32 dual-socket, dual-core (Opteron) nodes connected by Gigabit Ethernet. Since our goal was noise tracking, application-level tracing was performed using TAU for the MPI level operations with KTAU noise metrics recorded with each event. After trace collection, the parallel noise analysis algorithm was run to account for performance effects due to each noise component and their composition (overall noise).

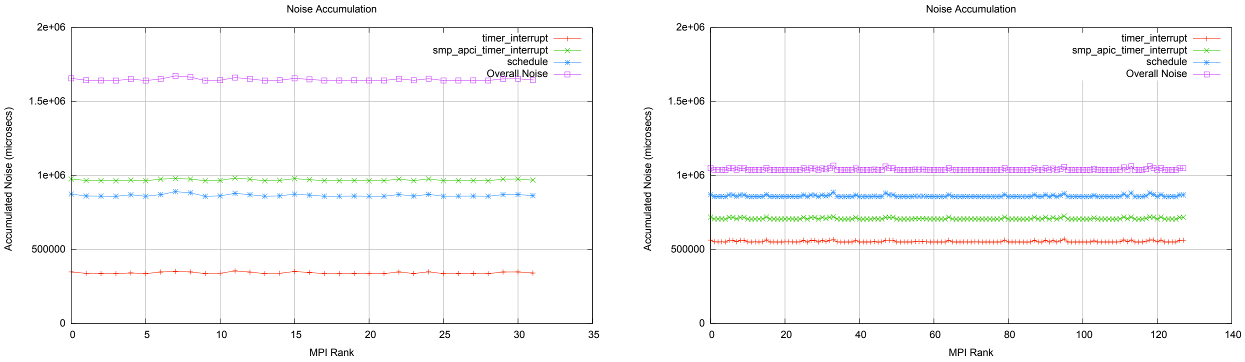
Three metrics (and corresponding graphs) are presented to help identify key noise features. The *accumulated noise* graph shows the total time lost by each process of the parallel application from noise, for each noise component and the overall noise. Subtracting this value from each process' total execution time gives an estimate of the process performance in the absence of noise. The *amplified noise ratio* graph shows the ratio of the *accumulated noise* to the *local noise* (i.e., how much noise is amplified). The *noise composition* graph shows the ratio of the *accumulated noise component* to the *overall accumulated noise*. This graph indicates how much a single noise source contributes to the overall noise and how the effects of the different noise-sources combine together.

We begin with a 32-processor run. Figure 5(a) plots the total accumulated time spent due to the different noise-sources on every rank. This time is a consequence of the local noise, additions due to noise passed on from other nodes and subtractions due to absorptions. Figure 5(a) shows that the application might have run 1.6 seconds faster if all three observed noise sources were removed from the system. It also shows that the largest contributor in this case is *smp_local_timer_interrupt*.

Figure 6(a) shows the sorted *noise amplification ratio*. *timer_interrupt* is shown to be largest amplified noise component. This can be explained by the fact that there is only coarse grained irq-balancing of the global *timer_interrupt* (every core is chosen to receive the timer interrupts for 10 seconds at a time). Because the total local noise duration due to *timer_interrupt* is small, the accumulated noise value for this component (Figure 5(a)) remains the lowest in-spite of the large amplification. It is important to note that the *overall noise amplification* for most ranks (except a few) is less than 1. This means that most noise was actually absorbed by the application and did not contribute to amplification.

Figure 7(a) plots the contribution of a particular noise component to overall noise. This is defined as the ratio of accumulated noise from a single source to the accumulated noise combined from the three sources. This view is meant to point out dominant noise sources. Additionally we plot a fifth curve labeled *Orthogonal Accumulated Noise Sum*, which is the ratio of the sum of the accumulated noise from the 3 sources to the combined overall accumulated noise. If this value is equal to 1, that means the effects of the three noise sources are orthogonal to each other and the overall effect is just a simple addition of the separate components. If this value is larger than 1 it means that some portion of one noise component was subsumed or overlapped with other noise and hence did not contribute fully to the combined noise. Values of less than 1 are not possible. In Figure 7(a) *smp_apic_timer_interrupt* and *schedule* are shown to be approximately 60% and 50% of the overall noise. It is interesting to note that 35% of the total noise from the different sources is subsumed or overlaid.

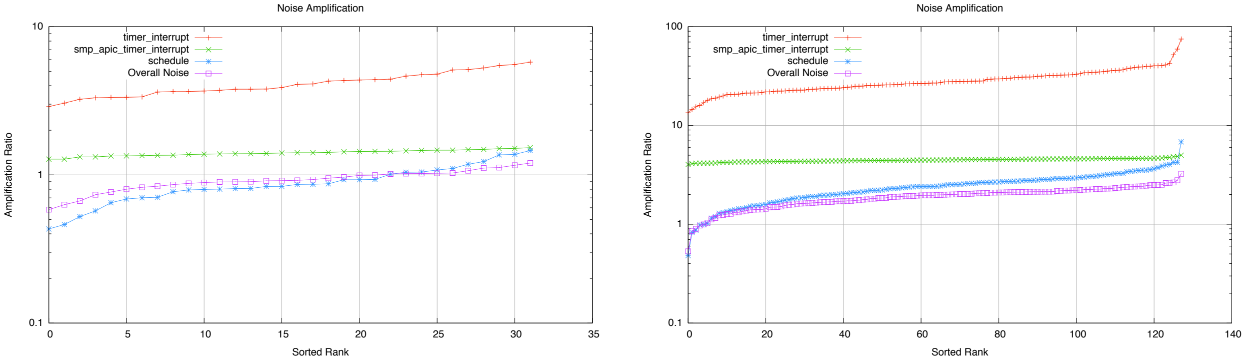
Next we investigate the effect of strong scaling on noise behavior by increasing the processor count to 128. Accumulated noise (as it is not normalized) does not directly compare to the 32 processor case. This is reflected in Figure 5(b), which seems to suggest that the 128 processor run experienced less noise than the 32 processor run (as it does not account for the lower run-time of the 128 processor run). The other two metrics shed further light. Figure 6(b) shows a significant amount of noise amplification of *timer_interrupt*. It also shows that the noise-amplification of the combined metric is largely above 1 and ranges upto 4, depicting significant parallel noise amplification as opposed to minimal (or zero) amplification in the case of the 32 processor run. Figure 7(b) shows that more than half the total accumulated noise from



a. N=32

b. N=128

Figure 5: Noise Accumulation (Base configuration)



a. N=32

b. N=128

Figure 6: Noise Amplification (Base Configuration)

the three sources is overlaid. It also shows that *schedule* is the dominant cause of noise and, if removed, would result in 85% of the noise effects being eliminated. Identification of the dominant noise source allows removing or reducing noise, without having to remove all noise sources.

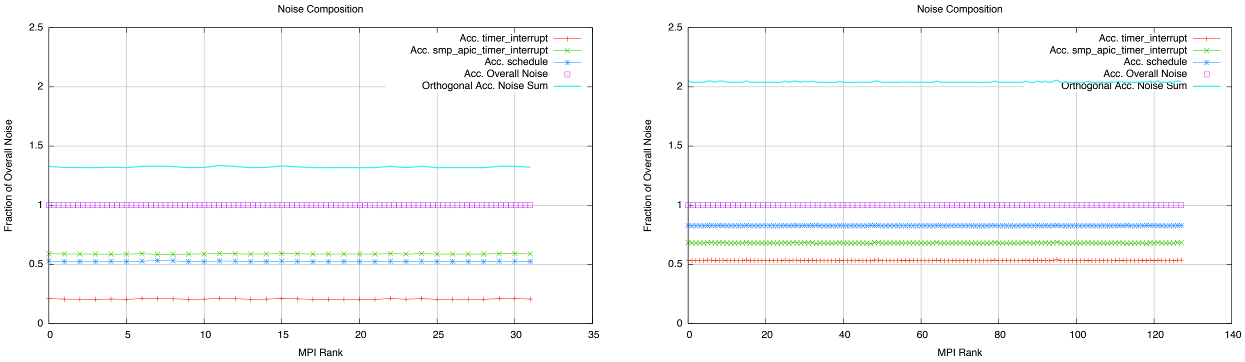
Having identified the dominant noise source as pre-emptive scheduling, we tested the pinning of the Sweep tasks to prevent their movement across cores. We also turned off irq-balancing of *timer_interrupt* to observe the effects. Figures 8 and 9 show the runs for 32 and 128 processors with the new pinned configuration. Comparing Figures 8(a) and 8(b), it is evident that the magnitude of noise reduces by the same factor as the speed-up (i.e., 4 times, going from 32 to 128 processors). This suggests that the noise problem was not exacerbated due to scaling (as happened in the base configuration). Figure 9(b) shows that *schedule* is no longer the dominant noise source, in fact, no single noise source is clearly dominant.

We have demonstrated our methodology in measuring and tracking parallel effects of individual noise components and overall combined noise, allowing identification of dominant noise sources (if any) and alleviation of noise effects due to scaling.

5. RELATED WORK

Trace-based analysis of message-passing programs has a long history. Some of the most recent work related to ours includes the DIMEMAS performance prediction tool [3] and Chama OS interference simulator [21]. The work we present in this paper addresses a problem not previously addressed by trace analyzers, that of understanding how measured interference propagates across all processors in a parallel program due to delays in synchronous operations. Doing so requires both trace analysis and a method for quantifying interference itself.

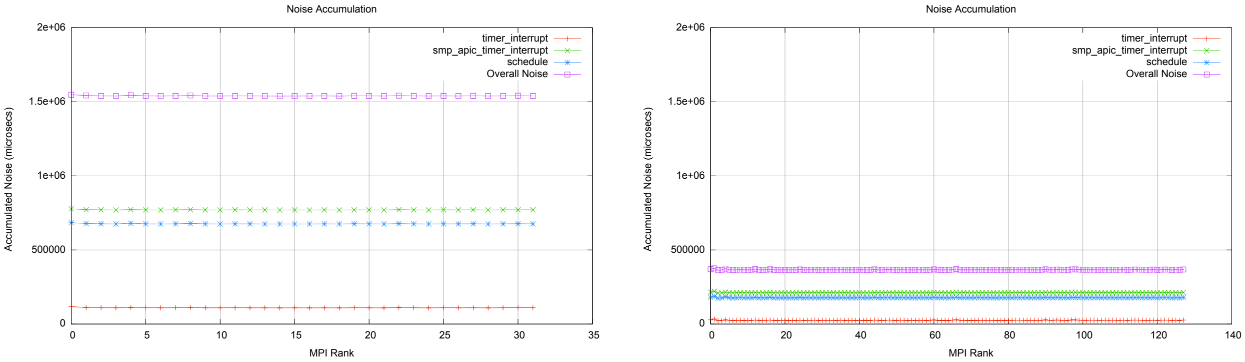
Prior work in quantifying interference has relied on micro benchmarks to infer the features of interference by observing its impact on simple programs. In 1994, Mraz presented work that analyzed variance in simple message passing programs to understand the impact of OS configuration on parallel programs [14]. His work could be argue to mark the start of quantitative noise measurement and mitigation. Later, the Fixed Time Quantum micro benchmark (FTQ) was created to quantify the effect of preemptive multitasking on single processor performance in a manner that was compatible with the application of rigorous mathematical analysis using tools from signal processing [20]. This allows one to reason about noise that delays arrival at synchronous operations on individual processors, resulting in a potential global program slowdown.



a. N=32

b. N=128

Figure 7: Noise Composition (Base configuration)



a. N=32

b. N=128

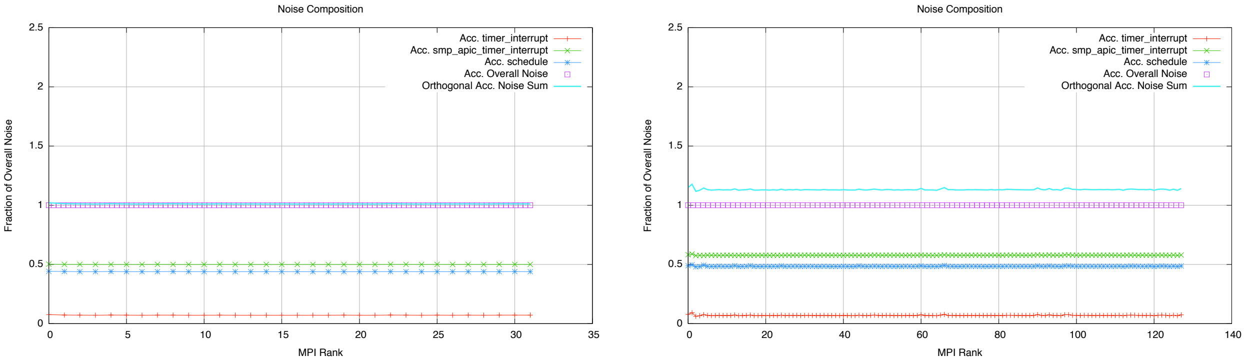
Figure 8: Noise Accumulation (Pinned configuration)

A different approach taken recently with great success is that of modeling parallel programs to predict their expected behavior in the presence of different sources and levels of noise. The work in tuning the ASCI Q cluster nodes that resulted in significant performance improvements demonstrated the success of this methodology [17]. Our work differs from this in that instead of relying on a performance model for both applications and machines, we base our study purely on observations of the actual machine and application execution. The use of the actual program is key in reasoning about the performance of complex applications with highly input dependent behavior, such as adaptive mesh refinement or time varying data decompositions (eg: molecular dynamics and n-body codes).

It has long been recognized that a complete system performance analysis requires measurements of OS behavior in addition to that of applications. Existing OS measurement efforts have included the Linux Trace Toolkit (LTT) [24], Solaris DTrace framework [7], and KernInst [22] projects. In addition DeBox [18], CrossWalk [13] and the tool described in [19] specifically try to correlate application/kernel performance. While these projects do provide access to kernel profiling data, they do not offer a straightforward mechanism for observing *all forms of kernel – application interaction.* In particular, they provide *no* support for observing ***asynchronous*** *kernel – application interactions* such as interrupts or scheduling. Instead, they focus primarily on the system-call interfaces (and those kernel routines directly invoked by the system-calls). Unfortunately, because of these limitations, these tools are of little utility in noise-tracking. Noise is a phenomenon that is largely a consequence of asynchronous OS interference amplified by parallel application communication patterns. The ability to unify OS and application observations (for both synchronous and asynchronous events) is vital for understanding how the full system behaves under real workloads. The KTAU framework discussed in this paper addresses this issue. Other projects such as MAGNeT [8] provide kernel profiling data, but for specific regions of the kernel such as the network protocol layer, limiting their applicability to full-system analysis and tuning activities.

6. CONCLUSION AND FUTURE WORK

Extending KTAU to derive kernel metrics and map them to application memory for fast access has created a powerful new capability for observing kernel operation and understanding its effects on application performance. In particular, the KTAU metrics support has allowed us to probe sources of noise and quantify their behavior in the context of application-level events. The intimate relationship of noise and communication operations, especially with re-



a. N=32

b. N=128

Figure 9: Noise Composition (Pinned configuration)

spect to how noise influences performance scaling, requires event-specific noise measurement and analysis to fully capture noise dynamics.

While we believe an integrated performance environment combining kernel-level and application-level measurements is important for next-generation parallel systems, we recognize the reluctance to support instrumentation in the OS kernel on production platforms. Nevertheless, our future plans include the inclusion of KTAU and its extension in Linux-based OS distributions for high-end, extreme-scale parallel environments. In the course of this work, we believe we can prove the worth of an integrated performance framework for addressing challenging problems such as noise assessment, noise elimination, and dynamically adaptive OS/R components that require online kernel performance data that KTAU can provide. Our first target will be new DOE leadership class facilities where Linux is being pursued for a lightweight computer node kernel.

7. ACKNOWLEDGMENTS

This research is supported by the U.S. Department of Energy, Office of Science, under contracts DE-FG02-05ER25663 (Extreme Performance Scalable Operating Systems) and DE-FG02-05ER25680 (Application-Specific Performance Technology for Productive Parallel Computing). We thank Don Thorp and the San Diego Supercomputing Center for their generous support and access to system resources for experimentation. We also thank Argonne National Laboratory for providing access to their clusters.

8. REFERENCES

[1] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/projects/papi/.

[2] TAU: Tuning and Analysis Utilities. http://www.cs.uoregon.edu/research/paracomp/tau/.

[3] R. M. Badia, J. Labarta, J. Giménez, and F. Ascalé. DIMEMAS: Predicting MPI applications behavior in Grid environments. In *Workshop on Grid Applications and Programming Tools (GGF8)*, 2003.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] P. Beckman, K. Iskra, K. Yoshi, S. Coghlan, and A. Nataraj. Benchmarking the Effect of Operating System Interferences on Extreme-Scale Parallel Machines. *IEEE Cluster Computing Journal.* to appear.

[6] R. Bell, A. D. Malony, and S. Shende. A portable, extensible, and scalable tool for parallel performance profile analysis. *Lecture Notes in Computer Science*, 2790:17–26, 2003.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04: Proceedings of the 2004 USENIX Annual Technical Conference*, page 13, Boston, MA, USA, 2004. USENIX.

[8] W. Feng, M. K. Gardner, and J. R. Hay. The magnet toolkit: Design, implementation and evaluation. *Journal of Supercomputing*, 23:67–79, August 2002.

[9] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. 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*, page 10, Washington, DC, USA, 2003. IEEE Computer Society.

[10] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65:198–199, 1992.

[11] A. Malony and S. Shende. Overhead Compensation in Performance Profiling. In *EuroPar'04: European Conference on Parallel Processing*, pages 119–132, Sept. 2004. (Best paper award).

[12] A. Malony, S. Shende, and A. Morris. Phase-based Parallel Performance Profiling. In *PARCO'05: Conference on Parallel Computing*, Sept. 2005.

[13] A. Mirgorodskiy and B. P. Miller. Crosswalk: A tool for performance profiling across the user-kernel boundary.

[14] R. Mraz. Reducing the variance of point to point transfers in the IBM 9076 parallel computer. In *SC'94: ACM/IEEE Conference on Supercomputing*, 1994.

[15] A. Nataraj, A. Malony, S. Shende, and A. Morris. Integrated parallel performance views. *IEEE Cluster Computing Journal*. to appear.

[16] A. Nataraj, A. Malony, S. Shende, and A. Morris. Kernel-Level Measurement for Integrated Parallel Performance Views: the KTAU Project. In *IEEE Conference on Cluster Computing*, Sept. 2006. (Best paper award).

[17] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of AS CI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16 2003.

[18] Y. Ruan and V. Pai. Making the "box" transparent: System call performance as a first-class result. In *USENIX '04: Proceedings of the 2004 USENIX Annual Technical Conference*, page 15, Boston, MA, USA, 2004. USENIX.

[19] S. Sharma, P. G. Bridges, and A. B. Maccabe. A framework for analyzing linux system overheads on hpc applications. In *LACSI '05: Proceedings of the 2005 Los Alamos Computer Science Institute Symposium*, page 17, Santa Fe, NM, USA, 2005.

[20] M. Sottile and R. Minnich. Analysis of Microbenchmarks for the Performance Tuning of Clusters. In *Cluster'04: IEEE Conference on Cluster Computing*, 2004.

[21] M. J. Sottile, V. P. Chandu, and D. A. Bader. Performance analysis of parallel programs via message-passing graph traversal. In *IPDPS'06: 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006.

[22] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI'99: Operating Systems Design and Implementation*, pages 117–130, 1999.

[23] F. Wolf, A. Malony, S. Shende, and A. Morris. Trace-based Parallel Performance Overhead Compensation. In L. T. Yang, et. al., editor, *HPCC'05: High Performance Computation Conference*, volume LNCS 3726, pages 617–628. Springer, Sept. 2005.

[24] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX'00: USENIX Annual Technical Conference*, Boston, MA, USA, 2000.