System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications

Dan Tsafrir Yoav Etsion Dror G. Feitelson Scott Kirkpatrick
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
{dants,etsman,feit,kirk}@cs.huji.ac.il

ABSTRACT

As parallel jobs get bigger in size and finer in granularity, "system noise" is increasingly becoming a problem. In fact, fine-grained jobs on clusters with thousands of SMP nodes run faster if a processor is intentionally left idle (per node), thus enabling a separation of "system noise" from the computation. Paying a cost in average processing speed at a node for the sake of eliminating occasional processes delays is (unfortunately) beneficial, as such delays are enormously magnified when one late process holds up thousands of peers with which it synchronizes.

We provide a probabilistic argument showing that, under certain conditions, the effect of such noise is linearly proportional to the size of the cluster (as is often empirically observed). We then identify a major source of noise to be indirect overhead of periodic OS clock interrupts ("ticks"), that are used by all general-purpose OSs as a means of maintaining control. This is shown for various grain sizes, platforms, tick frequencies, and OSs. To eliminate such noise, we suggest replacing ticks with an alternative mechanism we call "smart timers". This turns out to also be in line with needs of desktop and mobile computing, increasing the chances of the suggested change to be accepted.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; C.4 [**Performance of Systems**]: Modeling Techniques, Measurements Techniques

General Terms

Performance, Measurement, Experimentation, Design

Keywords

Modeling system noise, HPC, operating systems, timing services, ticks, timer interrupts, smart timers, synchronization

1. INTRODUCTION

Clusters are becoming very popular for high-performance computation (HPC) and range in size from tens to thousands of nodes. Such systems typically employ a general-purpose operating system (OS) on each node (e.g. in last year's Top500 supercomputer list [5] ten of the top twenty machines ran Linux). Parallel jobs are usually executed on such clusters by spawning one process per CPU, and running to completion with no interference [9]. HPC applications are often *bulk-synchronous* which means each participating process is composed of iterative computation phases separated by barriers, where speedy processes wait for lagging ones to catch up. The *granularity*, or time it takes to complete a single computation phase, can be a millisecond or even less for real world applications [25].

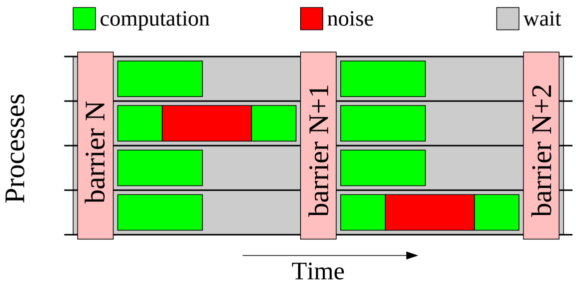

Figure 1: *Due to synchronization, each computation phase is prolonged to the duration of the slowest process.*

It is within this context that systems began to experience great difficulty in scaling applications to make use of hundreds to thousands of compute nodes [26, 16, 25, 29]. Phillips et al. [26] noted that the problem stems from a variability in the duration of computation phases: They observed that while in most cases phases take a relatively constant time to complete, they are sometimes prolonged. The delay is arguably unnoticeable for a sequential application. But in the context of a bulk-synchronous job, the price is dramatically amplified, as all the other processes across the cluster must wait for a delayed process to catch up. This effect is commonly illustrated [26, 16, 25, 29] as in Fig. 1. Phillips et al. noticed that the variability problem is greatly alleviated if one processor in each (4-way) node is left idle. They concluded that "the inability to use the 4th processor on each node for useful computation" is a major problem.

A year later, Petrini et al. [25] faced a similar problem with an application running on the ASCI-Q, that actually terminated sooner when utilizing only three processors of the available 4-way nodes. They managed to explain this phenomenon when discovering that variability is due to system *noise*, that is, per-node system activity, which is largely unrelated to the parallel application. Such activity is assigned to an idle processor if such exists. However, if all the processors are assigned to processes belonging to the paral-

ICS '05, June 20-22, Boston, MA, USA

lel job, one will be preempted in favor of the system activity and will be forced to wait until the system activity is done. This delay was traced to be the major source of variability.

The first part of this paper (Section 2) suggests a simple theoretical model that quantifies this effect of noise, regardless of its source. Each process of a parallel job has some probability to be delayed due to noise while it is computing on its home node. If this probability is small enough, we show that the harmful impact of noise (on the entire job) grows linearly with the cluster size. In fact, the job's probability to be delayed is simply a multiple of the cluster size and the single-node probability to suffer from noise. Linear performance degradation is indeed reported by several empirical studies that have measured the impact of noise as a function of cluster size [16, 25].

System noise is generated due to two types of activities. First, there are various system daemons that wake up once in a while. In Linux, these may include kswapd and bdflush (deal with swapping), ntpd (synchronizes clock), inetd (serves requests for rsh, telnet etc.), nfsd (file system) and so on. Additionally, within a cluster, there is usually at least one per-node daemon as part of the cluster infrastructure. The second type of noise-generating activity are interrupts handled by the OS, which can be responsible for up to two thirds of the slowdown incurred on applications by noise [25]. It is this type of noise that is the focus of the second part of our paper (Section 3).

The most common interrupts are network and periodic clock ticks. The former may be generated upon events such as arrival of data, when buffers are full etc. The latter serve as OS means of measuring passage of time, providing timing services, and maintaining control.

Periodic clock ticks are with us since the birth of general-purpose OSs, 30 years ago: At boot time, a general-purpose kernel sets a hardware clock to generate periodic interrupts every few milliseconds (this constant time interval is called a *tick*). The interrupts invoke a kernel routine responsible for important OS activities such as accounting for the CPU time used by the current process, designating it for preemption if its quantum is exhausted, or notifying the process if it has pending signals. Until recently, 100 Hz was the default tick frequency, used by *all* Windows and Unix flavors. This value hasn't changed much since the dawn of general purpose OSs, e.g. back in 1976, Unix 6 running on a PDP11 used a tick rate of 60 Hz [20]. Recently, it has been shown that a 100 Hz rate is inadequate for multimedia applications [7], and so the last year has seen a growing trend of increasing the tick rate to 1000 Hz (Linux, FreeBSD, DragonFlyBSD).

Section 3 shows that the indirect overhead of ticks (the cache misses they force on applications) is a major source of noise suffered by parallel fine-grained jobs.

The practical meaning of ticks is that general-purpose OSs are software components that are actually based on *polling.* This may have been a good design decision in the 1970s, but things have changed since. Section 4 notes that ticks-related problems manifest themselves in several non-HPC domains. These include mobile or autonomous systems that are uselessly wasting power on ticks that can be avoided, and soft realtime applications (e.g. multimedia) that receive inferior timing services, inherently limited by the tick resolution. The drawbacks of ticks are accumulating into a critical mass suggesting the price of polling is becoming too high, and that it's time to reconsider. Consequently, Section 4 concludes with the suggestion to replace ticks with an alternative novel mechanism called "smart timers", which eliminates tick noise as well as accommodating the needs of the other aforementioned domains.

Finally, the paper is finished with Section 5 that surveys related work and Section 6 that concludes.

2. MODELING THE EFFECT OF NOISE

We are interested in quantifying the effect of noise. As additional nodes imply a noisier system, one can expect the delay probability of the entire job to increase along with its size. Petrini et al. [25] assess the effect of noise through detailed simulation of its (per site) components. In contrast, this section analytically shows that if the single-node probability for delay is small enough, the effect of size is linear: it simply multiplies the single-node probability.

2.1 The Probabilistic Argument

Let n be the number of nodes allocated to a job. Given a node, let p be the probability some process running on it is delayed due to noise, within the current computation phase (a node may be an SMP, so it may run multiple processes from the same job). For simplicity, we assume independence and uniformity across nodes, namely, delay events on different nodes are independent of each other and have equal probability. Under these assumptions, the probability that no process is delayed on any nodes is $(1-p)^n$. Therefore,

$$d_p(n) = 1 - (1-p)^n \tag{1}$$

denotes the probability that the job *is* delayed within the current computation phase. Recall that p should be relatively small if we are to contain the negative effect of noise. If this is the case, we claim that

$$\bar{d}_p(n) = pn \approx d_p(n) \tag{2}$$

is a reasonable approximation. The rest of this subsection is devoted to finding out the constrains on p, in order for this approximation to be accurate, e.g. since $d_p(n)$ is a probability, p must surely be smaller than $\frac{1}{n}$.

Consider the difference Δ between the original function and its approximation

$$\Delta = \bar{d}_p(n) - d_p(n) = pn - 1 + (1-p)^n \tag{3}$$

Note that Δ is nonnegative (has positive derivative and is zero if $p = 0$). According to the binomial theorem

$$(1-p)^n = 1 - pn + \frac{n(n-1)}{2}p^2 + \sum_{k=3}^{n} \binom{n}{k} (-1)^k p^k \tag{4}$$

where the rightmost summation has a negative value[1]. Therefore, by combining Equations 3 and 4, we get that

$$0 \le \Delta < \frac{n(n-1)}{2}p^2 < \frac{n^2p^2}{2}$$

[1] Dividing the absolute value of the k and k+1 elements in the summation yields $\frac{1}{p} \times \frac{k+1}{n-k}$. Since $p < \frac{1}{n}$, the left term is bigger than n. The right term is bigger than $\frac{k+1}{n}$ which is bigger than $\frac{1}{n}$. The quotient is therefore bigger than 1, indicating the k-th element is bigger than its successor. Consequently, the summation can be subdivided into consecutive pairs in which odd k elements are negative, and are bigger (absolute value) than their even k+1 positive successors.

which means that Δ is bounded by some small ε if $p < \frac{\sqrt{2\varepsilon}}{n}$. For example, $\Delta < 0.01$ if $p < \frac{1}{7n}$. Additionally, the *relative* error of the approximation is smaller than (say) 5% if

$$\frac{\Delta}{\bar{d}_p(n)} < \frac{n^2p^2}{2np} = \frac{np}{2} \le 0.05$$

which holds if $p \le \frac{1}{10n}$. Another way of looking at this constraint is that the approximation is good as long as

$$d_p(n) \le 1 - \left(1 - \frac{1}{10n}\right)^n \approx 1 - e^{-\frac{1}{10}} \approx 0.1$$

The bottom line is that $d_p(n) \approx pn$ indeed serves as an intuitive linear "noise law" for small enough p.

2.2 Consequences of the Model

The $d_p(n)$ function (Eq. 1) attempts to quantify the probability to suffer from noise. The pn approximation (Eq. 2) states that the impact of noise on a parallel job is linearly proportional to the number of nodes it occupies, if p is small enough relative to $1/n$. Intuitively, under these conditions, the probability of overlap between the noise on two distinct nodes is negligible, so each additional node adds a probability of p to the probability that the whole job will be delayed. But if the condition is not met, and $\frac{1}{10n} \ll p$, we are nearly certain to experience noise on *some* node in each phase. In this case, more nodes will *not* introduce additional noise.

What would be a desirable value for p? Fig. 2 (bottom) plots $d_p(n)$ as a function of n for various p values. Evidently, if n is hundreds to thousands of nodes, than p should probably not exceed 10^{-5}, guaranteeing $d_p(n) \le 0.1$. To make noise-probability effectively zero across this range, p should be in the order of 10^{-6}, which is certainly preferable for machines of tens of thousand of nodes that begin to emerge (e.g. BlueGene/L). Generally, by the "noise law" approximation, systems should be designed such that $p \le \frac{\alpha}{n}$ in order to bound noise-probability below a small fixed threshold α.

To illustrate the accuracy of the "noise law" approximation, let us examine the point associated with $n = 10^3$ and $d_p(n) = 10^{-2}$ in the top of Fig. 2. Note that the curve that passes through this point is associated with $p = 10^{-5}$. This indicates that limiting the probability of delay due to noise endured by a thousand nodes job to be 1:100 requires a system with $p = 10^{-5}$, namely, that the probability of a process to be interrupted by noise on the node would be 1:100,000. In this example $p = 10^{-5} = \frac{10^{-2}}{10^3} = \frac{d_p(n)}{n}$ exactly as predicted by the approximate pn "noise law". This argument holds for any n and $d_p(n) \le \frac{1}{10}$ we choose.

2.3 Assumptions of the Model

Recall the probabilistic argument relies on the assumptions of independence and uniformity. Arguably, these assumption don't always hold. For example, as nodes occasionally communicate, one might expect that a message sent from one node would generate noise at the receiving end. However, the independence assumption may still be reasonable as supercomputers usually employ high-end communication networks (Myrinet or Quadrics) that provide dedicated processors for handling of network events, thus masking them from node processors. Additionally, noise activity that happens more or less simultaneously across nodes (e.g. conversing daemons), has less severe implications on the application, as the cost is amortized by the fact it is paid synchronously. It is therefore reasonable to expect that independent noise sources have worse implications on performance than dependent ones. Consequently, our model can serve as an upper bound on noise impact.

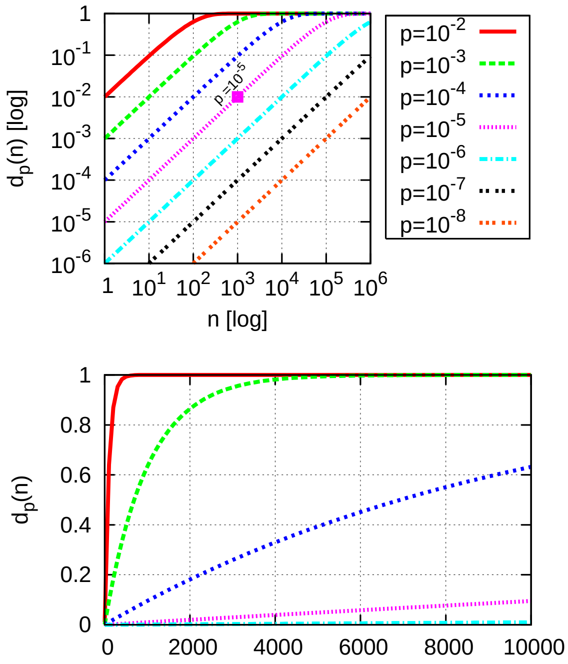

Figure 2: *Bottom figure shows the probability a job is delayed in the current computation phase, as a function of its size. This is the true value of $d_p(n)$ (Eq. 1), not the approximation. Top figure is the same with log scale applied to both axes.*

Uniformity is also not entirely true, e.g. some nodes may host special cluster daemons thus experience greater noise. However, this is expected to have marginal effect on our probabilistic argument as it requires only trivial changes that do not fundamentally alter the main result.

It is therefore our opinion that the model truly reflects the *nature* of noise, despite possible inaccuracies. This is satisfactory when seeking a general understanding of the phenomenon, rather than exact numbers. It is nevertheless possible dependent noise sources will require a different model that should somehow be combined with ours. In any case, we would like to point out that the noise component targeted by this paper (OS clock ticks) is both independent and uniform across nodes, as assumed.

2.4 Assessing Slowdown

So far, we have discussed the *probability* of a job to be delayed. The next step is to devise its actual *slowdown*, namely, the relative delay which noise components impose on the job. Firstly, note that our model implicitly assumes a Bernoulli delay distribution: either the delay occurred (p) or it didn't ($1 - p$). But reality might not be bimodal as often there are different probabilities for different delays. This is exemplified in Fig. 3 that shows the phase-duration CDF of one task that has G=1ms granularity, and is running on a uniprocessor node (this is the result of a simple microbenchmark; more details in Section 3). Considering the tail of this distribution, we note that the per-node chance for a 1.5ms phase or bigger is $p = 10^{-2}$. This can be used with our model to devise a loose lower bound on the slowdown: For example, if the size of the cluster is $n = 10$, then the linear version of the noise law applies (with respect to $p = 10^{-2}$), in which case $d_p(10) \approx pn = 0.1$ and therefore the expected slowdown factor is bigger than

$$(0.9 \cdot 1ms \; + \; 0.1 \cdot 1.5ms) \; / \; 1ms \; = \; 1.05$$

On the other hand, for an $n = 100$ cluster,

$$d_p(100) \; = \; 1 - (1 - 10^{-2})^{100} \; = \; 0.63$$

and therefore the slowdown is bigger than

$$(0.37 \cdot 1ms \; + \; 0.63 \cdot 1.5ms) \; / \; 1ms \; = \; 1.32 \qquad (5)$$

A generalization of this, that produces an accurate assessment of the slowdown, must take the whole distribution of phase durations into account. Let X_j be a random variable representing the phase duration of an application with granularity G on node j; assuming uniformity, $\{X_j\}_{j=1..n}$ all have the same distribution. Let $Y = \max\{X_j\}_{j=1..n}$. Since a computation phase is completed only after the latest task arrives, the expected value of Y represents the average phase duration of the whole job. Dividing this by the granularity gives the slowdown: $E(Y)/G$.

Note that the above procedure is simply a generalization of our model that assumes X_j is a Bernoulli variable: If G is the noise-free phase duration, and G' is the longer noisy duration, then $E(Y) = G \cdot (1 - d_p(n)) + G' \cdot d_p(n)$.

2.5 Model vs. Reality

Our linear approximation of the probability to be affected by noise seems to agree with measurements that have been conducted on a number of systems. For example, measurements taken on the 'ASCI-White' and 'Blue-Oak' of the all-reduce operation with an increasing number of nodes, show that performance degradation due to noise is linear (see Fig. 6 in [16]). Measurements of a barrier operation on the 'ASCI-Q' seem to follow the same pattern (see Figs. 7 and 16 in [25]). More interesting are measurements of the complete SAGE application on ASCI-Q, shown in Figs. 1 and 17 of [25]. A detailed (noiseless) model of this application [17] predicts that the phase duration should initially increase steadily with the number of nodes, but then it should stabilize when more than about 512 nodes are used. However, the actual performance curve shows that the duration continues to grow linearly in this range, adding ∼0.13 sec for every additional 1024 processors.

Based on the the detailed description of the 'ASCI-Q' noise sources in Fig. 13 of [25], we further conjecture that for $n \geq 512$ all fine grained frequent noise becomes a certainty (by $d_p(n)$). Thus, from that point on, results become dominated by the coarser, less frequent, noise. The probability of such noise is small enough for the linear approximated "noise law" to apply (per-node event every 1-2 minutes, on only 2 nodes out of each 32 'ASCI-Q' sub-cluster).

Based on the above, we claim that individually applying the $d_p(n)$ argument on various noise sources (very few according to [25]), is a feasible approach that can yield a reasonable assessment of noise impact.

3. MEASURING THE NOISE

The previous section has shown that the extent of noise experienced by a parallel job is a function of its size (in nodes) and p. The latter term expresses the probability a process is delayed within a computation phase, which is relatively easy to quantify. Our next step is therefore evaluating p along with the duration of delays (which will allow approximation of actual slowdown). As we do not have a working supercomputer at our disposal, we only deal with native OS noise. This turns out to be dominated by periodic clock ticks. To complete the picture, coarser noise generated by various cluster daemons should also be characterized.

3.1 Methodology

Calibration. The simplest manner in which one can measure p is to carefully calibrate a computation phase to take a certain amount of fixed time T_{grain} (which represents the granularity of the application). The computation phase should then be executed iteratively a large number of times, while measuring the *actual* duration. This would yield a distribution D from which p can be extrapolated. The problem with this approach (as anybody who ever attempted it would know) is that it is impossible to just "calibrate a computation" to take a predefined amount of time. This is actually the main problem we are facing, that is, constant work takes a variable amount of time to complete due to noise (i.e. any calibration loop is susceptible to noise). All one can do is choose a *heuristic* that is the most appropriate.

One possible heuristic is calibrating the computation such that the *minimum* is T_{grain} (suffered the least amount of noise and represents a quieter environment). Another is calibrating the *average* to be T_{grain} (probably closer to reality). The heuristic we chose was to try and place the vertical part of D's CDF (the part between the head and the tail; see Fig. 3) on the given T_{grain}. For this purpose we sequentially executed an empty loop of one million iterations (a "phase") for 100,000 times. We measured the duration of each phase using a cycle resolution counter [6]. Measurements were ascendingly sorted and averaged between the ten and twenty percentiles (2nd decile). This average was translated to actual time (by dividing it with the processor clock frequency) and served as the interpolation basis when approximating the number of per-loop empty iterations required to obtain a T_{grain} long computation.

Note that regardless of the heuristic chosen, what matters most is the *span* of D, namely, how it relates to its minium (shortest phase), which is closest to the noiseless optimum.

Micro Benchmarking. The micro benchmark we used to generate the D distribution is a simple program that gets a T_{grain} parameter, and performs a million computation phases (empty loops) calibrated to take T_{grain} time. The actual duration of phases is recorded in a preallocated array which is printed upon completion[2]. No other user process was running while the measurements took place. However, the machine ran the default-installation system daemons. Note that since "computation phases" are modeled as empty loops, the measured noise-impact actually constitutes an optimistic approximation, as phases are hardly vulnerable in terms of data locality (the only data used is the array).

Using an SMP to measure noise-impact for when it is fully utilized, requires running a benchmark instance on all pro-

[2] The array is initialized beforehand to make sure it is allocated in physical memory rather than using a single copy-on-write page, so that allocation will not effect the benchmark.

cessors, and merging of results upon completion (recall p is the probability that *at least one* phase is delayed, it doesn't matter in which of the node's processes). A simpler alternative is to run the benchmark on a uniprocessor, thus ensuring only one benchmark instance suffers all system noise.

Sample Space. The value of p and the D distribution are products of many factors. To establish our claim (regarding periodic OS ticks) as universal, we have chosen to examine various combinations of four such factors: architectures, OSs, grain sizes, and tick frequencies. The various (uniprocessor) platforms we have used are listed in Table 1. For convenience, each configuration is given an ID (*M1-M4*). Unless stated otherwise, the OS kernel is Linux-2.6.9 with the default 1000 Hz tick rate (SMP enabled, so that associated overheads are included). The rest of the factors that influence D are specified when relevant.

KLogger. When needed, kernel profiling was performed using *klogger*, a kernel logger we developed that supports efficient fine-grain events. While the code is integrated into the kernel, its activation at runtime is controlled by applying a special sysctl call using the /proc file system. In order to reduce interference and overhead, logged events are stored in a 64 MB buffer in memory, and only exported at the end of each benchmark instance. The implementation is based on inlined code to access the CPU's cycle counter and store the logged data. Each event has a header including a serial number and timestamp with cycle resolution, followed by event-specific data. The overhead of each event is only 90 cycles on P-III and roughly twice that much on P-IV. In our use, we log all interrupts and their outcome so that we have complete knowledge of system activity while klogger is on. Note that klogger and the benchmark use the (same) CPU's cycle counter and therefore their output can be combined into a global view of the system. Consequently, we are able to pinpoint which computation phases were interrupted, for how long, and the nature of interruptions.

3.2 Results

In this subsection we demonstrate the instability of the phase duration distribution and track the source of this phenomenon to be the cache effects caused by clock ticks. We then go on to gradually consider the factors influencing the D distribution as listed above (granularity, architecture, tick frequency and OS).

Instability. The resulting D distribution of running the micro benchmark on *M3* with a T_{grain} of 1ms are shown in Fig. 3 and are summarized in Table 2. These results are very disturbing: the OTHER average phase duration is 1060 μs whereas the minimum is only 668 μs, a slowdown factor of $\frac{1060}{668} \approx 1.6$ even for a uniprocessor. Recall that no other user processes run while measurements take place. FIFO scheduling (1.4 slowdown) insures that even system processes are not interrupting our benchmark. As this has never been reported (to the best of our knowledge), we hypothesize that this remarkable "head phenomenon" is overlooked by other noise studies which might ignore it by considering the frequent duration (1ms in this case) to be their base case. This can happen when estimating an application's phase duration using its average execution time on one node, and in so doing, implicitly including fine grained noise elements.

ID	Processor		Main Memory			Cache Size			Bus
	Type	Clk	Size	Clk	Type	L2	L1		Clk
							date	code	
		GHz	*MB*	*MHz*		*KB*	*KB*	*K*	*MHz*
M1	P-III	0.9	256	133	SDRAM	256	16	16	133
M2	P-IV	2.4	1024	266	DDR-SDRAM	512	8	12	533
M3	P-IV	2.8	512	400	DDR-SDRAM	512	8	12	800
M4	P-IV	3.0	1024	400	DDR2-SDRAM	1000	16	12	800

Table 1: *Different platforms on which the benchmark was run. The size of the L1 instruction cache is specified in KB for Pentium-III and in micro operations (Kuops) for Pentium-IV.*

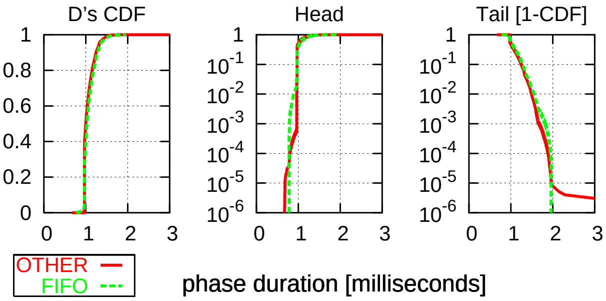

Figure 3: *Cumulative Distribution Function (CDF) of time actually taken to run a T_{grain}=1ms loop, under the default (OTHER) and realtime (FIFO) schedulers. The right plot focuses on the tail by showing the survival function. Head and tail are shown on a log Y scale.*

Scheduler	Avg.	Med.	Stddev	Min	Max
OTHER	1060	999	127	668	8334
FIFO	1080	1025	140	778	1959

Table 2: *Statistics of the D distribution shown in Fig. 3 [μs].*

Instability is much worse for clusters. Recall that the Y axis of Fig. 3 (right) can serve as an approximation of p. For example, there is a p=1:100 chance a phase duration would take longer than ~1.5ms. Repeating the rough approximation procedure detailed in Section 2.4, with the difference of changing the denominator in Eq. 5 to be 668μs (the minimum) yields a slowdown factor lower bound of ~2.

Sources of Noise. The difference between the FIFO and OTHER schedulers sheds some light on the source of the observed phase duration variability[3]. When running the benchmark as FIFO, we know for a fact it was not preempted in favor of other processes. We can therefore conclude that noise generated by system daemons is only responsible for the *difference* in variability exhibited by the two policies, that is, the horizontal "right turn" taken by the OTHER curve at $p = 10^{-5}$ (right of Fig. 3; intersecting the X axis beyond 8 ms, as indicated by Table 2). We conclude that the major guilty party of committing harmful frequent fine grained noise are system interrupts — executed in a non-process context. Instrumenting the kernel to log all inter-

[3] Both scheduling policies are mandated by POSIX [27, 12]: OTHER is the default time-sharing policy; FIFO is a First In First Out realtime policy. Different processes may have different policies. Realtime processes are *never* preempted in favor of non-realtime processes.

Interrupt / IRQ		Count	Direct Overhead	
Description	ID		Cycles	Percent
per node tick	0	1,082,744	23,437,759,192	0.775%
per CPU tick	239	1,082,731	1,674,717,580	0.055%
network	18	5,917	102,987,708	0.003%

Table 3: *The system interrupts that occurred within the M3/1000Hz/FIFO/1ms benchmark. The benchmark's total duration was eighteen minutes (3,023,642,905,020 cycles on a 2.8 GHz machine). Data instrumented by klogger.*

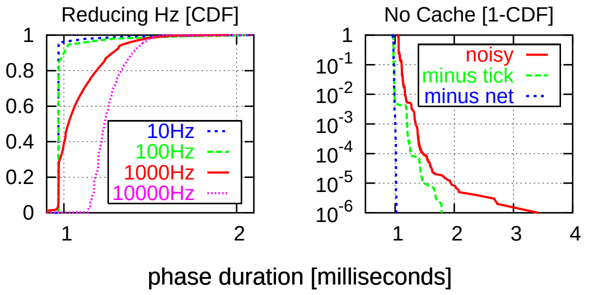

Figure 4: *Left figure shows the phase duration CDF of the M3/FIFO/1ms benchmark with various tick rates. Right figure shows the tail of the D distribution obtained by the M3/1000Hz/FIFO/1ms experiment with disabled caches (inner curves gradually subtract the direct overhead of interrupts).*

rupts (Table 3) revealed that the only activity present in the system while the FIFO measurements took place were about a million ticks and ∼6000 network interrupts, indicating ticks are probably the main cause of variability.

Cache Effects. An immediate experiment we have conducted to test our hypothesis regarding ticks, is to run the same benchmark with different tick rates (Hz). The results are shown in the left of Fig. 4 and indeed indicate that reducing Hz significantly diminishes variability. On the other hand, one is unable to ignore the variability that is still there. Attributing this to the direct overhead of the remaining interrupts (ticks and network) seems unrealistic. Further, note that the direct overhead of interrupts within the original 1000 Hz FIFO experiment amounts to a mere 0.83% (Table 3). This isn't remotely enough to explain the observed variability (Fig. 3) and the overall 39% overhead (FIFO average vs. minimum in Table 2). The conclusion is that variance is mostly the outcome of indirect overhead, that is, cache misses incurred when context switching back and forth between interrupts and the application. This was verified by repeating the 1000 Hz experiment with a disabled cache. Naturally, everything ran slower and results were more stable, but this has finally allowed us to provide a full explanation of variability: The "noisy" curve in Fig. 4 (right) shows the tail of the no-cache phase duration distribution. To its left, according to the information provided by klogger, the "minus tick" curve explicitly subtracts direct overhead caused by ticks from the phases in which they were fired. Finally, the "minus net" curve further subtracts direct overhead of network interrupts. The end result is a prefectly stright line indicating all phase durations are equal and all variability is accounted for.

To conclude, our findings indicate that the observed variability is a product of ticks, network interrupts, and the cache effects they cause. Supercomputers usually employ high-end communication networks (Myrinet, Quadrics) that provide dedicated processors for handling of network events. Such hardware has potential to largely eliminate most network interrupts as a source of variability, which would leave ticks as the major source of degraded performance.

We remark that the reason behind the shorter phases at the head of the distribution (Fig. 3) remains somewhat of a mystery. Indeed, Fig. 4 serves as strong evidence that this happens due to interactions between cache and OS interrupts. However, our benchmark uses very little memory: the (volatile) loop index, the measurements array (assigned at the end of each phase), and the loop instructions themselves: a working set of presumably very few cache-lines, that seems too small to justify such an effect. Nevertheless, the effect was consistently reproduceable on all P-IV machines we have checked.

Granularity. The p probability of per-node noise is tied to the granularity of the computation. Real world applications exploit fine grained parallelism ranging from several milliseconds (sometimes referred to as "medium grain") to less than a millisecond [15, 17, 26, 25, 16, 13]. Fig. 5 shows the tail of the phase duration distribution obtained by the platforms specified in Table 1, for various grain sizes. Durations are expressed as percentile addition relative to the shortest associated phase. Setting *M2* aside, one apparent observation is that finer grain sizes imply an increased p, as indicated by the Y-aligned curves at the right of Fig. 5 (short delays with low probability) gradually becoming orthogonal to the Y-axis when moving to the left (longer delays with higher probability).

This can be easily explained when considering the 1μs subfigure: Phases of 1μs are short enough so that most mange to escape the noise, that is, ticks occurring at a 1000 Hz frequency. However, every thousand phases (1 ms) one is bound to be interrupted by a tick and pay its full price ($> 1\mu$s). As a consequence, 1μs-curves become perpendicular to the Y-axis for $p \approx \frac{1}{1000}$.

A coarser granularity implies reduced variability due to a combination of two factors: the overhead penality gets smaller *relatively* to the grain size, while at the same time more phases suffer from similar noise. For example, at T_{grain} of 1 ms most phases suffer from one tick (while a small fraction mange to escape them or alternatively suffer from two). The fact that the situation is worse for T_{grain} of 1μs than of 1 ms, is somewhat different than the observation made by Petrini et al. [25] that the effect of noise is most pronounced when the application "enters in resonance with noise of a similar harmonic frequency and duration". If this was the case than the situation was reversed. Indeed, phases of 1μs are rarely interrupted by 1000 Hz ticks, but according to the "noise law" this is sufficient to slow the entire application down for a large enough n. In accordance with Petrini's observation, variability is further reduced when moving (to the right of Fig. 5) into a state where noise frequency is finer than T_{grain} and the relative overhead penalty becomes much smaller.

Hardware and Tick Rate. The data shown in Fig. 5 indicates that platforms have decisive effect on the amount of noise. Recall that according to Fig. 2, systems should as-

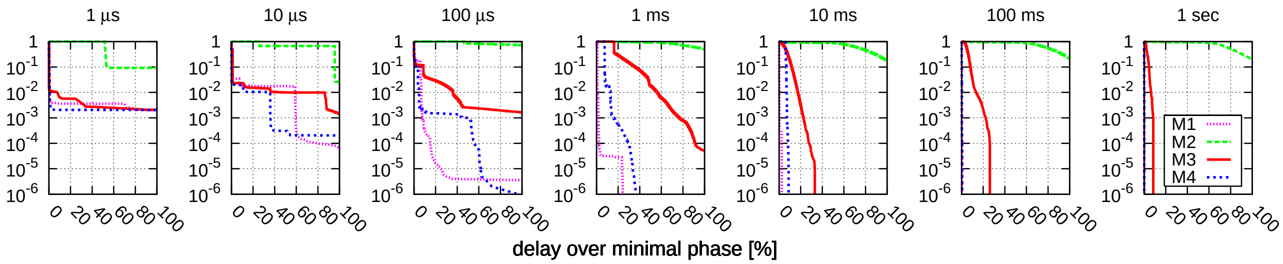

Figure 5: *The survival function of the D distribution associated with the four platforms, for grain sizes ranging from 1µs to 1sec (1000 Hz tick rate and FIFO priority). The X axis shows percentile delay with respect to the minimal phase duration obtained by the associated platform/grain combination. To allow for a meaningful display, the X-range is limited to double the minimum (100%).*

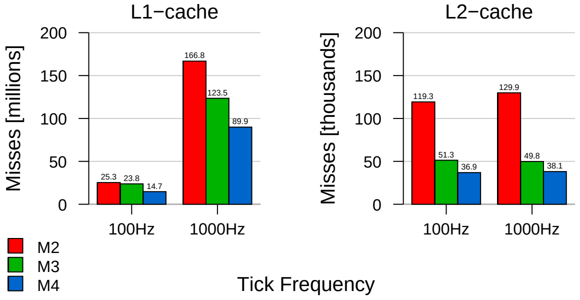

Figure 6: *Cache miss counts for the FIFO 1 ms benchmarks.*

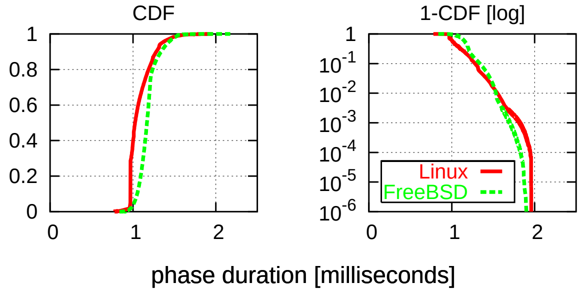

Figure 7: *Similar variability for different OSs.*

pire to reduce per-node noise below a threshold of $p = 10^{-5}$ (or better yet, $p = 10^{-6}$) so that the noise effect would be marginal for most practical cluster sizes. Unfortunately, this is not the case for any of the tested platforms, as all of them suffer from significant noise with higher probabilities for grain $\leq$ 1ms. Evidently, the P-III machine (*M1*) is usually the least vulnerable to ticks' cache effects, closely followed by the P-IV 3.0 GHz machine (*M4* — the fastest we have tested), while not that far behind we find the P-IV 2.8 GHz (*M3*) at third place. The results of the slowest P-IV (*M2*) are simply disastrous.

When examining Table 1 it is somewhat tempting to categorize the machines into a vulnerable (*M2/M3*) and invulnerable (*M1/M4*) classes based on the L1-data size, as this is the only quality in which vulnerable machines are consistently inferior to invulnerable ones. However, considering the fundamental differences between the P-III and P-IV architectures, such a division might be arbitrary and unjustified. On the other hand, the inherent similarities between the P-IV machines suggest that a better methodology would be to focus on the differences between these platforms. Our next step was therefore to instrument the P-IV kernel with Intel's Performance Monitoring Counters technology [4]. Using special registers, we were able to count the number of L1 and L2 cache misses that occurred while the P-IV benchmarks were running.

The results are shown in Fig. 6 and coincide with our findings so far: cache miss rates of both L1 and L2 perfectly correlate with the variability exhibited by the P-IV machines (Fig. 5), with *M2* suffering the largest number of misses and *M4* suffering the least. Note that increasing the tick frequency does not effect L2 much. The effect is much more pronounced for the L1 cache, with a miss rate growth of up to a factor of 6.6 (*M2*).

We can only speculate as to the reason why the different configurations yield such dissimilar results in terms of cache behavior. It is probably reasonable to assume that the fact *M4* enjoys a bigger L1-data cache (Table 1) immediately translates to improved cache hit rates. It is also reasonable to expect *M3* to be superior to *M2* in this respect, due to improved memory and bus speed. Regardless of the exact reason, the bottom line is that relatively minor differences in hardware configurations can have decisive effect on the variability of computation duration, even for an application as trivial as our benchmark, which only records the ending time of empty loops. We expect variability to significantly increase when replacing this artificial benchmark with real world applications, as these will stress the competition over memory between the application and periodic interrupts.

Operating System. Ticks are employed by all general purpose OSs. The goal of our final experiment was to make sure variability is not the artifact of Linux implementation specifics. For this purpose we have also bechmarked another OS — FreeBSD (5.3 stable). Fig. 7 shows the result of running the FIFO/1ms/1000Hz/M3 benchmark on the two OSs, indicating FreeBSD suffers from similar deficiencies.

4. ELIMINATING THE NOISE

We have shown that periodic OS ticks are a main source of noise. It has been suggested [22, 25, 16, 29] that noise effects can largely be reduced if noise activity is gang scheduled [8], that is, set up to simultaneously run across the parallel machine at the exact same time. While this approach does not avoid the per-node price of noise, it does make sure the price is paid simultaneously, thus preventing the harmful amplification effect as quantified by the "noise law".

Indeed, such an approach is especially suitable for coarser and longer noise generated by various system daemons [25, 16, 29]. These can actually be considered as sub-processes of a "system job" that alternately executes along with the user's job according to the gang scheduling rules (time sharing). However, in order to apply this approach on fine grained periodic noise as generated by ticks, the nodes composing the cluster must effectively have perfectly synchronized clocks. Establishing this might be technically problematic for large clusters, though some machines do in fact offer a global clock [22, 16, 29]. Nevertheless, this is not guaranteed and even if available, might entail a price of increased complexity, overhead, and asynchronous noise.

Regardless of the actual price, we argue that synchronizing ticks is analogous to treating the symptom rather than the disease. Instead of paying the price of synchronicity, why not eliminate ticks altogether? There is no real technical difficulty, and it actually turns out there are other non-HPC domains that will greatly benefit from ticks removal. This section briefly describes *why* this is indeed the case and *how* to accomplish the task. More details can be found in [31].

4.1 The Case Against Ticks: A Global View

We have already made the first non-HPC case for removing ticks when showing their indirect overhead at the desktop level can be significant for some platforms. In fact, all machines associated with the data presented in Fig. 5 are standard workstations at our department. None of us were happy to find out that e.g. *M3* operates 1.6 times slower than its optimum. The fact this optimum is obtained at a rare rate of 1:10,000 (Fig. 3) does not weaken this argument.

The second case for removing ticks concerns OS timing services, that are crucial for multimedia and other soft realtime tasks. Tick resolution (Hz) is important because systems measure time by it, including when timer alarms should go off. If Hz is too coarse, deadlines might be missed. For example, we have shown that when playing a standard MPEG 60 frames per seconds movie, a player might discard $\frac{2}{3}$ of its frames if played on a 100 Hz OS [7]. In this particular case, an increase to 1000 Hz was enough to solve the problem. But even finer timing services are required (ranging up to tens of thousands of events per seconds), e.g. for video rates of crash experiments [32] and rate-based networking [1]. Increasing Hz to accommodate this generates significant overheads [7].

The third case for removing ticks involves power consumption considerations crucial for mobile and autonomous systems. Ticks occur even if the system is otherwise idle, thus uselessly wasting power (20W according to [19]). We have shown that increased tick rate translates to significant power loss [31]. It seems remarkable such a huge effort is spent on throttling (slowing the hardware clock down when possible), while useless OS-clock activity is completely overlooked.

4.2 The Solution: Smart Timers

Evidently, ticks are problematic in domains transcending HPC noise. It seems their drawbacks are significant enough to constitute a true incentive for change, that may actually be sufficient to overcome conservative opposition (recall that this is a 30 years old design decision that affects all modern OSs). On the other hand, ticks seem like an inevitable compromise between contradicting system requirements of reduced overhead (mobile and HPC systems) vs. improved timing services (multimedia and soft realtime systems).

Indeed, all existing solutions to the problems raised above are *domain-specific* in that one domain's solution ignores or worsen the problem of other domains. An excellent example for this is handling HPC (tick) noise by synchronizing it. Another example are realtime systems which replace periodic ticks with *one-shot* timers that are set only for specific needs and fired exactly at their designated time [18, 28, 11]. These are unsuitable for general purpose OSs as any user can bring the system down by generating numerous events with nanosecond differences. Other timing mechanisms that can be utilized by a general purpose OS (*soft-timers* [1], *firm-timers* [14]) are yet again domain-specific because they build on and extend periodic ticks. Solutions that target power consumption silence ticks when the system is idle (but timing, noise, and non-idle power issues still remain).

We contend a unified solution to *all* the problems is in fact available and within reach. We call this *smart timers*, defined to combine **(1)** accurate timing with a settable bound on maximal latency, **(2)** reduced overhead by aggregating nearby events, and **(3)** reduced overhead by avoiding unnecessary periodic ticks. Perhaps the most intuitive implementation of smart timers would be to preserve alignment of all events on Hz boundaries (Hz is the "settable bound on maximal latency"; time alignment obtains event aggregation) while avoiding useless ticks by setting a one-shot timer to the next (aligned) event if such exists, along with doing all kernel accounting upon each kernel entry rather than periodically. If no timers exist (idle system or only one CPU bound process is running), no timer would be set: the OS will resume control only upon other device interrupts (e.g. keyboard, network) or system calls. See details in [31].

5. RELATED WORK

5.1 About Noise

The negative effect of noise on parallel systems has been known for more than a decade [22]. Our contribution includes suggesting a probabilistic model to explain this phenomenon, placing the focus on ticks' indirect overhead as a major source for noise endured by fine-grained jobs, noting ticks price might be significant even if they are synchronized, showing the implications of architecture and tick frequency on this price, and suggesting this should be dealt with by means of tick elimination rather than synchronization, as part of a global case against ticks (smart timers).

To the best of our knowledge, the first study recognizing the effect of noise on parallel machines was conducted in 1994 by Mraz [22], who noticed that the time it takes a token to circulate between 128 machines connected in a virtual ring is dramatically amplified due to large variance in peer to peer communication. The variance source was traced to be rare activity of system daemons and OS interrupts, namely, noise. In spite of its rarity, variance impact was nevertheless decisive as the token's chances to escape it in all nodes were slim (probably obeying the "noise law"). Various techniques were attempted in an effort to reduce variance which included raising the priority of the application relative to the system daemons (up to the point where these are starved), and synchronizing ticks across all nodes.

The next report on HPC noise was by Phillips et al. [26] and involved the NAMD application spanning up to 3,000 processors on the Pittsburgh Supercomputer Center's Lemieux (750 4-way nodes). This work was the first to note the inability to productively use the fourth processor, which nowadays seems to be the common wisdom.

A year later at SC'03, two noise related studies were published. The first, by Jones at el. [16], utilized several supercomputers, the largest was Lawrence Livermore National Laboratory's ASCI-White, composed of 512 16-way SMP nodes (8,192 processors). This paper serves as a comprehensive roadmap for practical noise reduction techniques, including gang scheduling of daemon noise (by raising the application's priority and periodically reducing it in a coordinated manner such that daemons are allowed to run) and, yet again, synchronizing ticks (while reducing their frequency). Unfortunately, the most effective technique was nevertheless refraining from using the sixteenth processor.

The second SC'03 paper by Petrini at el. [25], discovered "missing performance" on the (2,048 4-way) 8,192-processors Los Almos ASCI-Q, when comparing its observed performance to the performance predicted by a suitable model [17]. In accordance with the above, this study also recommends gang-scheduling of noise along with not using the fourth processor. This paper characterized noise as a collection of "harmonics" and claimed that different types of noise "resonate" with applications differently, depending on their granularity (a noise component is most harmful if its frequency and duration matches the granularity of the application). After characterizing the ASCI-Q noise components, the "kernel activity" was found to be responsible for up to 66% of the noise effecting fine-grained jobs. In a later paper [13], this activity was identified as mostly ticks and network interrupts, coinciding with our findings in Table 3.

Based on the findings in [25], Terry et al. [29] developed a Linux scheduler kernel patch for the Cray XD1 system, which is essentially a full fledged gang scheduler with a tick-long time slots. The kernel is made aware of jobs (of user or system) and makes sure these are co-scheduled. Obviously, this relies on nodes having fully synchronized clocks.

Finally, there is Fast OS[4] ("Forum to Address Scalable Technology for runtime and Operating Systems") which is backed by IBM, Intel, HP, SGI, Bell Labs, LLNL, LANL, Caltech, and more. This forum embodies ten projects aimed to provide adequate OS for large clusters, eight of which list noise as a problem (four with "high priority") [21].

5.2 About Ticks

Research regarding timing and ticks was mostly conducted in the realtime domain. Many studies have recognized and tried to solve the inherent problem of general-purpose OSs not being able to adequately support multimedia and other soft realtime applications [23, 1, 3, 2, 14, 7, 24]. The fundamental problem is the coarse granularity of Hz (OS clock tick frequency). As general-purpose OSs cannot use one-shot timers (popular within hard realtime systems), relevant research efforts were inclined to pay additional overhead penalty for the sake of improved accuracy, consequently making the OS less suitable for HPC environments.

Three papers specifically targeted ticks: Etsion et al. [7] checked the benefits and inherent limitations of *periodic timers* by increasing Hz (up to 20,000) and found this may be quite beneficial in terms of accuracy, but results in severe overhead penalty. Aron and Druschel [1] implemented *soft timers* keeping Hz unchanged, while opportunistically using each kernel entry (e.g. due to system call), when the context switch is already payed, to check whether expired timers exist. Goel et al. [14] defined *firm timers* to combine all timer mechanisms (periodic, one-shot, soft) to provide better timing services while making sure that the potential one-shot overhead is bounded. Since all these mechanisms incorporate periodic ticks, HPC systems do not benefit.

Finally, the "High Res POSIX timers" Linux project[5] includes five timing patches. None of these are "smart timers". Those that state they are tickless, "no HZ tick test" and "VST tick elimination", are pure one-shot or stop ticking only when the system is otherwise idle, respectively.

6. CONCLUSION AND FUTURE WORK

System noise is widely recognized as a major obstacle to applications' scalability [26, 16, 25, 21]. We develop a model that formulates the intensity of noise, suggesting a simple theoretical explanation for the well known empirical fact that noise effects increase with cluster size (n). Assume p is the per-node probability for a running task to be delayed by noise. If p is small enough (e.g. $p \leq \frac{1}{10n}$), our intuitive "noise law" states that noise is linearly proportional to n such that $p \times n$ is a good approximation of the parallel job's probability to be delayed upon each computation phase. This is in line with measurements that have been conducted on a number of systems showing linear performance degradation [16, 25].

Given a system, trivial (node) benchmarking of the application's phase-duration, yields a distribution from which p can be deduced, along with the approximated delay-duration caused by noise. Combining this with the "noise law", results in a quick-and-dirty lower bound of slowdown penalty. Indeed, our model is more valuable when seeking rough figures and a general understanding of the noise phenomenon, rather than the exact numbers. (However, we do outline the manner in which better estimates can be obtained.)

An important consequence of the noise model is its implication on how systems should be designed. We find that practical cluster sizes (up to thousands) require a p which is smaller than 10^{-5}, if noise effects are to be kept tolerable. A p value of 10^{-6} would essentially eliminate the effect. We evaluate p for a large number of grain sizes, architectures, and OSs, and consistently find it to be much larger than the appropriate thresholds if granularity is fine (few ms or less).

Some architectures exhibit (much too) large p values with significant delays, even for coarser grain sizes of tens to hundreds of milliseconds. In fact, p is shown to be surprisingly sensitive to seemingly minor changes in the hardware configuration, especially considering the application we used for evaluation is hardly vulnerable in terms of data locality. We expect this phenomenon to be amplified for real world applications which typically possess these vulnerabilities. System designers should be made aware that such modest changes might have decisive implication on applications' runtime.

The main guilty party of generating noise that is mostly harmful to fine grained applications is identified to be periodic OS clock interrupts (ticks), that almost exclusively constitute the fine grained noise activity within the system. While the direct overhead of ticks is relatively small (below 1% across all benchmarked architectures), their indirect overhead might reach tens of percents, possibly amplified by the "noise law" to hundreds of percents. In particular, we identify a strong correlation between increased variability in

[4] http://www.cs.unm.edu/~fastos

[5] http://sourceforge.net/projects/high-res-timers

phase-duration and L1/L2 cache miss rates. Additionally, we find L1 miss rates to be tightly correlated to tick frequency, in contrast to L2 misses which remain relatively unchanged, even when tick frequency is significantly increased.

Finally, a major contribution of this paper is challenging the common wisdom of handling ticks-noise by means of ticks synchronization [22, 16, 25, 29]. This approach can indeed alleviate the noise problem to some extent, but might be hard or impossible to implement on large clusters. Additionally, while overcoming the noise amplification effect (by making sure ticks occur simultaneously across all nodes), this approach still suffers from the potentially significant per-node indirect-overhead penalty incurred by ticks, which can sometimes reach tens of percents.

In addition to the above drawbacks, we argue that the most important deficiency of the ticks synchronization approach, is that it actually treats the symptom rather than the cause. The only way to fundamentally get rid of fine grain noise is by eliminating ticks altogether. A basic OS principle suggested by Finkel is that events at a high level are actually polling at a lower level [10]. For over 30 years OSs have used polling. It is now time to consider pushing the polling down to the hardware, leaving the OS to be event-based. We suggest "smart timers" as an alternative to ticks. These allow accurate timing with a settable bound on maximal latency and reduced overhead by aggregating nearby events and by avoiding unnecessary periodic ticks [31]. We show that these properties make the "smart timers" mechanism appealing to several non-HPC domains (desktops, multimedia, soft realtime, mobile and power-aware computing), thereby increasing its chances to be incorporated within general purpose operating systems. We are in the process of developing an appropriate patch to the Linux-2.6 kernel.

Acknowledgments

Many thanks are due to Danny Braniss for providing access to various platforms, and to Ofer Malcai and Or Hershkovitz for useful discussions regarding the "noise law".

7. REFERENCES

[1] M. Aron and P. Druschel, "*Soft timers: efficient microsecond software timer support for network processing*". *ACM Trans. Comput. Syst.* **18(3)**, pp. 197–228, Aug 2000.

[2] S. A. Banachowski and S. A. Brandt, "*The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes*". In *Multimedia Computing and Networking*, Jan 2002.

[3] S. Childs and D. Ingram, "*The Linux-SRT Integrated Multimedia Operating System: Bringing QoS to the Desktop*". In *Real-Time Technology & App. Symp.*, pp. 135, May 2001.

[4] Intel Corp., *IA-32 Intel Achitecture Software Developr's Manual. Vol. 3: System Programmin Guide.*

[5] J. J. Dongarra, H. W. Meuer, H. D. Simon, and E. Strohmaier, "*Top500 supercomputer sites*". URL http://www.top500.org/. (updated every 6 months).

[6] Y. Etsion and D. G. Feitelson, *Time Stamp Counters Library - Measurements with Nano Seconds Resolution.* Technical Report 2000-36, The Hebrew University, Aug 2000.

[7] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of clock resolution on the scheduling of interactive and soft real-time processes*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 172–183, Jun 2003.

[8] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

[9] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "*Parallel job scheduling — a status report*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), Springer Verlag, Jun 2004.

[10] R. A. Finkel, *An operating systems Vade Mecum.* Prentice-Hall, 2nd ed., 1988.

[11] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "*The pebble component-based operating system*". In *USENIX Technical Conf.*, Jun 1999.

[12] B. O. Gallmeister, *POSIX.4: Programming for the Real World.* O'Reilly & Associates Inc, Jan 1995.

[13] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "*Analysis of system overhead on parallel computers*". In *IEEE Intl. Symp. on Signal Processing and Information Technology*, Dec 2004.

[14] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "*Supporting time-sensitive applications on a commodity OS*". In *Symp. Operating Syst. Design & Implementation*, pp. 165–180, Dec 2002.

[15] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme, "*A general predictive performance model for wavefront algorithms on clusters of smps*". In *Intl. Conf. on Parallel Processing*, pp. 219, Aug 2000.

[16] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "*Improving scalability of parallel jobs by adding parallel awareness to the operating system*". In *Supercomputing*, pp. 10:1–20, Nov 2003.

[17] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "*Predictive performance and scalability modeling of a large-scale application*". In *Supercomputing*, pp. 37, Nov 2001.

[18] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "*The design and implementation of an operating system to support distributed multimedia applications*". *IEEE J. Select Areas in Commun.* **14(7)**, pp. 1280–1297, Sep 1996.

[19] T. Li and L. K. John, "*Run-time modeling and estimation of operating system power consumption*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 160–171, Jun 2003.

[20] J. Lions, *Lions' Commentary on UNIX 6th Edition, with Source Code.* Annabooks, 1996.

[21] A. Maccabe, "*FAST-OS: Forum to Address Scalable Technology for runtime and Operating Systems*". URL www.cs.unm.edu/fastos/05Status/FASTOS-Feb2005.pdf, Feb 2005.

[22] R. Mraz, "*Reducing the variance of point to point transfers in the IBM 9076 parallel computer*". In *Supercomputing*, pp. 620–629, Nov 1994.

[23] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "*SVR4 UNIX scheduler unacceptable for multimedia applications*". In *Network & Operating Syst. Support for Digital Audio & Video*, Nov 1993.

[24] J. Nieh and M. S. Lam, "*A SMART scheduler for multimedia applications*". *ACM Trans. Comput. Syst.* **21(2)**, pp. 117–163, May 2003.

[25] F. Petrini, D. J. Kerbyson, and S. Pakin, "*The case of missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q*". In *Supercomputing*, Nov 2003.

[26] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale, "*NAMD: biomolecular simulation on thousands of processors*". In *Supercomputing*, Nov 2002.

[27] *Linux Programmer's Manual: sched_setscheduler System Call.*

[28] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "*A firm real-time system implementation using commercial off-the-shelf hardware and free software*". In *IEEE Real-Time Technology & App. Symp.*, pp. 112–119, Jun 1998.

[29] P. Terry, A. Shan, and P. Huttunen, "*Improving application performance on HPC systems with process synchronization*". *Linux Journal* **2004(127)**, pp. 68–73, Nov 2004. URL http://portal.acm.org/citation.cfm?id=1029015.1029018.

[30] L. Torvalds, A. Cox, R. Love, and many others, "*HZ, preferably as small as possible*". URL http://seclists.org/lists/linux-kernel/2002/Jul/index.html#2588, Jul 2002. Thread from the Linux Kernal Mailing List.

[31] D. Tsafrir, Y. Etsion, and D. G. Feitelson, *General-Purpose Timing: The Failure of Periodic Timers.* Technical Report 2005-6, The Hebrew University, Feb 2005.

[32] D. Tyrell, K. Severson, A. B. Perlman, B. Brickle, and C. Vaningen-Dunn, "*Rail passenger equipment crashworthiness testing requirements and implementation*". In *Intl. Mechanical Engineering Congress & Exposition*, Nov 2000.