The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops)

Dan Tsafrir
IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

ABSTRACT

The overhead of a context switch is typically associated with multitasking, where several applications share a processor. But even if only one runnable application is present in the system and supposedly runs alone, it is still repeatedly preempted in favor of a different thread of execution, namely, the operating system that services periodic clock interrupts. We employ two complementing methodologies to measure the overhead incurred by such events and obtain contradictory results.

The first methodology systematically changes the interrupt frequency and measures by how much this prolongs the duration of a program that sorts an array. The overall overhead is found to be 0.5-1.5% at 1000 Hz, linearly proportional to the tick rate, and steadily declining as the speed of processors increases. If the kernel is configured such that each tick is slowed down by an access to an external time source, then the *direct* overhead dominates. Otherwise, the relative weight of the *indirect* portion is steadily growing with processors' speed, accounting for up to 85% of the total.

The second methodology repeatedly executes a simplistic loop (calibrated to take 1ms), measures the actual execution time, and analyzes the perturbations. Some loop implementations yield results similar to the above, but others indicate that the overhead is actually an order of magnitude bigger, or worse. The phenomenon was observed on IA32, IA64, and Power processors, the latter being part of the ASC Purple supercomputer. Importantly, the effect is greatly amplified for parallel jobs, where one late thread holds up all its peers, causing a slowdown that is dominated by the per-node latency (numerator) and the job granularity (denominator). We trace the bizarre effect to an unexplained interrupt/loop interaction; the question of whether this hardware misfeature is experienced by real applications remains open.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interfaces; C.4 [**Performance of systems**]: Modeling/measurement techniques; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; D.4.1 [**Operating Systems**]: Process Management—*Synchronization*

ExpCS, 13-14 June 2007, San Diego, CA

General Terms

Performance, Measurement

Keywords

Operating system noise, clock interrupts, ticks

1. INTRODUCTION

A *context switch* is defined to be the act of suspending one execution thread in favor of another. This term is commonly associated with the notion of *multitasking*, by which operating systems provide the illusion of being able to simultaneously run a number of applications that exceeds the number of available processors. The reason for this association is the fact that multitasking is realized by means of systematic context switching between competing applications and *time slicing* the processor. And so, the phrase "context switching" is usually implicitly suffixed by the phrase "between applications", which is assumed without being said.

However, as implied by its definition above, "context switch" is in fact a broader term. It applies to *any* two threads of execution that share the same processor, even if one is not associated with an application per-se. Specifically, this is the case when a hardware interrupt fires: the operating system temporarily suspends the currently running application, switches to kernel space, and invokes the interrupt handler (during which the kernel is said to be running in "interrupt context"). When the handler finally terminates, the operating system returns to user space and restores user context.

The interleaving of interrupts and applications directly coincides with how context switching is defined. Yet the associated overhead is usually overlooked, or at least not considered in isolation. This may have to do with the perception that hardware interrupts often yield a "traditional" application-to-application switch. For example, when a user interacts with a text editor by pressing a keyboard key, the typical reaction of a general-purpose OS (to the consequent hardware interrupt) would be to immediately suspend whichever process that happens to be running, dispatch the editor, and deliver to it the associated event (illustrated in Figure 1). This policy is part of the dominant doctrine to favor "I/O bound" applications that are consistently blocked waiting for some event to occur. As hardware interrupts signify the occurrence of such events, they therefore often yield a regular application-to-application context switch (from the currently running- to the waiting application) and so the overhead of the associated handler is typically bundled with the overall overhead of this switch.

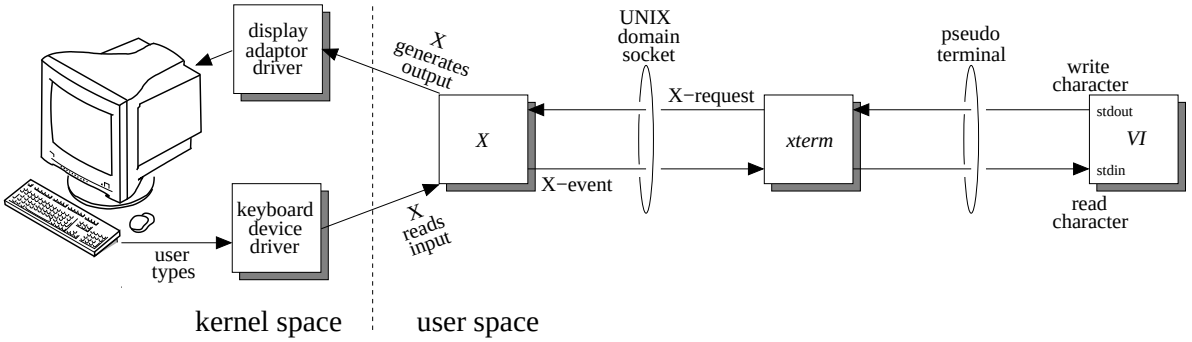

Figure 1: *The chain of events triggered by a user keystroke on typical a UNIX system. The key press generates a hardware interrupt that invokes an appropriate handler-routine and consequently propagates through a chain of connected blocked-waiting processes, each of which is awakened as a result of a matching context switch. This sequence usually takes place immediately after the keystroke, because all the processes involved are "I/O bound" and therefore possess a relatively high priority that allows them to preempt any currently running "CPU bound" application. The end result is that the context switch to and from the interrupt-handler is often temporally coupled with an application-to-application switch.*

The assumption underlying this approach of aggregating the overhead penalty is that the operating system is *reactive* in nature and just passively waits for events initiated by various external entities to occur. However, this is not the case. In fact, the OS embodies a distinct *proactive* component. This is manifested in the form of repeated periodic clock interrupts that are used by the OS to maintain control and measure the passage of time. The practice of utilizing clock interrupts for this purpose has started at the dawn of OS research in the 1960s [1] and has continued ever since, such that nowadays it is used by most contemporary OSs, including Linux, the BSD family, Solaris, AIX, HPUX, IRIX, and the Windows family. Roughly speaking, the way this mechanism works is that at boot-time, the kernel sets a hardware clock to generate periodic interrupts at fixed intervals (every few milliseconds; anywhere between 1/2ms to 15ms, depending on the OS). The time instance at which the interrupt fires is called a *tick*, and the elapsed time between two consecutive ticks is called a *tick duration*. The interrupt invokes a kernel routine, called the *tick handler* that is responsible for various important OS activities including (**1**) delivering timing services and alarm signals, (**2**) accounting for CPU usage, and (**3**) initiating involuntary preemption for the sake of multitasking.

Note that ticks always occur, regardless of the number of runnable processes that are present in the system (or even if it is idle). Importantly, the tick-rate is strictly time-driven and is independent of whether the respective handler triggers an application-to-application context switch or not. Consider the current stable FreeBSD release (6.2), which employs a 1000 Hz tick-rate and a 100 millisecond quantum. Assuming there is only one runnable process in the system, one might presume it will be allowed to continuously run, uninterrupted. However, in reality, it will endure 1000 context switch events per second during which it will be repeatedly preempted while the tick-handler takes charge (only to be resumed when the handler terminates). Likewise, if the system contains several runnable processes, it will not allow them to run uninterrupted throughout their quantum; instead each of them would be preempted/resumed 99 times before their quantum expires and an application-to-application context switch finally occurs.

We thus contend that the context switch overhead inflicted by interrupts deserves a focused attention and merits a separate evaluation that specifically addressees the issue. This paper attempts to supply such an evaluation: it builds on knowledge and results we gathered during the last 5 years under different titles [2, 3, 4, 5, 6, 7], aggregating and extending those parts that are most relevant to the topic at hand, as noted in each section.

We begin by noting that the common wisdom amongst system practitioners (partially reinforced by sporadic measurements [8, 9, 10]) states that the overall overhead incurred by ticks amounts to less than 1% of the available CPU cycles. This statement might be overly simplistic as it ignores

1. the platform being used,
2. the workload that is running,
3. the tick frequency, and
4. whether the specified overhead only relates to the *direct* component (time to execute the handler) or also includes the *indirect* penalty (degradation in performance due to cache state change).

We consider all of the above (along with a *fifth* point, which is the effect the associated overhead has on parallel jobs). For this purpose we evaluate the direct and indirect components in the face of different tick rates and workloads, across multiple generations of Intel processors spanning over a decade.

Section 2 deals with the first workload we use, which is a simple program that sorts an integer array. We

processor	trap		tick handler			
			TSC (default)		PIT	
	cycles	μs	cycles	μs	cycles	μs
P-90	153±024	1.70	814±180	9.02	498±466	5.53
PPro-200	379±075	1.91	1654±553	8.31	462±762	2.32
PII-350	343±068	0.98	2342±303	6.71	306±311	0.88
PIII-664	348±163	0.52	3972±462	5.98	327±487	0.49
PIII-1.133	364±278	0.32	6377±602	5.64	426±914	0.38
PIV-2.2	1712±032	0.72	14603±436	6.11	445±550	0.19

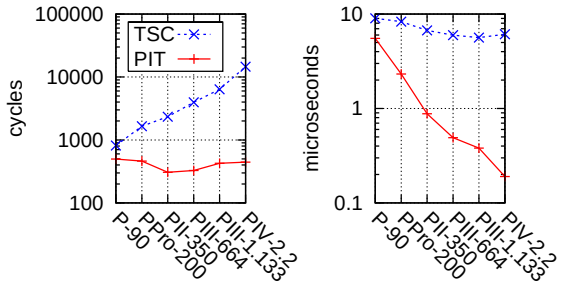

Figure 2: *The left table specifies the direct overhead component of tick context-switching, broken down to trapping to/from the kernel and the processing of the interrupt (average±standard deviation). In TSC mode, the slow access to the 8253 for the sake of a faster gettimeofday dominates the handler's runtime and makes it largely independent of the CPU speed; the PIT mode eliminates this access and makes the handler scale.*

measure the direct overhead by instrumenting the kernel, and the overall overhead by systematically changing the tick rate and observing the degradation in sorting performance. We then deduce the indirect component by analyzing the difference between the two, while identifying the trends emerging from varying the processor. In all cases, though, the widely used 1000Hz tick-rate yields around 1% of (overall) overhead, in accordance to the the common wisdom.

Section 3 deals with the second workload we use, which consists of repeatedly executing a simplistic loop that was calibrated to take one millisecond. Such loops are termed "do nothing" loops.[1] This workload attempts to model one thread within a *bulk synchronous* parallel job. In this setting, each thread is dedicated a processor and repeatedly engages in a compute-phase, after which it synchronizes with its peers. This means that no thread is allowed to continue to the next compute-phase until all its peers have completed the previous (every phase is followed by a barrier to insure safe access to shared data structure, or by a communication period to exchange data that is needed for the next computation). Thus, if one thread is late, all its peers must stand idle until it catches up, which dramatically amplifies the impact of any local disturbances that prevent the thread from completing its work on time. In general, this phenomenon is termed "OS noise" and is currently the focus of extensive research efforts in the supercomputing community, which attempt to both assess and reduce the noise [11, 12, 13, 14, 15]. We show that a delay of D seconds would cause a job with a compute-phase duration of G seconds to experience a slowdown of $1 + \frac{D}{G}$ if it spans enough nodes. Importantly, if D is caused by a tick, then the resulting global slowdown can only be considered as part of the indirect overhead inflicted by interrupt/application context switching.

Analyzing the perturbations in the do-nothing loops results in the contradictory conclusion that the overhead inflicted by ticks is in fact an order of magnitude bigger than what was reported above and sometimes much worse. This pathology was observed on Intel's Pentium and Itanium processors, and on IBM Power processors that are part of the ASC Purple [16, 17]. Through detail cache analysis we identify the cause of the effect as an unexplained interaction between ticks and do-nothing loops. Then, in Section 4, we address the contradictory results and describe our attempts to resolve this "overhead dispute". Finally, in Section 5, we conclude.

[1]Somewhat of a misnomer: a do-nothing loop can have an empty body, but it can also do some work.

2. IMPACT ON A SERIAL PROGRAM

In this section we explore the direct and indirect impact of ticks on a real application (that actually computes something useful). The two overhead components are individually addressed by the following two subsections, respectively. To generalize the results and to obtain a broader perspective, all experiments are conducted on a range of Intel platforms: (**1**) Pentium 90 MHz, (**2**) Pentium-Pro 200 MHz, (**3**) Pentium-II 350 MHz, (**4**) Pentium-III 664 MHz, (**5**) Pentium-III 1.133 GHz, and (**6**) Pentium-IV 2.2 GHz. The operating system is a 2.4.8 Linux kernel (RedHat 7.0): the same kernel was compiled for all architectures, which may have resulted in minor differences in the generated code due to architecture-specific ifdefs. The data which serves as the basis of this section was collected during October 2002 in preparation towards [2], but a significant portion of the analysis presented here is new.

2.1 Direct Overhead

Trapping. The direct component in the context switch overhead incurred by ticks can be divided into two measurable subcomponents. The first is the *trap* time, namely, the overhead of switching from user- to kernel-space and back. The second is the duration of the tick-handler, namely, the kernel routine that is invoked upon each tick and is in charge of performing all the required periodic activities. We estimated the first component by repeatedly invoking the getpid system call, which is reasonably suitable for this purpose because it is hardly doing any work other than trapping (we made sure that the pid was not cached in user space and that the traps indeed took place). The results are shown in the left of Figure 2 (second column). They do not coincide with Ousterhout's claim that "operating systems do not become faster as fast as hardware" [18], because trapping

takes roughly the same number of cycles regardless of the CPU clock speed (except on the P-IV 2.2GHz which was an early unsuccessful model that often fell behind its P-III predecessor [19]).

KLogger. The harder part was measuring the duration of the handler. For this purpose we have instrumented the kernel by using klogger, a kernel logger we developed that supports efficient fine-grain events [20]. 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 4 MB buffer in memory, and only exported at large intervals when space is about to run out. 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 (by which made sure events do not get lost) and timestamp with cycle resolution, followed by optional event-specific data. In our use, we logged all scheduling-related events, two of which mark the entry/exit points of the tick-handler and so by computing the difference we obtain the handler's execution time.

The Handler. In contrast to the trap duration, we find that the overhead for processing the clock interrupt is dropping at a much slower rate than expected according to the CPU clock rate — in fact, it is relatively stable in terms of absolute time. This turned out to be related to an optimization in the implementation of gettimeofday(): Linux keeps track of time using the standard 8253 PIT (programmable interrupt timer) chip, so whenever the kernel needs the wall-clock time it can simply access the 8253 through the I/O bus and read the data. However, as this is a relatively expensive operation, a possible alternative is to do it upon each tick instead, such that a gettimeofday() call would only have to interpolate the current time from the last value read, by using the time-stamp cycle counter (TSC). Since the latter is a much faster operation, this mode of operation limits the overhead incurred by the PIT to the number of timer interrupts per second. The two modes, common to both the 2.4.x and the 2.6.x kernel series, are somewhat confusingly called the PIT mode (each gettimeofday invocation accesses the 8253) and the TSC mode (only ticks do the access), such that the latter is the default. The respective measurements are specified and depicted in Figure 2.

2.2 Indirect Overhead

The Benchmark. The indirect overhead of clock interrupt processing can only be assessed by measuring the *total* overhead in the context of a specific application (as was done, for example, in [21]) and then by subtracting the direct component. The application we used is sorting of an integer array, somewhat similarly to the LMbench benchmark that sums a large array to assess cache effects of context switching [22, 23]. The sorted array occupies half of the L2 cache (the L2 cache was 256 KB on all platforms but the P-II 350 which had an L2 cache of 512 KB). The sorting algorithm was introsort, as implemented by the STL version that ships with gcc. The sorting was done repeatedly, where each iteration first initializes the array randomly and then sorts it (but the same random sequences were used to compare the different platforms). By measuring the time per iteration under different conditions, we can factor out the added total overhead due to additional clock interrupts (as is shown below). To also check the overhead inflicted by ticks along side regular application-to-application context switching, when cache contention is increasingly growing, we used different multiprogramming levels, running 1, 2, 4, or 8 copies of the test application at the same time; we used the combined throughput of the multitasked processes to determine the time per iteration. All this was repeated for the different CPU generations running kernel versions compiled with different tick rates of 100Hz, 1000Hz, 5,000Hz, 10,000Hz, and 20,000Hz, amounting to a total of 120 runs (= 6 processors times 4 multiprogramming levels times 5 tick rates).

processor	proc	time to sort one array [milliseconds]				
	num.	100Hz	1KHz	5KHz	10KHz	20KHz
P-90	1	120.10	117.48	127.73	136.03	156.56
	2	120.65	117.86	126.30	136.54	157.07
	4	120.03	119.98	126.70	138.57	161.10
	8	121.05	122.13	129.21	140.03	162.97
PPro-200	1	44.05	44.54	46.29	49.10	54.56
	2	44.26	44.66	46.41	49.31	54.68
	4	44.96	45.17	46.98	49.88	55.55
	8	45.26	45.41	47.28	50.13	55.93
PII-350	1	50.90	51.33	52.68	54.64	58.22
	2	51.14	51.48	52.84	55.01	58.37
	4	51.58	52.08	53.55	55.56	59.17
	8	51.92	52.40	53.81	55.75	59.53
PIII-664	1	12.67	12.72	12.97	13.40	14.05
	2	12.70	12.75	13.00	13.43	14.08
	4	12.77	12.82	13.09	13.52	14.18
	8	12.80	12.85	13.12	13.56	14.23
PIII-1.133	1	7.40	7.43	7.55	7.77	8.06
	2	7.41	7.44	7.57	7.77	8.06
	4	7.43	7.46	7.58	7.79	8.09
	8	7.45	7.47	7.58	7.80	8.11
PIV-2.2	1	10.93	11.01	11.25	11.62	12.26
	2	10.90	10.98	11.21	11.58	12.16
	4	10.89	10.98	11.22	11.57	12.15
	8	10.90	10.97	11.22	11.58	12.16

Table 1: *The time it takes to sort an integer array that occupies half of the L2 cache as a function of (1) the processor, (2) the tick frequency, and (3) the number of simultaneously sorting processes. Kernels were compiled with the gettimeofday/TSC optimization turned off.*

Raw Results and Anomalies. The results of all the runs are shown in Table 1 and typically make sense when examining the individual machines, namely, moving left the right (increasing the tick rate) or top to bottom (increasing the multiprogramming level) results in a longer sorting time due to the added (application-to-interrupt or application-to-application) context switch overhead. However, this is not always the case, especially in relation to the P-90 machine. We failed to explain the horizontal anomaly. But the vertical anomaly

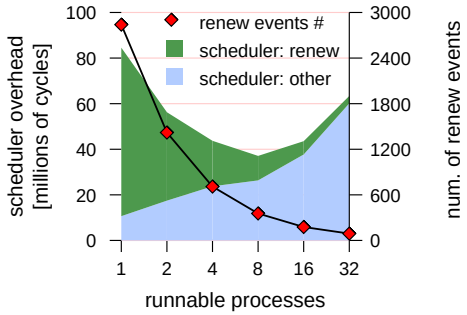

Figure 3: *Overall scheduling overhead as a function of runnable processes is U shaped, a result of two contrasting effects: less renew events vs. a longer run-list.*

is due to the scheduling algorithm employed by Linux 2.4, which heavily depends on the notion of *epochs*. An old epoch ends and a new one begins when all *runnable* (not sleeping) processes in the system exhaust their allocated quantum. The scheduler identifies this when attempting to choose the next process to run by iterating through the runnable processes list, only to find no process is eligible because all quanta are worn out. The scheduler then proclaims a new epoch has started and cycles through all the processes (runnable and sleeping alike) to renew the quanta. This linear renew-loop can be time consuming, as there are dozens of sleeping daemons that are resident in the system by default. Importantly, with a multiprogramming level of one, the loop is executed more frequently — whenever the sorting-process finishes its quantum (every 50ms) and therefore a new epoch is started (recall that this is the only runnable process in the system). Increasing the multiprogramming level reduces the renew frequency and hence the overall overhead. Figure 3 plots the overhead of the scheduling routine as a function of the number of competing sorting processes and illustrates the effect, which is gradually overshadowed by the fact that the run-list traversal (done upon each scheduling decision) becomes longer with each additional runnable process.

Deducing the Indirect Component. The measurements that are specified in Table 1 allow for an assessment of the relative costs of direct and indirect overhead. For example, when switching from 100 Hz to 10000 Hz, the extra time can be attributed to 9900 additional clock interrupts each second. By subtracting the cost of 9900 calls to the interrupt processing routine (from Figure 2), we can find how much of this extra time should be attributed to indirect overhead, that is mainly to cache effects. For example, consider the case of a P-III 664 MHz machine running a single sorting process. The average time to sort an array once is 12.675 ms on the 100 Hz system, and 13.397 ms on the 10,000 Hz system. During this time the 10,000 Hz system suffered an additional $9900 \times 0.013397 = 133$ interrupts. According to Figure 2 the overhead for each one (without accessing the 8253 chip) is 0.49 μs for the handler plus 0.52 μs for the trap, amounting to 1.01 μs. Hence, the total additional *direct* overhead was $133 \times 1.01 = 134\mu$s. But the difference in the time to sort an array is $13397 - 12675 = 722\mu$s! Thus $722 - 134 = 588\mu$s are unaccounted for and should be attributed to other effects. In other words, $588/722 = 81\%$ of the overhead is *indirect*, and only 19% is direct.

Consistency and Applicability. In the above paragraph, the decision to use a multiprogramming level of one, along with a Hz value of 100 vs. 10000, was an arbitrary one. The fact of the matter is that we could have used another multiprogramming level and Hz values when figuring out the indirect component in the overhead penalty inflicted by ticks. The question is whether we would have gotten the same result had we used a different set of parameters? (Say, a multiprogramming level of 4 and a tick frequency of 5000 Hz vs. 20000 Hz.) For our reasoning to be sound, and for our deduction methodology to have general applicability, the answer should be affirmative. In other words, we should make sure that any choice of the above three parameters yields the same result for a given processor, which would mean that our model is able to foresee the impact of changing the tick rate on the system performance. Otherwise, it only reflects sporadic measurements with no boarder applicability. Indeed, the linear approach of attributing a constant overhead penalty per-tick might simply be inadequate; in this case, our modeling effort would loose its predictive merit: it would be unable to infer the overhead of a hypothetical system (with a different Hz or multiprogramming level) based on the present observations.

Luckily, this is not the case. We have computed the indirect overhead component, as specified above, for *each* multiprogramming levels and *each* possible Hz pair. The results are shown in Figure 4, e.g. the 81% of indirect overhead associated with the PIII-664MHz / one-process / 100Hz / 10000Hz example given above is positioned within the fourth subfigure and aligned with the "100Hz : 10,000Hz" X-stub as marked by the arrow (the cross associated with one process is hardly visible because it occupies the same space as the other multiprogramming levels). Note that every processor is associated with 40 results: 4 possible multiprogramming levels times $\binom{5}{2}$=10 possible Hz pairs. Evidently, the methodology successfully passes this sanity check, with results being largely invariant to the specific manner by which they were obtained on a per-processor basis. The exception is the two slower machines, when lower tick frequencies and smaller multiprogramming levels are involved, which corresponds to the data anomaly discussed above. Overall, the relative weight of the indirect component appears to be growing as processors get faster, possibly due to the increasing gap between the CPU and memory clocks.

Putting It All Together. Table 2 summarizes our findings in terms of percentage of CPU cycles lost due to a thousand ticks: for each processor, the indirect component is the average over the 40 alternatives by which it can be computed (as explained above), whereas the direct component simply transforms the tabulated data

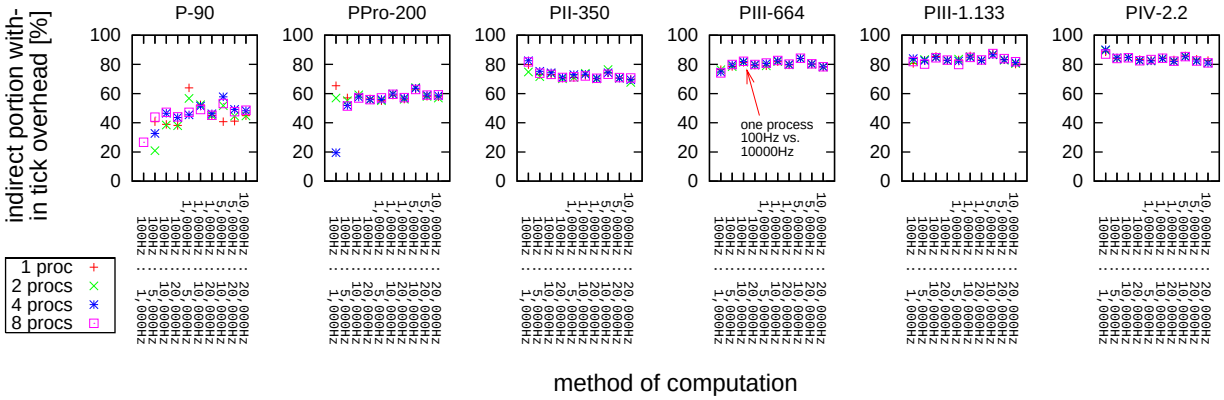

Figure 4: *The indirect portion of the overall context-switch overhead incurred by ticks is largely invariant to the configuration from which this data is derived, indicating our methodology is sound. See text for details.*

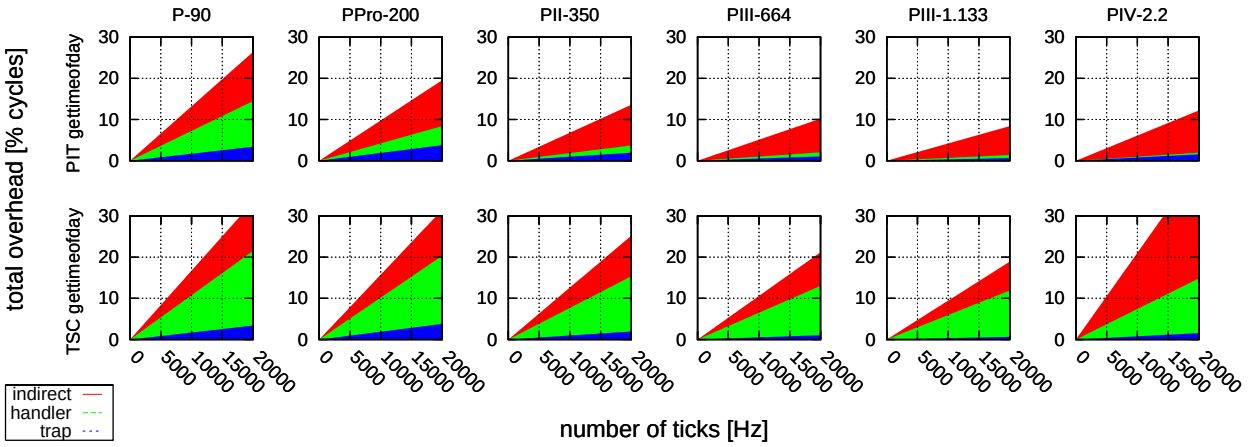

Figure 5: *Visualizing the contribution of each component to the overall overhead, based on Table 2. This highlights the downside of using the TSC-optimized* gettimeofday *that causes the handler to replace the indirect component as the dominant slowdown factor.*

processor	overhead per 1000 ticks [%]							
	PIT gettimeofday				TSC gettimeofday			
	trap	han-dler	indi-rect	**sum**	trap	han-dler	indi-rect	**sum**
P-90	0.17	0.55	0.60	**1.32**	0.17	0.90	0.60	**1.67**
PPro-200	0.19	0.23	0.56	**0.98**	0.19	0.83	0.56	**1.57**
PII-350	0.10	0.09	0.49	**0.68**	0.10	0.67	0.49	**1.26**
PIII-664	0.05	0.05	0.41	**0.51**	0.05	0.60	0.41	**1.06**
PIII-1.133	0.03	0.04	0.35	**0.42**	0.03	0.56	0.35	**0.95**
PIV-2.2	0.08	0.02	0.51	**0.61**	0.08	0.66	0.51	**1.25**

Table 2: *Percentage overhead of a thousand ticks.*

from Figure 2 to percents (by dividing the respective overhead cycles with the processor's speed). The overall overhead of a thousand ticks turns out to be 1-1.7% when the faster TSC version of gettimeofday is employed Out of this, only 35-40% is attributed to indirect effects, a result of the slow 8253 access being factored in as part of the direct portion. Note that the general trend is a declining overall overhead, but that this trend is overturned for P-IV. The same observation holds for the slower PIT version of gettimeofday, though this configuration yields a smaller overall overhead of only 0.4-1.3%, with the relative indirect component steadily growing to account for 85% of this (see also Figure 4).

As our model is linear, we can express the impact of ticks on a system that employs a frequency of n clock interrupts per second using the function

$$f(n) = n \times (P_{trap} + P_{handler} + P_{indirect})$$

where the coefficients are taken from Table 2. This function is visualized in Figure 5, which breaks down the contribution of each coefficient to the overall overheads. Similarly, by using Tables 1-2, we can calculate the time to sort an array on a hypothetical tickless system. For example, sorting on a P-III 664GHz at 10000 Hz with one running process takes 12.675 ms (Table 1). However, we know that $0.51\% \times 10 = 5.1\%$ of this time was wasted on ticks (Table 2), so a tickless system would require only $12.675 \times (1 - 0.051) = 12.028$ ms to complete the task. By applying this reasoning to the other tick frequencies associated with the same configuration (a multiprogramming level of one within P-III 664GHz) and averaging the results, we obtain a more represen-

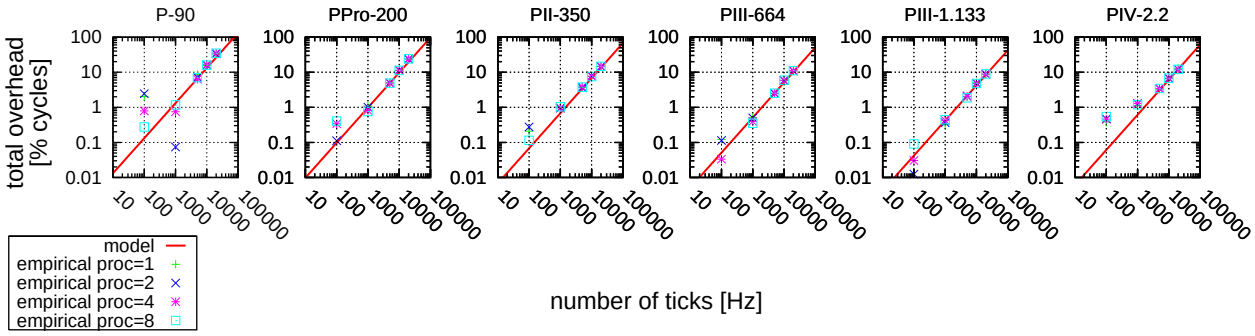

Figure 6: *Plotting the predictions of the model against the added overhead of Table 1's empirical observations relative to a hypothetical tickless base case, reveals a reasonable match. The practical limit on tick frequency lies somewhere between 10,000 Hz and 100,000 Hz. (Data associated with the PIT* gettimeofday *version.)*

tative value, denoted B_{p664}^{proc1}. This value allows us to put the model to the test, as we can compare the added overhead of each frequency as predicted by $f(n)$ to the added overhead as observed by our measurements. In our initial example, we would be comparing $f(10000)$ to

$$100 \times \frac{12.675 - B_{p664}^{proc1}}{B_{p664}^{proc1}}$$

A consistent small difference for *each* of the associated five n values would indicate the model is successful. Figure 6 does this comparison for all measurements in Table 1, and indicates that this is indeed largely the case.

3. IMPACT ON A PARALLEL JOB

The previous section was only concerned with how clock interrupts interfere with the run of a serial program, and with the extent to which they slow it down. But ticks have potentially a far worse effect on a bulk-synchronous parallel program, where one late thread deems all its peer idle until it catches up. This dynamic, which is illustrated in Figure 7, is commonly referred to as operating system "noise", and is the focus of intensive research efforts within the supercomputing community [11, 12, 13, 14, 15]. The reason being that Supercomputers composed of hundreds, thousands, or tens of thousands of processors are increasingly susceptible to this pathology. A single long tick out of 10,000 clock interrupts will presumably have negligible impact on a serial program; but with a fine grained job that spans 10,000 processors, experiencing this long event on each compute-phase becomes almost a certainty. (This is true for any noise event, not just ticks.) Thus, bulk-synchronous jobs have the property of dramatically amplifying an otherwise negligible slowdown incurred by sporadic OS activity. Importantly, if the delay is due to the fact the OS is handling some interrupt, *then the resulting slowdown can only be attributed to the indirect overhead of the associated application/interrupt context switch.*

We begin by suggesting a simple probabilistic model to quantify the effect of noise. Incidentally, similarly to the model that was developed above in the context of a serial program, this too has a linear nature (Section 3.1). But the new model is not enough to assess the

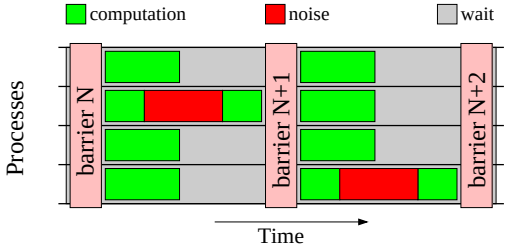

Figure 7: *Due to synchronization, each computation phase is prolonged to the duration of the slowest thread.*

slowdown, as it only addresses the probability for a delay, not the duration. We therefore conduct some additional analysis that highlights the granularity-to-delay ratio as the dominant factor in the resulting slowdown (Section 3.2). Consequently, we investigate the delays a fine-grained thread might experience by measuring the perturbation in the completion times of a short fixed do-nothing loop (emulating a compute phase). The results are overwhelming and contradictory to the previous section, indicating the effect of ticks on a *serial* program (let alone a parallel one) can actually be an order of magnitude worse than was suggested earlier in this paper (Section 3.3). The experiments which serve as the basis of this section were conducted during February 2005 in preparation towards [5], on which Sections 3.1 and 3.3 are largely based. Section 3.2 is new.

3.1 Modeling the Effect of Noise

In this subsection we are interested in quantifying the effect of OS noise in general (not just the one incurred by ticks). As additional nodes imply a noisier system, one can expect the delay probability of the entire job to increase along with the number of the nodes that it utilizes. Petrini et al. [13] assessed the effect of noise through detailed simulation of its (per-site) components. In contrast, here we analytically show that if the single-node probability for delay is small enough, the effect of increasing the node-count is linear: it simply multiplies the single-node probability.

Let n be the size of the job (number of nodes it is al-

located). Let p be the per-node probability that some process running on it is delayed due to noise, within the current compute phase. Assuming independence and uniformity (namely, delay events on different nodes are independent of each other and have equal probability, as is the case for ticks), then the probability that no process is delayed on any nodes is $(1-p)^n$. Therefore,

$$d_p(n) = 1 - (1-p)^n \quad (1)$$

denotes the probability that the job *is* delayed within the current computation phase. Based on this equation, we claim that if p is small enough, then

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

constitutes a reasonable approximation. To see why, consider the difference Δ between the two functions

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

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

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

where the rightmost summation has a negative value.[2] Therefore, by combining Equations 3 and 4, we get that

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

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

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

Thus, $d_p(n) \approx pn$ may serve as an intuitive approximated linear "noise law" for small enough p. This approximation coincides with empirical observations from real systems, notably the ASCI White and Blue-Oak [12], and the ASCI Q [13]. Measurements from the latter are shown in Figure 8 and demonstrate the effectiveness of the linear approximation. But if p is too big to meet the above constraint then one can use the more accurate

$$d_p(n) \approx 1 - e^{-np}$$

approximation, which applies to a wider p value range.

[2]Dividing the absolute value of the k and $k+1$ elements in the summation yields $\frac{1}{p} \times \frac{k+1}{n-k}$. Assuming $p < \frac{1}{n}$ (otherwise $d_p(n) \approx 1$), the left term is bigger than n. The right term is bigger than $\frac{k+1}{n}$, which is bigger than $\frac{1}{n}$. Hence, the quotient is bigger than 1, indicating the k-th element is bigger than its successor. Consequently, the summation can be divided to consecutive pairs in which odd k elements are negative, and are bigger (absolute value) than their even $k+1$ positive counterparts.

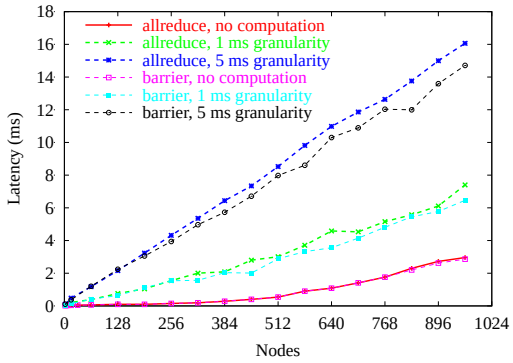

Figure 8: *The average delay experienced by a job upon each compute-phase is linearly proportional to the number of (quad SMP) nodes being used. The compute-phase is either 0, 1, or 5ms long. Synchronization between threads is achieved by either a barrier or an all-reduce operations. (Measured on LANL's ASCI Q; published in [13]; reprinted with permission of authors.)*

3.2 The Grain-Delay Ratio

Knowing the $d_p(n)$ probability for a job to get hit by a noise event is a first step; but in order to complete the picture and realize the impact this would have in terms of slowdown experienced by the job, we must also factor in G (the granularity of the jobs, namely, the duration of the compute phase in seconds) and D (the delay, in seconds, incurred by the noise event on the corresponding thread). This is true for two reasons. The first is that G affects p: if the system employs a tick rate of HZ interrupts per second, then $p = G \times HZ$ (e.g. if G = 1ms and $HZ = 100$ then $p = \frac{1}{10}$; and if $HZ = 1000$ then $p = 1$). The second reason is that that ratio between D and G defines the slowdown, which is

$$\frac{d_p(n)\cdot(G+D) + (1-d_p(n))\cdot G}{G} = 1 + d_p(n)\cdot\frac{D}{G}$$

Thus, in essence, the D/G ratio dominates the effect of noise: the smaller the granularity, the more significant the slowdown, which can be arbitrarily big if G goes to zero; a bigger delay prompts a similar effect, with an arbitrarily big slowdown incurred as D goes to infinity. Figure 9 illustrates how the slowdown is affected by the granularity and size of the job, assuming a delay event of $D = 1ms$ every ten seconds (we will show below that this does in fact occur due to interrupt context switching). Note that since $D = 1$, the granularity axis also shows the inverse of the grain-delay ratio. Increasing the number of nodes makes sure that at some point the delay would occur in every compute phase, thereby gradually equalizing the slowdown to be the grain-delay ratio plus one (regardless of the granularity). We note in passing that Figures 8–9 do not conflict, even though the former associates a bigger penalty with coarser grain sizes and the latter does the opposite. The reason is that Figure 8 presents the delay — D — in absolute terms, whereas Figure 9 presents the slowdown which is dominated by the grain-delay ratio. Transforming Figure 8 to do the same proves the trends are actually identical.

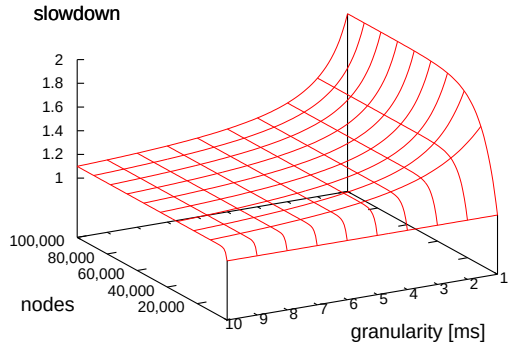

Figure 9: *As the node count grows, experiencing the $D = 1ms$ delay on each compute phase quickly becomes a certainty. For a $G = 10ms$ job, this means a slowdown of 1.1. But for a $G = 1ms$ the slowdown is 2.*

When assuming a fine enough granularity G, then it is reasonable to argue that p is actually $G \times HZ$, and that the linear noise approximation of $d_p(n)$ applies. Further assuming that ticks *always* inflict the same fixed delay of D seconds, we can apply this to the above slowdown expression and conclude that it is

$$1 + n \cdot p \cdot \frac{D}{G} = 1 + n \cdot HZ \cdot D$$

due to ticks. Indeed, if $n = 1$, this is the slowdown one would expect to get. Intuitively, for bigger n values (under the above assumption regarding p), the probability of overlap between the noise on two distinct nodes is negligible, so each additional node simply adds the delay to the overall slowdown (as the delay is propagated to the entire job in the next synchronization point).

As will shortly be demonstrated, our above "perfect world" assumption that all ticks inflict a fixed delay is incorrect. In fact, some ticks inflict a much longer delay than others. To truly get an idea about the impact of ticks (and other interrupts) on parallel jobs, we have no choice but to investigate the nature of the "D" and "p" they incur. The sort application from the previous section will not suffice for this purpose as we must be able to repeatedly do a fixed amount of fine-grained work (G) in order to obtain our goal. Likewise, Table 2 is not helpful either because it only addresses the average.

3.3 Assessing Delays and their Probability

Methodology. Our approach to measure the overhead inflicted by interrupts is the following. On an otherwise idle system we run the following microbenchmark:

```
for(int i=0 ; i < 1000000 ; i++) {
   start    = cycle_counter();
   for(volatile int j=0; j<N; j++) {/*do nothing*/}
   end      = cycle_counter();
   arr[i]   = end - start;
}
```

such that N is carefully calibrated so that the do-nothing loop[3] would take $G = 1$ms (a common grain size for parallel jobs [13]; we explored other grains in [5]). As we quickly found out, obtaining such a "definitive" N is not possible due to an inherent variance in the associated completion time. We have therefore chosen an N that attempts to place the main body of the distribution (of the values stored within the arr array) on 1ms. The heuristic we used is to run the above microbenchmark with an arbitrary $N = 10^6$, ascending-sort the content of arr, and average the samples between the 20 and 30 percentiles in the resulting sequence (3rd decile). This average was translated to an actual time (by dividing it with the CPU frequency) and served as the interpolation basis to approximate the N which is required to obtain a 1ms long computation. We note that regardless of this particular heuristic, what matters when assessing the impact of interrupts is the *span* of the values in arr, namely, how spread or concentrated are they.[4]

[3]We made sure that the loops are not optimized out.

ID	CPU	main memory			cache size			bus
	clk	size	clk	type	L2	L1		clk
						date	code	
	GHz	*MB*	*MHz*	*SD = SDRAM*	*KB*	*KB*	*Kμops*	*MHz*
M1	2.4	1024	266	DDR-SD	512	8	12	533
M2	2.8	512	400	DDR-SD	512	8	12	800
M3	3.0	1024	400	DDR2-SD	1000	16	12	800

Table 3: *The Pentium-IV machines we used.*

We ran the experiment on three Pentium IV generations as listed in Table 3 (note that each configuration is assigned an ID, *M1-M3*, for reference purposes within this text). All the machines ran a Linux-2.6.9 kernel with its default 1000 Hz tick rate. No other user processes were executing while the measurements took place, but the default daemons were alive. In order to make sure that all the perturbation we observe within the samples that populate arr are not the result of intervening daemons, but rather purely due to interrupts, the microbenchmark was also executed with the standard SCHED_FIFO realtime priority (no non-realtime process is ever allowed to preempt a realtime process, even if the former is a kernel thread; the default non-realtime priority is denoted SCHED_OTHER). Note that since compute phases are modeled as empty loops, the measured noise-impact is supposed to constitute an optimistic approximation, as phases are hardly vulnerable in terms of data locality (the only data used is the array).

Results. The very disturbing results are presented in Figure 10 (left) in a *CDF* format, where the X axis shows the phase (=loop) duration, and the Y axis shows the fraction of samples from within arr that were shorter than or equal to the associated X value. We can see that nearly 60% of the samples are indeed 1 ms long, but that the rest are lengthier. The *survival function*, shown in Figure 10 (right), exposes the tail of the distribution by associating each X value with one minus its original CDF value from the left. We can therefore see, for example, that one in 10,000 samples takes 2 ms. The statistics of the samples distribution are summarized in Table 4. With no overheads, the average should

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

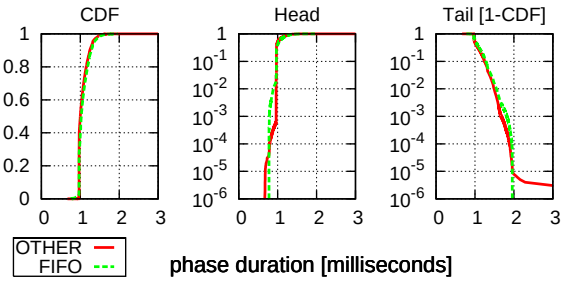

Figure 10: *Cumulative Distribution Function (CDF) of the time actually taken to run a 1ms loop on the M2 machine, under the default Linux (OTHER) and standard realtime (FIFO) schedulers. The right plot focuses on the tail by showing the survival function. Head and tail are shown with a log Y scale.*

priority	avg.	median	stddev	min	max
OTHER	1060	999	127	668	8334
FIFO	1080	1025	140	778	1959

Table 4: *Statistics associated with the two distributions presented in Figure 10, in microseconds.*

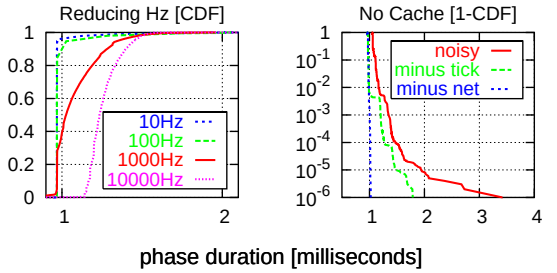

Figure 11: *Left shows the phase duration CDF of the 1ms/M2/FIFO benchmark, with decreasing tick rates. Right shows the tail of the distribution obtained by this benchmark with disabled caches (inner curves gradually subtract the direct overhead of interrupts).*

interrupt / IRQ		count	direct overhead	
description	ID		cycles	percent
tick	0	1,082,744	25,112,476,772	0.830%
network	18	5,917	102,987,708	0.003%

Table 5: *Interrupts that occurred in the M2/FIFO benchmark (total duration was 18 minutes or 3,023,642,905,020 cycles, on the 2.8 GHz machine).*

have been 1ms (=1000μs). When comparing this to the empirical averages we got, we have no choice but to conclude that the overhead penalty is 6-8%, an order of magnitude bigger than was reported in the previous section. By further examining the maximal sample duration in Table 4, we see that with the default OTHER priority the situation is especially bad, as system daemons might interfere. But a realtime process will never be preempted in favor of a non-realtime process, and our benchmark was in fact the sole realtime process in the system. The only remaining activity that is able to perturb our benchmark is therefore system interrupts.

Indeed, instrumenting the kernel to log everything that it is doing (with the help of klogger) revealed that the only additional activity present in the system while the FIFO measurements took place were about a million ticks and 6,000 network interrupts (see Table 5), indicating ticks are probably the ones to blame. On the other hand, the total *direct* overhead of the interrupts was approximately 0.8%, an order of magnitude smaller than the 8% reported above.[5] Nevertheless, we still know for a *fact* that it is ticks which prompt this bizarre result, because gradually reducing the tick frequency has a significant stabilizing effect on the results, as indicated by the CDF line that gradually turns vertical when the tick frequency is reduced (left of Figure 11).

Caching Analysis. The only remaining factor for causing the 8% penalty is the *indirect* overhead of interrupts, due to cache misses. Nevertheless, due to the distinct peculiarity of this result we decided to investigate further. We therefore repeated the experiment with all caches disabled. Under this setting, there aren't any "indirect" overheads: we have complete knowledge about everything that is going on in the system and everything is *directly* measurable. This has finally allowed us to provide a full explanation of the loop duration variability: The "noisy" curve in Figure 11 (right) shows the tail of the no-cache loop duration distribution. To its left, 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 perfectly straight line, indicating all loop durations are equal and all variability is accounted for.

We were still unsatisfied: We wanted to see those alleged misses "with our own eyes". This was made possible by using Intel's performance counters: the result of the FIFO run is shown in Figure 12. It appears that the weaker and older the machine, the more cache misses it endures. Note that the L1 graph show millions, while the L2 graph shows thousands (three order of magnitude difference), so the problem is mostly due to L1 because an L2-miss is only one order of magnitude more expensive; indeed, increasing the tick frequency from 100 Hz to 1000 Hz has an order of magnitude affect on the L1 and almost no affect on the L2. Not surprisingly, as shown by Figure 13, the per-machine cache-misses statistics perfectly coincided with the delays endured by the machines, such that additional misses were immediately translated to longer delays.

A final question we addressed was what's causing the cache misses, considering our "do nothing" benchmark hardly uses any memory? Figure 14 reveals the answer by partitioning the misses to those that occurred in user- vs. kernel-space. It turns out the problematic L1 misses

[5]Note that the *direct* component is in agreement with the values reported in Table 2 about the P-IV 2.2GHz (in sharp contrast to the overall overhead).

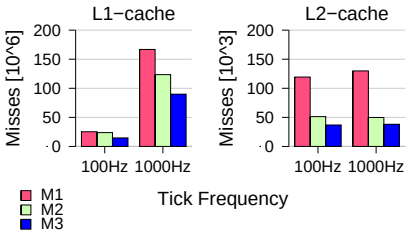

Figure 12: *The total number of L1 / L2 cache misses endured by the FIFO benchmark, for different tick rates.*

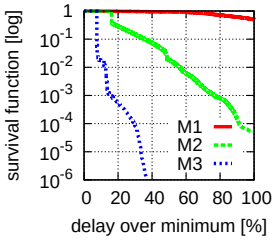

Figure 13: *Delay (%) relative to the shortest sample in the respective* arr.

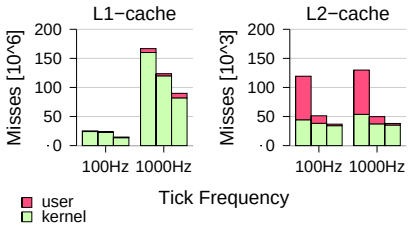

Figure 14: *Partitioning the misses from Figure 12 to user- vs. kernel-space exposes the kernel as the problem.*

are of the handler, not the microbenchmark. This portrays an almost absurd situation where the kernel-user roles are seemingly reversed, and the handler may be viewed as an application that pays the indirect price of context switching to and form the user. Nevertheless, since all kernel processing occurs at the expense of the running application (as an interrupt indeed perturbs the empty loop which is suspended until the interrupt-handling is completed), this price is immediately translated into a slowdown experienced by the benchmark.

4. THE CONTRADICTORY RESULTS

Aggregating the above two sections to one continuity puts us in an awkward situation. On the one hand, we have Section 2 stating that the overhead of 1000 Hz worth of ticks is around 1%, and that this should account for both direct and indirect effects (Table 2). On the other hand, we have Section 3 that indeed agrees with the results regarding the direct component (Table 5), but provides an undeniable proof that the indirect component is an order of magnitude more expensive (Table 4). There appears, however, to be an easy way to seemingly sort out this discrepancy: the machines used by the two sections for experimentation are, conveniently enough, different. Section 2 ends with a P-IV 2.2 GHz, whereas Section 3 begins with a P-IV 2.4 GHz. One might argue that the difference in the results is due to the difference in the experimental settings and leave it at that. But we believe this is far from being the case. Thus, in this section we explicitly address the contradiction and attempt to resolve the "overhead dispute". Apparently, some light can indeed be shed on the matter, but unfortunately some unknowns remain.

4.1 The Fasties Phenomenon

We begin by tackling a strange phenomenon that we neglected to address in the previous section. Reinspecting the middle of Figure 10 highlights this unexplained phenomenon: the loop duration distribution appears to have a small "head" composed of samples that are considerably shorter than the 1ms baseline. We denote such samples as "fasties". Table 4 specifies the shortest fasties. This is 778 μs in the FIFO case (where all perturbations are due to interrupts, because no other process was allowed to interfere with the benchmark). Recall that, on our test system, a tick occurs every 1ms. Likewise, the granularity of our benchmark was 1ms too. We therefore conjectured that the fasties that populate the distribution's head may possibly be samples that managed to "escape" the tick. We found that such samples do in fact exist, and that they indeed reside in the "head", but that they do not account for all of it.

In the face of such results, one might argue that the 778 μs value is actually more appropriate (than 1ms) to serve as the baseline, as this fastie may arguably be more representative of what would have occurred on a tickless system. Following this rationale, the overhead of interrupts would actually (presumably) be

$$100 \times \frac{1080}{778} - 100\% \approx 140\% - 100\% = 40\% \quad (!)$$

Note that our reasoning methodology is standard (and in any case is identical to the one we applied in Section 2). Upon informal consultation with Haifa's Intel research on the matter, in an attempt to explain the phenomenon, we were told that "it is possible the processor occasionally works faster" [24]...

Figure 15 shows the histogram of 10,000 samples as measured on an a Intel(R) Xeon(TM) MP CPU 3.00GHz employing a tick frequency of 1,000 Hz.[6] The left subfigure is associated with a grain size of 500 μs, implying that half the samples would experience a tick and half will not. The intent was to explicitly address the question raised above regarding the difference between the two kinds of samples. The results were largely as one would expect, namely, a bimodal distribution with nearly half the loops taking 499 μs, and nearly half the loops taking 501 μs. Not running the benchmark under a realtime regime accounts for some of the tail (extends to 8147 μs; not shown). But nothing accounts for the three fasties at 460, 486, and 492 μs.

To further complicate things, fasties turned out to be elusive. For example, when using a third of a tick's duration instead of half, the fasties mysteriously disappeared (right of Figure 15). This prompted a finer measurement, the finest possible, of adjusting the "do nothing" loop to do "something", which is to repeatedly read the cycle counter and save the values in a large array. (Writing such a program was somewhat tricky, as too often the fasties mysteriously disappeared.) Most of the time, when there were no fasties, the shortest

[6]The Xeon experiments were conducted at HP labs by David C. P. LaFrance-Linden, March 2006.

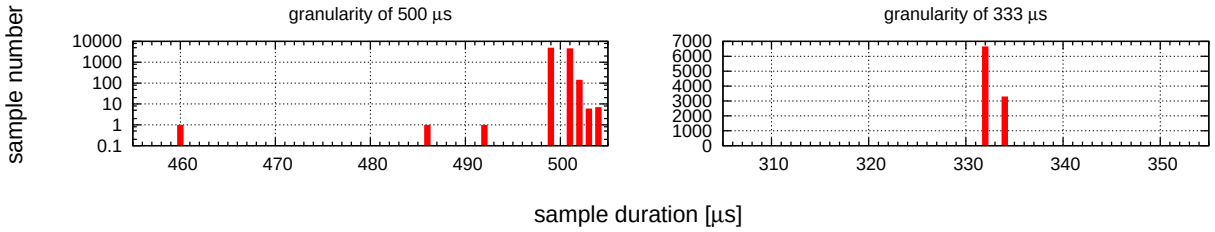

Figure 15: *A grain size of half a tick's duration (left) and a third (right) creates a bimodal histogram in which half the samples, or a third, get hit by a tick, respectively. But with the former, three of the 10,000 samples are fasties.*

and dominant delta between successive reads was 116 cycles. But when a fastie was finally encountered, then many adjacent deltas of 104 cycles were found, accounting for up to a 50 μs difference between the fastie and the norm. Strict 104-sequences, however, were never longer than 511-deltas in a row (suspiciously close to 2^8). And many of the 511-counts were separated by a single 116-delta.

We view fasties as a fascinating and curious phenomenon, which we unfortunately do not fully understand. It appears, however, that the "maybe it just runs faster sometimes" conjecture might actually be true.

4.2 The Slowies Phenomenon

We now address the other side of the samples distribution. This is the "tail" in Figure 10, which is populated by the longer loops termed "slowies". Under a conservative estimate — that doesn't factor in fasties as the baseline but rather uses the 1ms calibration target value for this purpose — this yields an 8% slower execution, with an average loop duration of 1080 μs instead of 1000. Recent measurements from real systems, that used a similar methodology to ours indicated the situation can be far worse. Specifically, a slowdown of 1.5 was observed while trying to assess the impact of noise with the P-SNAP benchmark [25] running on an IA64 cluster at a granularity of 1ms and a multiprogramming level of a mere four threads; finer grain sizes of 100–150 μs increased the slowdown factor to as high as 3 [26]. Experiments associated with the same application that were conducted on the ASC Purple [16] (composed of Power5 processors) resulted in similar findings [17]. Therefore, the problem is not just Pentium-IV, or Intel specific.[7] Slowies were also observed on a FreeBSD system [5] and on an AIX system [17], so the problem is not OS-specific too. In all cases Hyper-threading was not supported or was explicitly disabled. The bottom line is that the problem is (1) real, and (2) widespread.

The researchers that conducted the experiments on the IA64 cluster and on the ASC Purple were unsatisfied with the above results and sought a benchmark that would be more "well behaved". In the ASC Purple case, this turned out to be surprisingly similar to code appearing at the beginning of Section 3.3, requiring only a minor modification to the empty loop and turning it from "do nothing" to "do something":

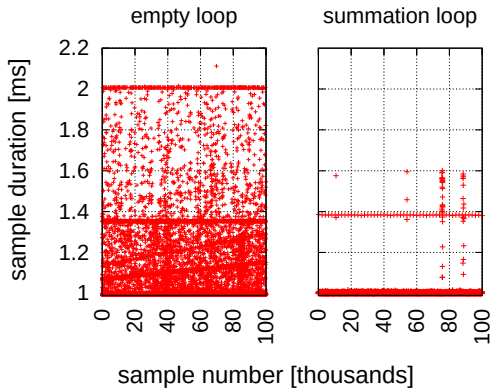

Figure 16: *The real durations of 100,000 M2/FIFO consecutive samples (that were supposed to take 1ms) exhibit significantly different variability for slightly different versions of "do nothing" loops.*

```
for(int i=0, sum=0 ; i < 1000000 ; i++) {
   start    = cycle_counter();
   for(int j=0; j<N; j++) { sum += j; }
   end      = cycle_counter();
   arr[i]   = (end - start) + (sum & zero);
}
```

The difference between this code and the one presented earlier is the sum accumulator that is bitwise 'and'-ed with a zero and added to to the sample (obviously with no affect). The compiler is presumably not sophisticated enough to figure out that the zero global variable always holds a zero value, as its name suggests. This code was not satisfactory for the IA64 case.[8] But it was enough to "rectify" the situation in the case of our $M2$ machine (Table 3). Figure 16 compares the duration of 100,000 consecutive samples, as were obtained by the older and newer versions of the microbenchmark and illustrates the dramatic change.

[7] The IA64 measurements, however, exhibited no fasties as were defined above, that is, the baseline was (1) the minimal value, and (2) the most frequent value. While we have no knowledge if this was also the case for the ASC Purple, we suspect that fasties are a Pentium-IV specific problem, e.g. they were also *not* observed on a Pentium-III, a PowerPC 970MP, and an AMD Opteron.

[8] For IA64, the desired effect was achieved by placing a different expression in the body of the inner loop, which was inspired by a Fibonacci series structure: summing up three previous elements instead of two.

4.3 Discussion

In summary, we have two slightly different versions of "do nothing" loops that produce completely different results, such that the "empty" version is much noisier than its "summation" counterpart. We feel that this observation is fascinating as well as very surprising, regardless of any practical considerations. However, there is in fact a distinct practical aspect that strongly motivates making a decision regarding which of the two versions is more representative of "the truth". The reason is that do-nothing loops often serve to model real applications, for system design and evaluation purposes (the P-SNAP benchmark [25] is a good example). In this context, "being more representative of the truth" translates to "exhibiting a behavior that is more similar to that of real applications" (which was our purpose to begin with). Additionally, if the noise phenomenon is not anecdotal, it might possibly highlight some deficiency in the design of the underlying hardware.

When weighing the two alternatives, the quieter "summation" version might initially seem to have the upper hand. Firstly, it has an inherent appeal of making the analysis simpler and inspiring the notion that everything is as it "should" be, and that the system is well understood. An argument in this spirit would be that a summation loop that does in fact "something" constitutes a better analogy to real applications that also do "something". But such an argument conveniently ignores the many "do something" loops that were nevertheless found to prompt the noisy behavior, e.g. recall the IA64 case for which both the summation loop and the Fibonacci loop were not enough, necessitating an even more complicated do-something loop that finally obtained the desired effect. Additionally, in one of the Xeon experiments, making the loop *less* complicated was actually the thing that got rid of the noise.[9] The question that follows is therefore which *objective* properties make the sufficiently-complicated-so-that-the-noisy-behavior-will-not-occur loops more representative? "A quieter system" is *not* a good answer. Especially when considering real short loops (like the inner part of matrix addition) that can be quite similar to those "misbehaved" do-nothing loops that prompted a noisy system. Further, Figure 16 suggests that even with the summation alternative the situation is not perfect, and that there is a bigger than 1/2000 chance to experience a 0.4 ms latency. This would translate to an almost definite $\frac{1.4}{G}$ slowdown for jobs that make use of only a few thousands of nodes. Should we therefore search for a "better" loop that will make this phenomenon disappear too?

A second argument that seemingly favors the quieter alternative is our analysis from Section 2. This involved a real application and yielded the conclusion that the overhead of a thousand ticks should be in the order of 1%, in contrast to the noisy conditions triggered by the "misbehaved" loops that suggest the overhead is much bigger. Such an argument is flawed in two respects. The obvious first is that this is only one application and other applications might be influenced by interrupts in other ways. The more illusive second flaw is that our methodology from Section 2, which undoubtedly appears reasonable, might nevertheless be inadequate, as will be argued next.

Consider, again, the data presented in Figure 11. If nothing else, the right subfigure *proves* we have complete knowledge about what's going on the system. Likewise, the left subfigure *proves* that interrupts have a large part in the nosiness we observe, since reducing the Hz stabilizes the system (period). However, the figure also proves we do not understand the phenomenon we're observing, as with the lowest frequency, the only remaining non-application activities are 6 network interrupts (Table 5) and 10 ticks, per second: not enough to account for the 50 (5% out of 1000) samples that are longer than 1ms (by Figure 11, left). Moreover, when normalizing the aggregated cache miss counts (from Figure 12) by the number of ticks that occurred throughout the measurement (a million), we see that there are only a few tens to hundreds of misses per tick, which by no means account for the overall overhead.

Thus, there is some weird interaction between the application and the interrupts that tickle various processor idiosyncrasies, which might not comply with the linear reasoning applied in Section 2. Specifically, it may very well be the case that, under some workloads, the mere presence of interrupts incur some (misbehaved) overhead penalty that is not strictly proportional to the exact Hz frequency being used (as suggested by the 10 Hz curve in Figure 11); in such a case, our analysis from Section 2, which targets throughput differences, would be unable to expose the actual price. Importantly, there is no real justification for assuming that the differences we observed in the behavior of the various do-nothing loops do not exist when real applications are involved. Until the phenomenon is fully understood, one can't just rule out the possibility that real loops also have fastie and slowie behaviors, and that the modes interact with or triggered by interrupts.

5. CONCLUSIONS

The results regarding the indirect overhead inflicted by periodic interrupts are inconclusive and workload dependent. Focusing on hardware clock interrupts (called "ticks"), we found that their overall impact on an array-sorting application is a 0.5-1.5% slowdown at 1000 Hz, such that the exact value is dependent on the processor and on whether each tick is slowed down by an external timer chip read or not. (Access/no-access versions are termed TSC/PIT, respectively.) Multiple experiments that varied the Hz and the system conditions revealed that the impact of ticks is robustly modeled by

$$overhead(hz) = hz \times (P_{trap} + P_{handler} + P_{indirect})$$

where hz is the tick frequency and the respective coefficients are the time to trap to the kernel and back,

[9] In this experiment a "misbehaved" loop included a function call, but compiling it with a gcc -O4 flag (that inlined the function) turn it to be "well behaved".

the direct overhead of the interrupt handling routine, and the per-application indirect overhead due to cache effects. Considering a set of increasingly faster Intel processors, we identify the following general trends:

1. The overall overhead is steadily declining.
2. For PIT, the dominant indirect component is steadily growing in relative terms (up to 6 times the direct component), but is declining in absolute terms.
3. For TSC, the overhead is dominated by the direct component (incurred by the slow external read with fixed duration across processor generations).

Generalizing the above, we show that the slowdown endured by a bulk-synchronous job spanning n nodes is

$$1 + (1 - e^{-np}) \cdot \frac{D}{G} \quad \underset{n \to \infty}{\longrightarrow} \quad 1 + \frac{D}{G}$$

where G is the granularity of the job and p is the per-node probability for one thread to experience a latency D due to an interrupt (or any other reason). When emulating a single serial G=1ms thread (by repeatedly invoking a do-nothing loop that is calibrated to spin for 1ms) and assessing the resulting duration/probability for a delay, we find that certain do-nothing loop trigger extremely high values for both p and D, indicating that the *serial* overhead due to ticks is actually an order of magnitude higher than was stated above. This type of behavior, and much worse, was observed on IA32, IA64, and Power5 processors (the latter being part of the ASC Purple [16, 17]). Importantly, some of the "do-nothing" loops involved, did enough operations to resemble "real" loops. We prove (through systematic Hz changes, detailed cache analysis and kernel instrumentation) that the effect is triggered due to an interaction between the interrupts and the do-nothing loops. But we are unable to pinpoint the exact hardware feature that is directly responsible. The question whether the phenomenon identifies a real hardware problem, experienced by real applications, remains open.

Acknowledgment

Many thanks are due to David C. P. LaFrance-Linden (HP High Performance Computing Division) for invaluable data and insightful discussions, to Greg Johnson (LANL) for providing information about the ASC Purple, and to Darren Kerbyson (LANL) for allowing the reprint of the ASCI Q measurements.

6. REFERENCES

[1] E. W. Dijkstra, "*The structure of the "THE"-multiprogramming system*". *Comm. of the ACM (CACM)* **11(5)**, pp. 341–346, May 1968.

[2] 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.

[3] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Desktop scheduling: how can we know what the user wants?*". In 14th *Int'l Workshop on Network & Operating Syst. Support or Digital Audio & Video (NOSSDAV)*, pp. 110–115, Jun 2004.

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

[5] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "*System noise, OS clock ticks, and fine-grained parallel applications*". In 19th *ACM Int'l Conf. on Supercomput. (ICS)*, pp. 303–312, Jun 2005.

[6] D. Tsafrir, Y. Etsion, and D. G. Feitelson, *Is your PC secretly running nuclear simulations?* Technical Report 2006-78, The Hebrew University of Jerusalem, Sep 2006. Submitted.

[7] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Process prioritization using output production: scheduling for multimedia*". *ACM Trans. on Multimedia Comput. Commun. & Appl. (TOMCCAP)* **2(4)**, pp. 318–342, Nov 2006.

[8] P. Kohout, B. Ganesh, and B. Jacob, "*Hardware support for real-time operating systems*". In 1st *IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 45–51, ACM Press, New York, NY, USA, Oct 2003.

[9] "*Ece476 aos: experiments with a preemptive, multitasking OS for Atmel Mega32 microcontrollers*". URL http://instruct1.cit.cornell.edu/courses/ee476/RTOS/oldindex.html, Sep 2005.

[10] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "*Analysis of system overhead on parallel computers*". In *IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2004)*, Rome, Italy, December 2004.

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

[12] 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.

[13] 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.

[14] 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.

[15] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin, "*A performance comparison through benchmarking and modeling of three leading supercomputers: blue Gene/L, Red Storm, and Purple*". In *ACM/IEEE Supercomputing (SC)*, p. 74, Aug 2006.

[16] "*LLNL's ASC Purple*". URL http://www.llnl.gov/asc/computing_resources/purple/purple_index.html.

[17] G. Johnson, "*Slowies on the ASC Purple*". Feb 2007. private communication.

[18] J. K. Ousterhout, "*Why aren't operating systems getting faster as fast as hardware?*". In *USENIX Summer Conf.*, pp. 247–256, Jun 1990.

[19] "*Pentium 4*". Wikipedia, the free encyclopedia, URL http://en.wikipedia.org/wiki/Pentium_4.

[20] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson, "*Fine grained kernel logging with Klogger: experience and insights*". In *ACM EuroSys*, Mar 2007.

[21] 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.

[22] L. McVoy and C. Staelin, "*lmbench: portable tools for performance analysis*". In *USENIX Ann. Technical Conf.*, pp. 279–294, Jan 1996.

[23] L. McVoy and C. Staelin, "*LMbench - tools for performance analysis*". URL http://lmbench.sourceforge.net/, 2007.

[24] A. Mendelson, "*Attempting to explain the fasties phenomenon*". Oct 2005. Private communication.

[25] "*P-SNAP v. 1.0. open source software for measuring system noise. la-cc-06-025*". Available from http://www.c3.lanl.gov/pal/software/psnap.

[26] D. C. P. LaFrance-Linden, "*Slowies on an IA64 cluster*". Feb 2007. private communication.