Hardware Parallelism: Are Operating Systems Ready? (Case Studies in Mis-Scheduling)

Eitan Frachtenberg
Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences Division
Los Alamos National Laboratory
eitanf@lanl.gov*

Yoav Etsion
School of Computer Science and Engineering
The Hebrew University
Jerusalem, Israel 91904
etsman@cs.huji.ac.il

Abstract

Commodity parallel computers are no longer a technology predicted for some indistinct future: they are becoming ubiquitous. In the absence of significant advances in clock speed, chip-multiprocessors (CMPs) and symmetric multithreading (SMT) are the modern workhorses that keep Moore's Law still relevant.

On the software side, we are starting to observe the adaptation of some codes to the new commodity parallel hardware. While in the past, only complex professional codes ran on parallel computers, the commoditization of parallel computers is opening the door for many desktop applications to benefit from parallelization. We expect this software trend to continue, since the only apparent way of obtaining additional performance from the hardware will be through parallelization.

Based on the premise that the average desktop workload is growing more parallel and complex, this paper asks the question: Are current desktop operating systems appropriate for these trends? Specifically, we are interested in parallel process scheduling, which has been a topic of significant study in the supercomputing community, but so far little of this research has trickled down to the desktop.

In this paper, we demonstrate, using several case studies, that contemporary general-purpose operating systems are inadequate for the emerging parallel desktop workloads. We suggest that schedulers designed with an understanding of the requirements of all process classes and their mixes, as well the abilities of the underlying architecture, might be the solution to this inadequacy.

1 Introduction

Computer performance has improved at an exponential rate since the introduction of the first microprocessor. This improvement is particularly visible in the way commodity computing has permeated most aspects of our lives. To allow this growth, the operating system (OS), acting as the arbiter between hardware and software, has had to keep pace with advances in both software and hardware. Subsequently, we have witnessed marked improvements in the way OSs handle input and output (I/O) devices, user interfaces, networking, and interoperability. Process scheduling, however, has progressed little since the introduction of timesharing. While simple adjustments, tuning, and handling of special cases was incrementally added to all commodity schedulers, the basic scheduling principles have remained largely unchanged. This stagnation could be tolerated because the steady rate of performance increases masked most scheduler inefficiencies. However, in this paper, we posit that current trends in hardware and software will require a shift to new scheduling policies that are better suited for parallel hardware and richly complex workloads. We support these claims with experimental data on scheduling performance under forward-looking assumptions on commodity workloads and architectures.

The rest of this section describes recent trends in hardware and software technologies, while Section 2 elaborates on why these changes will render current commodity schedulers obsolete. To substantiate this claim, the main part of this paper (Section 3) demonstrates, using several case studies, some of the problems on which we expect schedulers to perform poorly with future hardware and software.

Hardware Trends: The Move to Concurrency. For more than two decades, the phenomenal increase of microprocessor performance has fueled a similarly explosive growth in commodity hardware and applications. However, the increase in single-processor performance is now showing signs of slowing down. Already, supercomputers (with performance growth that outpaces the growth predicted by Moore's Law) have shifted from single-processor architectures to parallel ones. Just as innovations in Formula One race cars are the precursor for many improvements in mainstream vehicles, innovations in high-end computers often portend advances in commodity architectures—as was the case for example with network and storage technologies.

*This work was funded in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number NBCH3039004. Los Alamos National Laboratory was funded in part by the Accelerated Strategic Computing program of the Department of Energy, and is operated by the University of California for the US Department of Energy under contract W-7405-ENG-36.

To maintain high rates of performance growth, manufacturers are turning to parallelism [21, 25]. Even single-processor and desktop computers are shifting toward parallelism, offered in SMT (though not always successfully [41]) and multicore (CMP) processors from Intel, AMD, IBM, Sun, and others [25, 35]. The Cell processor for media applications [21] is an example of such emerging architectures involving a relatively large number of special-purpose computing cores.

While parallelism itself will not solve all the problems that are curtailing performance growth—such as energy budget, cooling capacity, and memory performance—we focus this paper on the inherent problems facing operating systems on the way toward adequate support for parallelism.

Software Trends: Increased Diversity. We assume in this paper that ubiquitous parallel hardware brings with it complex workloads and that software will become increasingly more parallel and demanding. We posit that consequently, the scheduling requirements from the OS will grow more complex.

The wide availability of parallel hardware should conceivably provide an incentive for, and spark growth in, parallel programming. Already a typical uniprocessor desktop with a multitasking OS runs multithreaded applications, motivated by considerations of resource overlapping, increased responsiveness, and modularity [15]. These applications range from the multithreaded Web browser to database and Web servers. The increasing parallelism in hardware has also revitalized research in compiler autoparallelism [18, 24].

History teaches us that commodity hardware innovations often trickle down to software development quickly. If this trend continues, the move to hardware parallelism in desktops will soon be expressed in the emergence of novel parallel desktop applications, which leads to the central question in this paper: Are desktop operating systems ready for this change?

The need to manage increased *hardware* parallelism confronts operating systems with several challenges other than increased *software* parallelism: increased resource locking overheads, increased system noise caused by periodic interrupts, and the need for a more efficient timing mechanisms [12, 39]. We restrict the scope of this paper to process scheduling, but we believe that the principles that we lay out for a parallel-aware OS are also a good start for solving these other problems.

2 Problem Discussion: Parallel Workloads on Desktop Schedulers

The new parallelism in computer architectures challenges the general-purpose OS with workloads that were not factored into its design. To elaborate on this claim, this section expands on the inadequacy of desktop schedulers to handle common parallel-programming paradigms and workloads. The next section presents some actual case studies for these scenarios.

Changes in desktop computer architecture promote changes in typical desktop workloads, thus motivating research in OS and process scheduling. The increasing clock speed trend of the past two decades has promoted multimedia computing, motivating research into this field, and subsequently led to some novel scheduling policies for soft real-time and multimedia workloads [4, 8, 10, 19, 28, 29]. Nevertheless, the performance of the commodity OS does not have a good track record of scaling up with the underlying hardware performance [31], and, in particular, does not perform well with multiprocessor workloads [33]. Parallel job scheduling has been studied primarily in the context of supercomputers and clusters [14]. It has rarely been studied in the context of the commodity desktop. We predict that in the near future, critical issues of parallel thread scheduling will surface on the desktop as the degree of parallelism in commodity machines increases. Even today's simplest parallel machines, such as SMTs or small SMPs and CMPs, already have difficulties with many applications and workload mixes [1, 5, 20]. Commodity schedulers are challenged at all levels of parallel execution, from SMTs [5, 37] through SMPs [1, 40], the cluster [2, 13, 17, 39], and even supercomputers [23, 32, 40].

Current parallel programming paradigms are closely based on Flynn's classic categories [16], as follows:

- Single-Instruction-Multiple-Data (*SIMD*), also known as the *workpile* programming model. Under this model, multiple datasets are processed in a symmetric, independent manner.
- Multiple-Instruction-Single-Data (*MISD*), or the *systolic/pipelined* programming model. Under this model, the same dataset undergoes several independent transformations sequentially.
- Multiple-Instruction-Multiple-Data (*MIMD*), also known as the *Bulk-Synchronous Parallel* (BSP) programming model. Under this model, both the dataset and the computation are divided into subsets, with each thread processing *several* of the data subsets. The entire computation is divided into computational phases and synchronizing group communication phases.

The predominant approach to multiprocessing in general-purpose OSs is to treat each processing element as an independent entity—processes/threads are migrated between processing elements in an attempt to balance cache affinity needs with CPU load imbalance [7, 26, 27, 34]. This approach only supports the *workpile* model, since it does not take into account any interprocess dependence. The result is that contemporary general-purpose schedulers are too focused on satisfying a small set of requirements. They miss the "big picture" and overlook the two requirements that we believe are critical for performance and efficiency for parallel desktop workloads: separation of co-interfering processes and coscheduling of collaborating processes.

In our opinion, the growing popularity of parallel programming mandates OS support for all parallel compu-

Name	Processor technology	OS	CPUs
P3	Pentium-III 664 MHz	Linux 2.4.8	1
P4	Pentium-IV 2800 MHz	Various	1
ES40	Alpha EV6 833 MHz	Linux 2.4.21	4
ES45	Alpha EV6 1250 MHz	Tru64 5.1	4
IBM-2.4	Pentium-III 550 MHz	Linux 2.4.22	4
IBM-2.6	Pentium-III 550 MHz	Linux 2.6.9	4
Potomac	Xeon 3330 MHz	Linux 2.6.11	4(8)

Table 1. Experimental platforms (number in parenthesis is logical processors on SMTs)

tation modes from manual coarse-grained parallelization (implemented utilizing either paradigm), to compiler auto-parallelization techniques that are mainly based on BSP. The main challenge that general-purpose OSs are facing when it comes to scheduling both BSP and systolic parallel applications is how to incorporate the interthread dependencies in the process scheduling. The predominant approach manually conveys the dependence information through specific interfaces to the OS [9]. In contrast, a more interesting approach is to deduce these dependencies implicitly by tracking interprocess communication at runtime and deriving the resulting scheduling requirements—as shown in several studies [11, 17, 40, 43]. This approach is especially effective in uncovering hidden dependencies—for example, those involving user application and system dæmons [11]. The next section translates these challenges into actual mis-scheduling scenarios.

3 Case Studies

To demonstrate the importance of OS awareness of parallel constraints, we discuss several case studies—including both systolic and BSP applications, using various software and hardware variations.

These examples were kept intentionally simple 1) to facilitate reproduction of our results and "benchmarks"; and, 2) to reduce the effect of complex, unmodeled relationships between hardware and software components, such as memory hierarchy considerations, I/O performance, etc. We did attempt, however, to evaluate a wide range of hardware and OS configurations, detailed in Table 1.

3.1 Systolic Paradigm: A Movie Story

The scenario we describe here may be familiar to most readers. It involves running a latency-sensitive application—such as a media player or a voice-over-IP application with other tasks in the background, only to suffer from skipped frames and poor playback quality. If the root cause of the problem were inadequate computing capacity, we would be forced to accept lower quality or less multitasking. However, the problem results strictly from mis-scheduling, as evidenced by the many attempts to address latency-sensitive applications in the scheduling literature [4, 8, 10, 19, 29, 30]. Paradoxically, simple, low-resource, special-purpose hardware such as a portable video player can present a movie smoothly, while a movie played on a significantly more powerful PC is often jittery when another application (e.g., an anti-virus scanner or a download client) is running in the background. Media scheduling has received much attention lately and recent Linux schedulers provide far better support than older versions. We wish to leverage this experience in parallel desktop scheduling.

3.1.1 Experiment Description

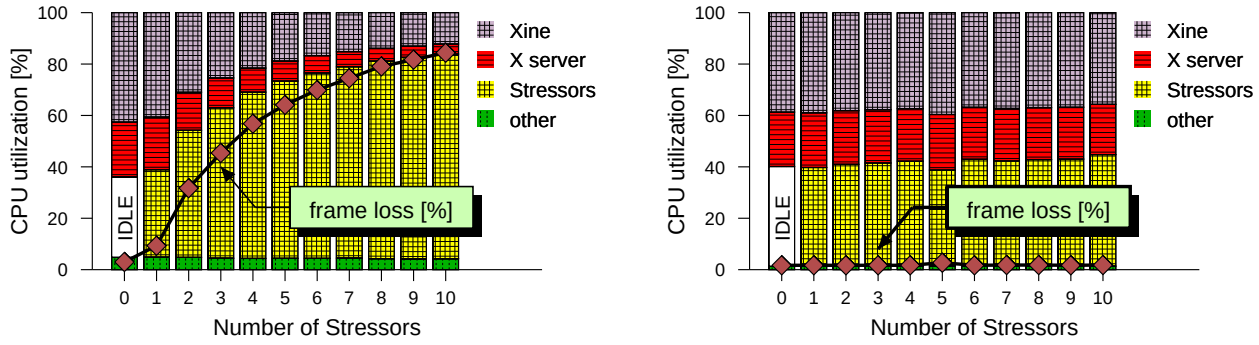
Our experiments are based on measuring the performance of Xine, a multithreaded movie player. Xine's design is systolic parallel, with the two most important threads performing the decoding and displaying of frames. Our workload consists of running Xine with increasing background load and analyzing how different Linux schedulers are affected. This workload was chosen to demonstrate how a scheduler incognizant of process dependencies might degrade the throughput of a real-life parallel application. Moreover, Xine is implicitly dependent on the graphical subsystem—specifically on the X windows system and on the video card's graphical processing unit (GPU). These two elements add an implicit stage (X server) to the systolic pipeline (which the process scheduler should identify), and an explicit stage (GPU) that handles the rendering per se, but is managed by the device driver layer rather then the scheduler, and is thus not discussed in this paper. We used the Klogger kernel logging framework to log context switches [12], thus exposing CPU consumption patterns. During the measurements, Xine processed a short MPEG clip, which resides in a memory filesystem, to avoid I/O clutter. The results, however, are mostly relevant to MS Windows as well, since both schedulers are based on similar principles [26, 34].

The quality metric used to evaluate the schedulers is the achieved frame rate (calculated using Xine's dropped frames statistics). This metric directly embodies the user experience, filtering out any scheduler inefficiencies too minor to be perceived by the user [36, 15].

3.1.2 Base Case: Uniprocessors

To expose the user-visible effects of poor scheduling decisions on movie playing, we reproduce here results from a lower-end machine (P3), exposed to disruptive background activity[1]. The experiments consisted of running Xine with an increasing number of CPU stressors (that iterate on an empty loop) with two Linux schedulers: 2.4.8, and an improved Linux research version (based on Linux 2.4) that automatically identifies and prioritizes interactive processes [10, 11]. Here, we measure the distribution of CPU utilization among processes and the user-noticeable percentage of skipped frames that Xine suffers. The Linux 2.4.8 results, shown in Figure 4(a), suggest that when a generic scheduler does not respect the scheduling requirements of a soft-real-time application's entire systolic thread group (Xine), the increase in background load translates directly to a decrease in interactive responsiveness.

[1] Even though computer performance continues to grow, low-end processors continue to be of interest because of their wide use in portable, power-aware, or embedded environments. Even on high-end machines, workloads continue to require increasingly more resources (Section 1), thereby increasing the background interference and load.



(a) Linux 2.4.8 scheduler

(b) Improved multimedia-aware scheduler

Figure 1. Comparison of two schedulers running Xine under increasing background load on "P3"

Conversely, an interprocess dependency tracking scheduler (Figure 1(b)) prioritizes the thread group as a whole. By tracking interprocess communication, this scheduler identifies all the processes/threads involved in the interactive computation, implicitly uncovering Xine's entire systolic pipeline and scheduling it accordingly, thus enabling good, sustained frame rates [10, 11].

3.1.3 Emerging Platform: Multiprocessors

Uniprocessor systems are limited to temporal partitioning, whereas multiprocessor schedulers can also utilize spatial partitioning. To show the inadequacy of the standard workpile-oriented scheduler in handling systolic applications and to expose the potential in spatial partitioning, we repeated the Xine experiment on IBM-2.6. Furthermore, to demonstrate that effective process placement is possible (albeit manually) even when the memory bus is loaded, we modified the stressors to access randomly a memory range larger than the L2 cache.

Figure 2(a) shows the relatively poor and noisy performance of the default Linux scheduler in this scenario. When running two or fewer stressors, Xine and X received a dedicated CPU each. The temporary performance loss observed when we were running three stressors was caused by the scheduler's load-balancing attempt to run both Xine and X on the same processor, placing them in competition for CPU resources. When we were running four to seven stressors, however, the scheduler could not balance the load and again separates Xine and X, allowing each to compete with different stressors, albeit with a slight interactive priority gain, effectively replacing the idle process with stressors. These zig-zag migrations explain the inconsistent performance seen throughout the measurement. Occasionally, these migrations also led to a positive effect on Xine—when the imbalances improved its performance (10 stressors).

If we manually assign Xine and X to run on processor 0 exclusively, Xine's performance is more consistent, as shown in Figure 2(b). The kernel log reveals that the two performance drops (12 and 18 stressors) are caused by the awakening of the kernel swap dæmon and not by the stressors. However, one CPU is still not enough for both X and Xine for adequate interactive experience, as attested by the high frame loss.

We further refined the experiment by manually partitioning the machine placing Xine and X on processors 0–1 and the stressors on the others (Figure 2(c)). Performance improved in terms of lower frame loss, since X and Xine now ran on dedicated processors. However, kernel dæmon activity still occasionally drove X and Xine to the same processor, again pitting them against each other in a competition for insufficient CPU resources. Even though this activity was very short, the scheduler only migrated Xine back after a few minutes, leaving processor 1 idle for four consecutive measurements. These effects were repeatedly verified.

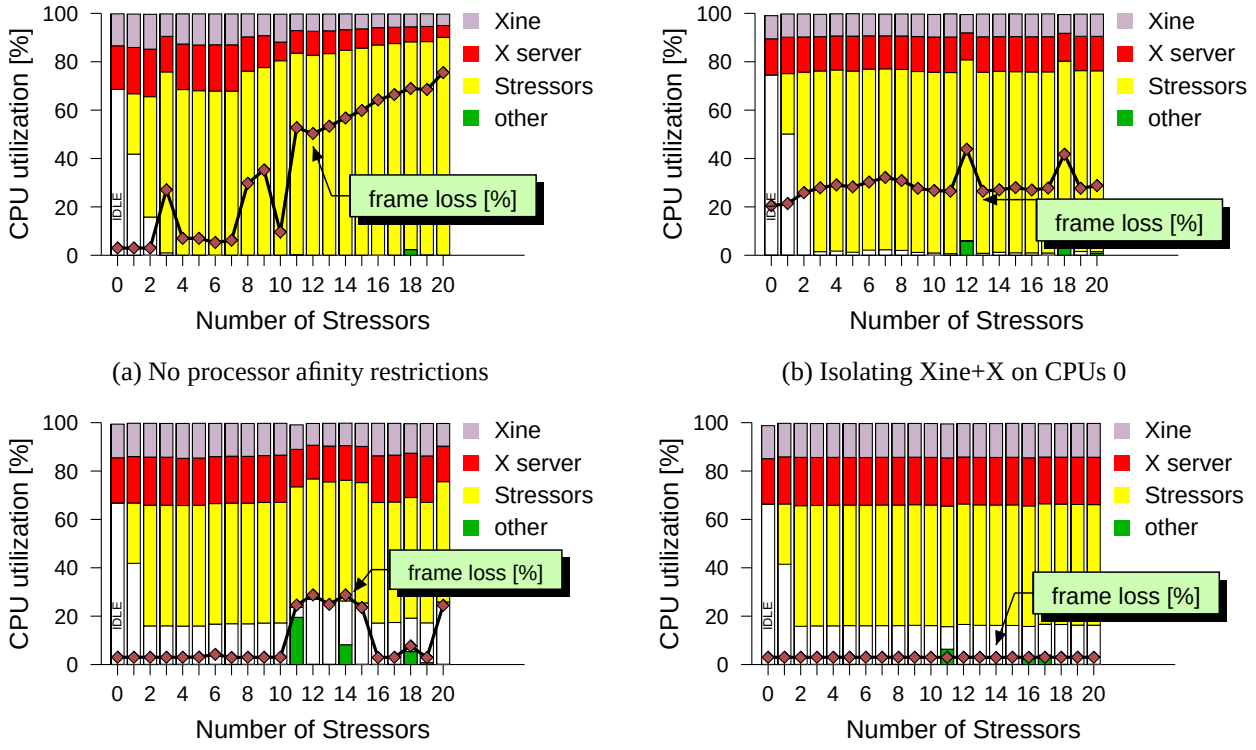
Only when manually fixing X to processor 0 and Xine to processor 1, and letting the scheduler manage the stressors on processors 2–3 as (Figure 2(d)) could we observe predictable and smooth movie playback.

3.1.4 Summary

Our first set of experiments shows that a multimedia-aware scheduler can produce a much better movie experience than a naive one. Our second example shows that with adequate knowledge of process requirements, scheduling on a loaded multiprocessor can be successful, but this approach currently requires manual intervention. Our point is that in both cases, knowledge about process needs and machine state can produce far better schedules. We believe that the required knowledge can be garnered automatically by future schedulers so that manual intervention is not required. In this section we have explored this statement for systolic parallel applications. The next section discusses how this statement applies to parallel BSP applications.

3.2 BSP Paradigm: Synchronization

Parallel job scheduling—an active and developed topic in the realm of supercomputing—has not received much attention in the context of commodity machines. Nevertheless, if we are indeed heading toward commodity parallel architectures as described in Section 1, the scheduling requirements of BSP jobs in mixed workloads will have to be addressed. Therefore, we designed a simple experiment to measure the basic interferences in a mixed workload of sequential and parallel applications.



(a) No processor afinity restrictions

(b) Isolating Xine+X on CPUs 0

(c) Isolating Xine+X on CPUs 0–1

(d) X on CPU0, Xine on CPU1

Figure 2. The effect of imposing processor affinity heuristics on the Linux 2.6.9 scheduler running on "IBM-2.6". The utilization is accumulated over all four CPUs.

3.2.1 Experiment Description

A wrapper program launches a set of sequential programs ("stressors") and parallel programs. The stressors consume cycles with simple computation running in an infinite loop. The parallel programs execute the same computation but for a predefined number of iterations. Each parallel program is composed of a number of threads equal to the number of processors less one[2]. Every few hundred of iterations, the parallel threads synchronize with each other using standard Unix semaphores. This simplified BSP structure captures the essential behavior of many real parallel programs [42] while allowing an exposition of scheduler-specific effects—without getting into issues of I/O, memory bandwidth, and instruction mixes.

The wrapper program waits for the completion of all the parallel programs before it terminates the sequential programs and measures the total time to completion. By varying the number of parallel programs and stressors, we can measure the effect of the host OS scheduler on different workload mixes. Additionally, the wrapper program has a gang scheduling mode in which time is sliced into slots. On each slot, either one parallel program, or all the sequential programs are running exclusively. (Other programs are suspended with a SIGSTOP signal.) Thus, parallel programs see a dedicated view of the machine for the duration of their timeslice, an approach that facilitates their frequent synchronization (because all threads are in running state when synchronization is required). Time slices are switched in a round-robin fashion.

To reduce the effect of variability and noisiness in the results, we repeated each run at least five times, discarded the minimum and maximum results, and averaged the remaining run times. By focusing on completion time, this experiment is designed to measure the effect on the parallel jobs, since the sequential applications continue to run for an indefinite amount of time, merely representing a relatively constant background load.

3.2.2 Results and Analysis

We ran this experiment with 1 to 10 parallel programs and 0 to 20 sequential stressors on all the parallel architectures in Table 1. Figure 3 compares the performance of the schedulers on ES40. As might be expected, the total run time for both gang and default scheduling increases with the number of parallel jobs. On ES40, the default scheduler (Linux 2.4.21) does a poor job of coscheduling the threads of the parallel programs. The result is higher run times than under GS. The gap in the scheduler performance grows as the load increases, both in terms of parallel and sequential programs. On the opposite end, running a single parallel program with few stressors yields performance that is slightly worse for GS. The main reason for this result is that the single parallel job, as the only one that

[2]The one unallocated processor allows the mitigation of the effect of system dæmons and unrelated user-level processes that might wake up periodically, especially when gang scheduling (GS) is involved [13, 32]. A "pure" GS implementation at the OS level would simply set aside a timeslot for housekeeping tasks and low priority processes [38].

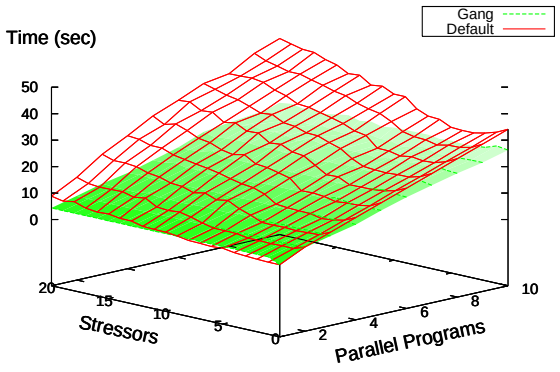


Figure 3. Gang scheduling vs. default OS scheduling on ES40

blocks, gets prioritized in Linux 2.4 over the nonblocking sequential jobs. In addition, the relatively high time quantum of GS (compared to total execution time) and overhead added by the extra scheduling layer contribute to some performance degradation with GS.

We cannot use absolute run times to compare default and gang scheduling across architectures, because absolute run times depend on the architecture. We therefore normalize the results by calculating and plotting the slowdown of the default scheduler when compared to GS. The results, shown in Figure 4, are compared to the unit surface, which represents equal run time of the gang and default scheduler. A slowdown higher than one represents a scenario in which gang scheduling performs better than the default scheduler, while the opposite is true for values below the unit surface.

We start by comparing the effect of different OS schedulers on the same architecture (Figures 4(a) and 4(b)). In the absence of sequential jobs, parallel jobs that block on communication tend to self-synchronize, since a thread that tries to synchronize with a blocked thread will eventually block itself, releasing the CPU for another program that is ready to run [3]. Consequently, explicit gang scheduling is not required to achieve synchronization in a mostly-parallel workload when blocking synchronization is used. This effect is evident when one looks at the Stressors=0 axis of the figures. In fact, the OS timer interrupt frequency (HZ in Linux) in version 2.6 is 10 times finer than in 2.4. This results in a faster context switch to a newly-unblocked parallel process, and, therefore, the parallel programs are prioritized over the CPU-bound sequential programs. Indeed, self-synchronization is so effective in 2.6 that the default scheduler significantly outperforms the coarse-grained gang scheduler in low sequential loads. However, as the number of stressors increases and can no longer be accommodated on the spare processor, synchronization of the parallel programs is hampered and the programs slow down. This fact is particularly evident on the Parallel=1 axis, where the weight of the single parallel program in the workload diminishes as the number of stressors grows. By not ensuring coscheduling of its threads, the Linux scheduler hampers the parallel program's progress.

Another important property of a scheduler is its stability and predictability. The IBM-2.6 figure shows a much smoother surface than that of IBM-2.4. Analyzing the results per data point reveals that the average span of measured values for the Linux 2.4 scheduler is more than triple that of Linux 2.6's scheduler, possibly because of the coarser grain at which scheduling decisions are made. In addition, the span of measured values for both OSs is larger than that of GS, especially for higher loads. This fact is a direct consequence of the scheduling order and determinism forced by GS [38].

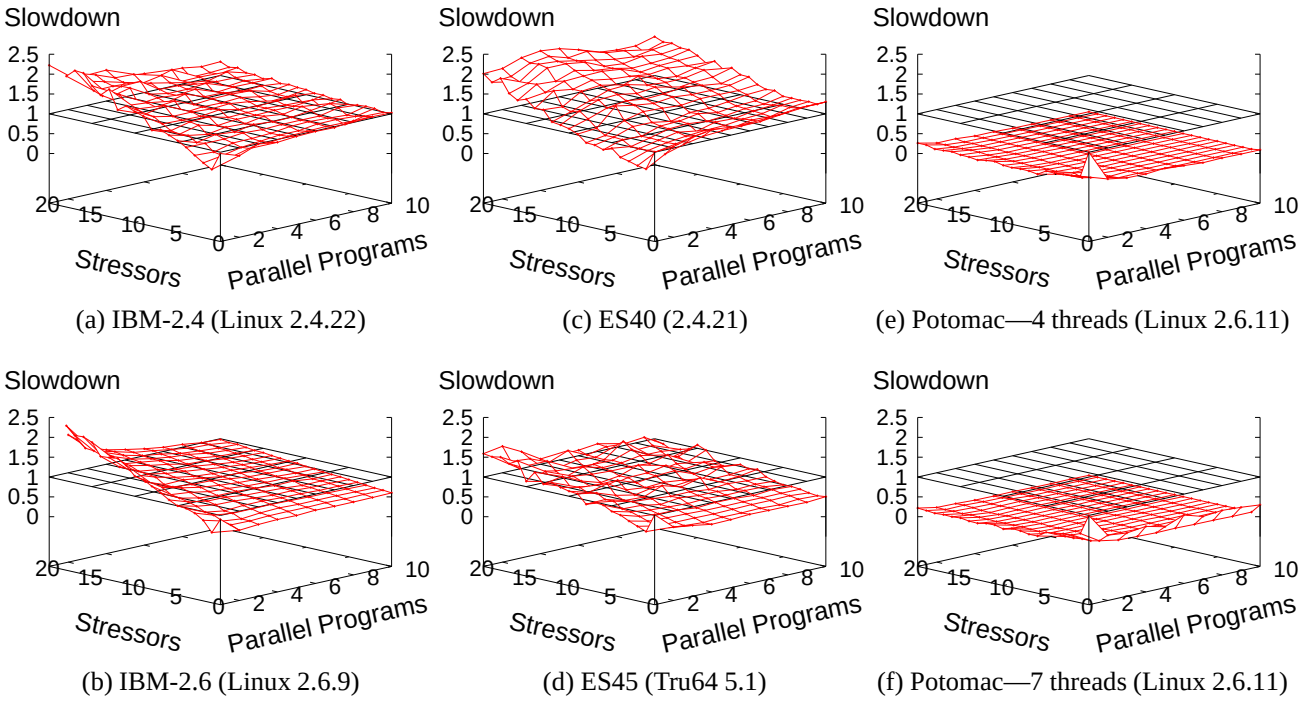
The next pair of figures (ES40 and ES45) again compares two different OS schedulers (Linux 2.4 and Tru64 5.1) on similar architectures. In this experiment, the Linux 2.4 scheduler performs even worse than on the IBM in terms of slowdown. Because of faster processors, the actual synchronization granularity of the parallel programs is finer on the ES40 than on the IBM, increasing its sensitivity to mis-scheduling and noise [22, 23, 39]. Tru64's scheduler, designed from conception for multiprocessor servers, does remarkably well with the parallel programs. For most runs with eight stressors or less, Tru64's scheduler outperforms GS regardless of the number of parallel programs. It does, however, show significant variability in results, occasionally taking twice as long to complete the same experiment. We believe this problem is related to Tru64's high susceptibility to OS noise when all processors are employed, as shown in a related analysis on a nearly identical architecture [32]. Unfortunately, we do not have access to Tru64's source code to verify this hypothesis.

In the examples described so far, GS is a preferable policy for the *parallel application*, potentially at the cost of sequential and interactive programs. Is this always the case? The last pair of graphs shows that enforcing coscheduling on a partial resource allocation or on an Asymmetric MultiProcessor (AMP) is not always beneficial to the parallel program. Figure 4(e) shows the results of running the same workload (with four threads per parallel program) on the Potomac, which has eight logical processors (four hyperthreaded Xeon MPs)[3]. Limiting the number of threads to four allows each thread to run on its own processor but wastes many internal compute resources that are otherwise filled by the default scheduler that sees eight CPUs. Moreover, resource wasting is not the only problem in an AMP, as shown in Figure 4(f). If we increase the number of threads per parallel program to seven so that most logical processors are busy, we again observe poor GS performance. The source of the problem is that every two virtual processors share many of their internal resources, so the threads of each parallel program end up competing with each other instead of complementing each other [37].

3.2.3 Summary

This section has demonstrated one principle that surfaces in many studies on parallel job scheduling: the need to

[3] With the widespread availability of dual-core processors and the near-future release of quad-core processors, Potomac may be considered a good representative of future desktop machines.



(a) IBM-2.4 (Linux 2.4.22) (c) ES40 (2.4.21) (e) Potomac—4 threads (Linux 2.6.11)

(b) IBM-2.6 (Linux 2.6.9) (d) ES45 (Tru64 5.1) (f) Potomac—7 threads (Linux 2.6.11)

Figure 4. Slowdown comparison of various architectures and OSs

coschedule tightly coordinated tasks, especially in mixed workloads. Our experiments confirm that as load increases and the workload mixes more sequential and parallel jobs, coscheduling offers substantial performance benefits, even in the form of the relatively rigid gang scheduling algorithm [17]. While this workload may or may not be representative of the richer workloads that will emerge with parallel desktops, we believe it captures the main weaknesses of naive parallel scheduling.

As architectures grow more parallel and heterogeneous, coscheduling can also become a boon or a bane for unrelated, (and unsynchronized) tasks. Coscheduling resource-complementing processes can lead to high parallel speedups, while ignoring these considerations can lead to a situation in which processes compete with each other for shared resources (such as the memory bus or processor cache) [1, 5, 6, 20].

4 Conclusions

As commodity computers and their workloads continue to evolve, the schedulers that manage them become less adequate. The limitations of aging schedulers attract little attention because the schedulers are good enough for single-processor computers. However, with the move to parallel architectures, we can no longer afford to ignore these limitations. On the other hand, while parallel job scheduling is a challenging research area with many open questions still under active study [14], it may not suffice, because these techniques were developed for much more homogeneous workloads.

The scheduling of a mixed workload, combining traditional and new desktop applications with parallel jobs, is only now starting to gain attention in conjunction with the expected prevalence of commodity parallel architectures.

We posit that if they are to be truly general-purpose for current and future workloads, commodity schedulers cannot maintain the same anachronistic principles with which they were conceived 30 years ago (with occasional ad-hoc provisions added to address specific problems such as multimedia). Our experimental results show that contemporary scheduling methods used on parallel workloads can lead to significant application slowdown, diminished user experience, and even unpredictability. Furthermore, we show how applying common parallel scheduling techniques can significantly affect performance. On the other hand, our experiments show that often, by employing some knowledge about the process requirements and the system's capabilities, one can manually find very good schedules. We believe that by adapting techniques from different scheduling domains, this can be done automatically instead of manually. We propose the following two principles to guide such a scheduler:

- **Maximizing collaboration:** Processes that require or benefit from coscheduling should be coscheduled.
- **Minimizing interference:** Processes with conflicting requirements should be separated in time or space.

Note that these principles also implicitly address different architectures and workloads. For example, interference can be the result of scheduling two processes on the same hyperthreaded processor, and assigning one of them to a different processor (spatial partitioning) can benefit both processes. Another example: Processes that are collaborating (parallel threads) or co-dependent (data dependency) could require coscheduling.

While these principles are not new, they are not reflected well in current desktop schedulers. In part, this situation arises because schedulers have an incomplete picture of processes and architecture. We believe that by combining insights from domain-specific scheduling research, with further research, one can design a unified scheduler that is able to discern which processes collaborate, which interfere with each other, and how they all fit together with the hardware.

We plan to evaluate these ideas by performing actual experiments on a variety of contemporary workloads and architectures. In the distant future, scalability issues could warrant even more changes in the scheduler. Tens and hundreds of cores may require the OS to be of a distributed nature. Additional challenges might stem from power management heterogeneity considerations. We hope these considerations can also fall under the umbrella of the principles we suggest in this paper.

References

[1] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *32nd Intl. Conf. Parallel Processing (ICPP)*, Oct. 2003.

[2] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pages 267–278, May 1995.

[3] A. C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Trans. Computer Systems*, 19(3):283–331, Aug. 2001.

[4] 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 (MMCN)*, Jan. 2002.

[5] J. R. Bulpin and I. A. Pratt. Multiprogramming performance of the Pentium 4 with hyper-threading. In *2nd Annual Wkshp. Duplicating, Deconstruction and Debunking (WDDD)*, pages 53–62, June 2004.

[6] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *14th ACM Symp. Operating Systems Principles (SOSP)*, pages 120–133, Dec. 1993.

[7] B. Darmawan, C. Kamers, H. Pienaar, and J. Shiu. *AIX 5L Performance Tools Handbook*. IBM Redbooks, 2nd edition, Aug 2003.

[8] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *17th ACM Symp. Operating Systems Principles (SOSP)*, pages 261–276, Dec. 1999.

[9] Y. Etsion and D. Tsafrir. A short survey of commercial cluster batch schedulers. Technical Report 2005-13, Hebrew University, May 2005.

[10] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Desktop scheduling: How can we know what the user wants? In *14th ACM Intl. Wkshp. Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 110–115, June 2004.

[11] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Process prioritization using output production: Scheduling for multimedia. *ACM Trans. Multimedia Computing, Communications, and Applications*, 2(4), 2006.

[12] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson. Fine grained kernel logging with klogger: Experience and insights. Technical Report 2005-35, School of Computer Science and Engineering, Hebrew University, June 2005.

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

[14] D. G. Feitelson, L. Rudolph, and U. Schwigelshohn. Parallel job scheduling – A status report. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *10th Wkshp. Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2004.

[15] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. In *9th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, number 9, pages 129–138, Nov. 2000.

[16] M. J. Flynn. Very high-speed computing systems. *IEEE*, 54(12):1901–1909, Dec 1966.

[17] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez. Flexible CoScheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *17th Intl. Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2003.

[18] B. Franke and M. F. O'Boyle. A complete compiler approach to auto-parallelizing c programs for multi-dsp systems. *IEEE Trans. Parallel and Distributed Systems*, 16(3):234–245, Mar 2005.

[19] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on a commodity OS. In *5th Symp. Operating Systems Design and Implementation (OSDI)*, pages 165–180, Dec. 2002.

[20] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pages 120–32, May 1991.

[21] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *11th Intl. Symp. High-Performance Computer Architecture*, Feb. 2005.

[22] J. P. Jones and B. Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In D. G. Feitelson and L. Rudolph, editors, *5th Wkshp. Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1999.

[23] T. Jones, W. Tuel, L. Brenner, J. Fier, P. Caffrey, S. Dawson, R. Neely, R. Blackmore, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *15th IEEE/ACM Supercomputing*, Nov. 2003.

[24] K. Kennedy and R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, Oct 2001.

[25] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors. In *Wkshp. Complexity-Effective Design (WCED)*, June 2003.

[26] R. Love. *Linux Kernel Development*. Novell Press, Indianapolis, IN, USA, 2nd edition, 2005.

[27] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. pub-AW, Aug 2004.

[28] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *4th ACM Intl. Wkshp. Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Nov. 1993.

[29] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *16th ACM Symp. Operating Systems Principles (SOSP)*, pages 184–197, Oct. 1997.

[30] J. Nieh and M. S. Lam. Multimedia on multiprocessors: Where's the OS when you really need it? In *8th ACM Intl. Wkshp. Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 1998.

[31] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[32] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *15th IEEE/ACM Supercomputing*, Nov. 2003.

[33] M. Rosenblum, E. Bugnion, H. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *15th ACM Symp. Operating Systems Principles (SOSP)*, pages 285–298, Dec. 1995.

[34] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. Microsoft Press, 4th edition, Dec 2004.

[35] S. Rusu. Trends and challenges in high-performance microprocessor design. Presentation available from `www.eda.org/edps/edp04/submissions/presentationRusu.pdf`, Apr. 2004.

[36] B. Shneiderman. *Designing the User Interface*. Reading, MA, USA, 3rd edition, 1998.

[37] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *9th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, Nov. 2000.

[38] P. Terry, A. Shan, and P. Huttunen. Improving application performance on HPC systems with process synchronization. *Linux Journal*, (127):68–73, Nov. 2004.

[39] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *19th ACM Intl. Conf. Supercomputing*, pages 302–312, June 2005.

[40] D. Tsafrir and D. G. Feitelson. Barrier synchronization on a loaded SMP using two-phase waiting algorithms. In *16th Intl. Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2002.

[41] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *12th Intl. Conf. Parallel Architecture and Compiler Techniques (PACT)*, pages 26–34, Sept. 2003.

[42] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[43] H. Zheng and J. Nieh. SWAP: A scheduler with automatic process dependency detection. In *1st USENIX/ACM Symp. Networked Systems Design and Implementation (NSDI)*, pages 183–196, Mar. 2004.