Process Scheduling for the Parallel Desktop

Eitan Frachtenberg
Modeling, Algorithms, and Informatics Group (CCS-3)
Computer and Computational Sciences Division
Los Alamos National Laboratory
eitanf@lanl.gov

Abstract

Commodity hardware and software are growing increasingly more complex, with advances such as chip heterogeneity and specialization, deeper memory hierarchies, fine-grained power management, and most importantly, chip parallelism. Similarly, workloads are growing more concurrent and diverse. With this new complexity in hardware and software, process scheduling in the operating system (OS) becomes more challenging. Nevertheless, most commodity OS schedulers are based on design principles that are 30 years old. This disparity may soon lead to significant performance degradation. Most significantly, parallel architectures such as multicore chips require more than scalable OSs: parallel programs require parallel-aware scheduling.

This paper posits that imminent changes in hardware and software warrant reevaluating the scheduler's policies in the commodity OS. We discuss and demonstrate the main issues that the emerging parallel desktops are raising for the OS scheduler. We propose that a new approach to scheduling is required, applying and generalizing lessons from different domain-specific scheduling algorithms, and in particular, parallel job scheduling. Future architectures can also assist the OS by providing better information on process scheduling requirements.

1 Overview

Commodity computers and the way we use them changed qualitatively in the last 30 years. The OS, as the intermediary between hardware and software, is required to adapt to the changes in hardware, and the software that uses it. Indeed, many aspects of commodity OSs, such as networking and storage have changed concomitantly. Yet the scheduling policies that lie at the core of the OS remained virtually unchanged during this time [8, 23], and are in dire need of modernization. Several important trends in commodity architectures warrant this change, such as increased chip parallelism and heterogeneity, the emergence of power management as a limiting factor in desktop processors, and more complex memory hierarchies. Applications and workloads are alsc evolving to meet consumer demands and hardware capabilities, each application with its own scheduling requirements.

In this paper, we argue that the confluence of parallelism and diverse workloads on the commodity computer warrants revisiting the scheduler's policies. Three postulates support this thesis: commodity hardware is becoming increasingly parallel, the software running on it will follow suit, and the OS will need to support these trends, particularly in terms of scheduling. The main goal of this paper, in Sections 2-4, is to establish these premises and present the case and design goals for scheduler modernization. Next, Section 5 suggests some principles for the development of novel scheduling policies. Finally, we conclude in Section 6.

2 Hardware Trends: The Move to Concurrency

The phenomenal growth of microprocessor performance has fueled a similarly explosive growth in commodity hardware and applications for more than twc decades. However, the growth in single-processor performance is now showing signs of slowing down [25]. Even under optimistic assumptions for the growth rate of transistor density and processor speed, there is mounting evidence that the annual single-processor performance growth rate for the coming years might not exceed about 15%, contrasted with the rate of $50-60\%$ that we enjoyed so far [1, 21, 36].

To maintain high rates of performance growth, manufacturers are perforce turning to parallelism [18, 22, 26, 36]. As Hofstee writes on efficiency per transistor, "The most obvious way to improve

Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)
1087-4089/05 $20.00 © 2005 IEEE

efficiency is to sacrifice per-thread performance (or per-thread performance growth)" [22]. Even single-processor and desktop computers are employing varying degrees of parallelism, covering the range from superscalar processors, through hyperthreaded (SMT) and multicore chips (CMP), all the way to multiprocessors (SMP). All the major chip manufacturers already sell dual-core chips [26, 30, 32], and some special-purpose chips can even run tens of parallel hardware threads [24, 33].[1]

Parallelism in itself will not solve all the problems that are restricting performance growth. Limiting factors such as energy budget, cooling capacity, and memory performance will still require innovative design solutions such as heterogeneous cores with selective shutdown, use of specialized coprocessors, and moving computation closer to memory. Ubiquitous computers, such as mobile phones, portable music and video players, and media-convergence appliances that have strict minimum service requirements on a low-power, low-performance platform could further stress the resource management requirements. Memory hierarchies are also growing more complex, e.g., with multicore and hyperthreaded chips. Such computers are essentially nonuniform memory access (NUMA) machines, and as such, may impose special scheduling requirements [4]. Other emerging architectures include a relatively large number of special-purpose computing cores, such as the Cell processor for media applications, the ClearSpeed 96-core chip for mathematical processing, and the Azul 24-core chip for Java applications [22, 24, 33].

These architectures will require OS support to allocate their resources intelligently. The challenge for the OS does not end however with hardware abstraction and arbitration: modern applications could have significantly diverging scheduling requirements, as is described in the next section.

3 Software Trends: Interdependent Tasks

This section asserts that ubiquitous parallel hardware brings with it complex workloads with complex scheduling requirements, especially as software becomes increasingly more parallel.

Two factors have limited the availability of commodity parallel software to date. First, the desktop has remained largely uniprocessor until recently, thus delaying the main incentive to develop parallel programs. Second, writing parallel programs is difficult, and programming technology has not advanced to the point where parallel programming is commonplace. We have seen in the previous section that parallel commodity hardware is no longer an unrealized promise. The wide availability of parallel hardware could in turn provide an incentive for, and spark a growth in, parallel programming. With increasing demand, powerful new parallel languages and programming environments could become widespread. This paradigm shift has been compared to the 1990s' move to object-oriented design (OOD), also a technology that had existed for many years before becoming widely used [36]. The comparison permits us some optimism about this adoption. With increasing demand, powerful new parallel languages and programming environments could become widespread, as was the case with C++ and Java for OOD.

Already a typical uniprocessor desktop with a multitasking OS runs multithreaded applications, motivated by considerations such as resource overlapping, increased responsiveness, and modularity [13]. These applications range from the multithreaded Web browser to database and Web servers. Many of these applications have particular synchronization requirements, e.g., dependency on pipelined concurrent threads to read, process, and output data. While not critical on a uniprocessor, on a parallel architecture these requirements become paramount for efficiency.

The emerging parallel hardware creates a stronger incentive for concurrency. History shows that software makers eagerly respond to additional computing capacity by developing new, resource-hungry applications. Some contemporary applications already benefit from parallel computing power, for example, parallel searches in terabytes of data, photo and video editing filters, and technical computing in science and industry, such as automated design, scientific simulation, financial analysis, etc.

We may soon see several novel desktop applications that benefit from parallelism. For example, the combination of smart video recording and processing of multiple streams with parallel on-line local indexing and searching of vast media libraries (DVRs). User interfaces can be enhanced to include compute-intensive tasks that are executed concurrently with other user applications. Gaming, which has been a driving force in many new commodity technologies, could lead the pack with intensive and latency-sensitive parallel computations used to create com-

[1] Intel's Gelsinger predicts that 70% of performance improvements will come from parallelism rather than increased clock speed [18]. Already on the drawing board are multicore SMTs such as Niagara from Sun (2006), Power6 from IBM, and Tanglewood from Intel, with up to 32 hardware threads each, and possibly hundreds later on [24]. Even laptops are expected to gain dual-core and hyper-threading capabilities by 2006 [26].

Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)
1087-4089/05 $20.00 © 2005 IEEE

plex, realistic game environments[2]. In fact, all the next generation game consoles are designed for various levels of hardware concurrency.

At this point in time, we can only speculate on how much more parallel commodity workloads will become. However, if we accept as basic assumptions that consumer demand for applications that require more computing resources exists, and that the main performance growth in the future will come from parallelism, then it follows that commodity workloads will indeed grow more parallel.

As we see in the next section, concurrency and special synchronization requirements can yield poor performance if not properly supported by a parallel scheduler. Scheduling for applications with particular synchronization needs has been successfully studied in specific domains. However, unlike OSs on supercomputers and media players, the commodity OS cannot assume a relatively homogeneous workload. Heterogeneous, parallel, imbalanced, and dynamic workloads will impose conflicting requirements on the scheduler, increasing its importance in the future OS.

4 Challenges for the OS

Commodity OSs already run on small multiprocessors (SMPs) and SMTs, and scheduling is not considered a particularly dire problem. So if it is not broken, why fix it?

We reason that contemporary scheduling is indeed "broken," and that its inefficiency is currently masked by the low degree of parallelism and the user's own low expectations. But critical scheduling problems will surface as the degree of parallelism increases. Even today's simplest parallel machines, such as SMTs or small SMPs, already have difficulties with many applications and workload mixes. For example, processes that contend for the same resources (e.g., the memory bus) can experience an overall slowdown on a hyperthreaded system, rather than speedups resulting from parallelism and latency hiding [2, 4, 20].

Commodity schedulers are challenged at all levels of parallel execution, from the thread [4, 35], through the SMP [2, 12], the cluster [10, 15], all the way to supercomputers [23]. In particular, parallel programs suffer tremendously from lack of coscheduling[3] [10, 23]. The main reason for this is that processes in parallel programs—as opposed to sequential and distributed programs—rely on frequent synchronization for their progress. The only ways to ensure that such programs enjoy adequate progress are to let no other program share their processors or to ensure that when the parallel program runs, all of its synchronizing processes are coscheduled. Supercomputers typically use the former solution and operate in batch mode [11]. For commodity computers and workstations that host a multiuser, time-sharing system, this is not an acceptable solution [27]. A solution based on coscheduling, whether implicit or explicit, is far more attractive and practical. Applying coscheduling to the commodity OS is therefore imperative given the trends covered in the previous sections.

4.1 Coscheduling Example

To demonstrate the effect of coscheduling, we designed an experiment to isolate the effect of the scheduler policy on a small SMP. The experiment consists of a varying number of parallel programs and sequential stressors. Both types of programs perform a memory-less computation, but the sequential stressors run in an infinite loops, while the parallel programs run for a predetermined number of computation loops. The threads of the parallel programs synchronize every few compute loops using a semaphore.[4] All programs are launched together, and the experiment ends when all *parallel* programs terminate. We repeated each experiment a minimum of five times, discarded the lowest and highest results, and averaged the remaining run times. Two schedulers were evaluated: the default underlying OS scheduler (i.e., with no intervention on our part), and a rudimentary user-level gang scheduler [10] that suspends and resumes entire jobs in round robin-order. Note that memory hierarchy effects, I/O subsystems, and factors other than scheduling that can affect performance are excluded from this experiment. We predict that with parallel computers, memory bandwidth will become an even more critical resource than it is today, and should be managed by the scheduler [2, 4].

Figure 1 shows the results obtained on a four-way HP ES40 SMP with 833MHz Alpha processors, comparing the Linux default scheduler to gang scheduling. Observe that as the number of parallel programs increases, the total run time necessarily also increases for both schedulers. However, as the number of sequential stressors increases, the schedulers exhibit different behavior. With gang scheduling the run time remains nearly constant, since sequential jobs are confined to their time slots and do not in-

[2] The porting of several game engines to multicore processors is already underway (see for example `http://www.intel.com/pressroom/archive/releases/20050308net.htm`).

[3] Coscheduling refers to scheduling all of a job's processes at the same time, to facilitate synchronization [28].

[4] This structure follows the bulk-synchronous parallel (BSP) model, which in practice serves to capture the structure of many real parallel programs [37].

Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)
1087-4089/05 $20.00 © 2005 IEEE

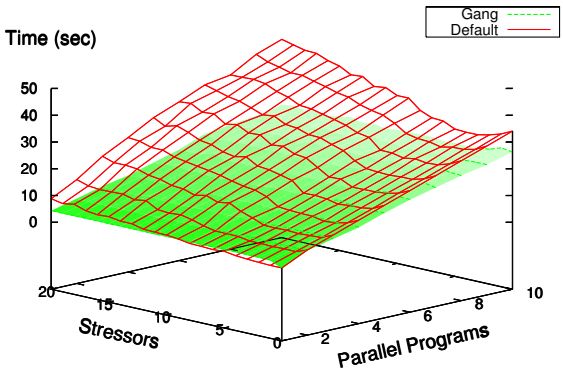

Figure 1: Scheduler comparison on ES40

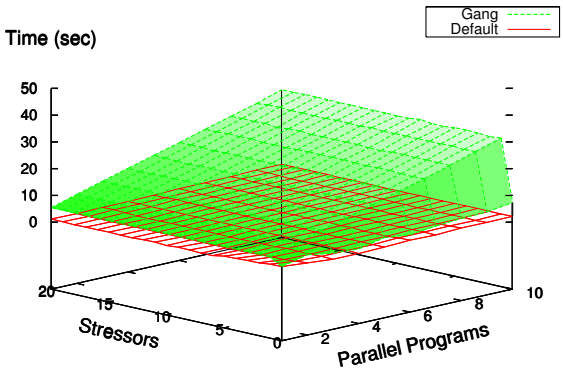

Figure 2: Scheduler comparison on Potomac

terfere with the parallel programs. When mixed together however, the Linux scheduler does not guarantee the coscheduling of parallel threads, resulting in frequent interruption to synchronization and wasted time spent blocking.

Is coscheduling always appropriate for parallel jobs? Not necessarily, as Figure 2 demonstrates. Here, we ran the same experiment on a four-way Intel Xeon MP 3.3GHz SMP ("Potomac" processors) with hyperthreading turned on (note that absolute run times are generally much shorter than on the ES40). While the OS may regard the virtual processors as independent processors, hyperthreaded logical processors do in fact share most of the chip's resources. Gang scheduling seven threads per parallel program on four actual processors results in self-competition among the threads. And since the threads are homogeneous, they often collide requesting the same CPU functional units [4]. Linux's rather oblivious scheduler outperforms in this example a coscheduling algorithm that is unaware of the architecture's subtleties. The reason is that under this oversubscribed gang scheduler, processes often block for lack of synchronization, but no other jobs can use the processor since they are currently suspended, resulting in significant waste.

4.2 Load-Imbalanced Workload

Even a workload that is entirely composed of parallel applications can be ill-suited for most commodity schedulers, especially if it exhibits load imbalance. Tc demonstrate this, we show a set of experiments that we had previously developed for the evaluation of a novel scheduling algorithm called Flexible Coscheduling (FCS) [16]. Although FCS was specifically designed to handle large-scale cluster scheduling, we believe that many of its design principles (some of which are discussed in the next section) are applicable to desktop parallel scheduling. In a nutshell, the main strength of FCS are its ability to dynamically identify the coscheduling requirement of each job and process, and schedule processes according to these requirements. Consequently, highly synchronous parallel jobs are allocated dedicated time slots so that their progress is not hindered. On the other hand, less synchronous jobs employ a variant of a spin-block algorithm in their communications, to reduce internal fragmentation.

The experiment we use to demonstrate these principles uses two dual Itanium nodes connected by the high-speed QsNet network. The workload consists of three synthetic applications that iterate computation and MPI synchronization, similar to the parallel BSP application of Section 4.1. All three programs have four communicating processes, with a basic granularity of $1ms$. However, we doubled the computation part of processes 3,4 in job 1 and processes 1,2 in job 2 to create a load imbalance, so that they take $120s$ to complete, compared to job's 3 $60s$. Note that the total CPU requirements of all three jobs is about $240s$ per processor. This workload is depicted in Fig. 3, where the basic iteration of each job is shown.

We compared five different synchronization schemes using the STORM infrastructure [17]: (1) Default—the unmodified Linux scheduler; (2) First-come-first-serve (FCFS), or batch scheduling; (3) gang scheduling (GS); (4) spin block (SB), which is a form of implicit coscheduling [3]; and (5) flexible coscheduling (FCS). We measure the finish time of each job separately. The maximum end time (in bold script) represents the makespan (the total

Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)
1087-4089/05 $20.00 © 2005 IEEE

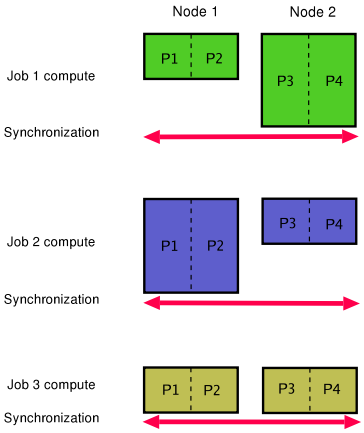

Figure 3: Imbalanced parallel workload

Scheduler	Job 1	Job 2	Job 3
Default	233	233	**284**
FCFS	120	240	**300**
GS	**303**	303	181
SB	210	211	**271**
FCS	248	**248**	151

Table 1: Imbalanced workload run times (sec). Numbers in bold represent total completion time.

completion time), which is a measure of how tightly a scheduler packs the jobs: the less fragmentation it incurs, the shorter the makespan. The results are summarized in Table 1.

Both FCFS and GS allow each program to run in dedicated mode on all four processes, whether for their entire lifetime (FCFS) or for the duration of a time slot at a time (GS). While job 3 enjoys this dedicated mode for its synchronization—and in fact requires it—jobs 1 and 2 end up wasting 25% of their alloted CPU time. Default and SB scheduling display better CPU utilization, since they are more prone to deschedule the idle processes of the imbalanced jobs, letting other processes use the CPU. However, they fail to coschedule job 3's processes effectively, resulting in poor synchronization and stalled processes. FCS avoids both these problems with the combination of monitoring, classification, and scheduling. First, it monitors MPI communication to measure the synchronization effectiveness and requirements of each process. After a short discovery period, it classifies job 3 as *CS* (requires coscheduling), and jobs 1 and 2 as *F* (prefer, but do not require coscheduling). This in turn translates to a schedule where job 3 always receives dedicated time slots (gang-scheduled), but job 1 and 2's processes can be descheduled if they stall too long. This results in an overall run time of just 8s over the optimal 240s schedule, or about 3%, compared to 10–30% with the other schedulers. A more detailed evaluation and analysis of FCS can be found in our previous reports [15, 16].

4.3 Composite Workload

Lastly, let us consider a different thought experiment using another small workload (this example may not necessarily be typical, but presents various potential problems compactly). Our workload is composed of three jobs on one dual-core chip: (1) a multi-threaded Web browser with high data locality, (2) a single-threaded music player with no data locality, and (3) a parallel computation with two fine-grain synchronous threads and medium data locality. Contemporary commodity schedulers, lacking awareness of parallelism and data locality, might commit any number of scheduling mistakes: they are not likely to coschedule the threads of job 3, leading to little progress on that job; they are prone to coschedule jobs 2 and 3, putting them in competition for memory access; and they might distribute job 1's threads over both processors, creating cache competition instead of cache sharing. Worse yet, schedulers such as those found in some Windows versions will allocate more time to the foreground job (e.g., the browser) even if it has the lowest CPU requirement. These mis-scheduling choices lead to extraneous context switches, wasted resources, and significantly reduced performance.

4.4 Generalizing the Challenges

Taking a more general view, we observe that parallelization poses two principal challenges to the desktop scheduler: (1) processes competing over resources suffer from degraded performance when coscheduled, and (2) collaborating processes suffer from degraded performance when *not* coscheduled. Architectural advances could also involve additional considerations in the future, such as heterogeneity/asymmetry, energy management, and NUMA, which need to be accounted for in the scheduling policies. But unlike classical parallel computers, the presence of various classes of applications in a single workload mix—including interactive and single-threaded applications—poses a significant additional challenge on top of the specific application requirements. Ignoring these scheduling considerations can lead to poor application performance because of lack of synchronization, as well as poor system-wide performance because of contention for resources [2, 10, 23, 35]. These factors paint a bleak picture for future OS per-

Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)
1087-4089/05 $20.00 © 2005 IEEE

formance unless they are accounted for in commodity multiprocessor schedulers. We propose initial guiding principles to this end in the next section.

5 The Road Ahead: Toward a Unified Solution

The challenges that schedulers face with the advent of ubiquitous parallelism and modern workloads add to the classical scheduling considerations. A considerable body of work has accumulated in specific domains, such as scheduling for soft-real-time systems [6, 8, 19], SMTs/SMPs [4, 34], and large-scale parallel machines [9, 11]. Some commercial OSs such as AIX, HP-UX, and IRIX already address some of these topics. We advocate a more holistic view of the scheduler as the reconciliator of coexisting synchronization requirements of all application types: sequential, interactive, continuous-media, distributed, and parallel. By generalizing ideas from these domains, we suggest a description of design goals for unified scheduling policies. More details on our proposed policies can be found in a separate technical report [14].

At the heart of these policies are data gathering and **classification** of processes (referring collectively to all entities of execution). The modern OS already engages in rudimentary instrumentation and classification for scheduling purposes—as is done for blocking I/O calls. Nevertheless, the complex hardware and workloads we face today allow and even require that we do more in terms of classification. For example, process communication and barriers (in and out of the node) can be used to classify processes based on their synchronization requirements [3, 5, 15], and pipelined or producer-consumer relationships can be traced though system-call usage [38]. Hardware counters can be used to measure processes' progress [31] and co-interference [34]. Recent trends in architectural design may also result in additional hardware support for process characterization.

To increase **cooperation** between processes, the scheduler can then use this classification to make better decisions about which processes require coscheduling (collaborative and codependent), require disjoint scheduling (interfering), or have special timing requirements (interactive).

Classification alone is unlikely to yield optimal scheduling. The scheduler can also benefit from dynamically testing different allocations of time and space, while monitoring the progress of processes [35]. Different combinations can have significant performance impact, especially if heterogeneity or energy management also come into play [4, 34].

In addition to classification, **adaptivity and automatic detection** of successful combinations can provide an intervention-free, self-tuning scheduler that perpetually attempts to optimize resource allocation and cost functions. The scheduler must also adapt to changes in the workload. Adaptivity can be expressed in various forms, such as choice of scheduling policies, parameters, process priorities, and timeslot frequency and length [19, 29, 34]. A unified scheduler should continue to draw from uniprocessor considerations (priorities, overlap of resources, etc.), and augment them in a scalable manner with considerations such as synchronization requirements, data locality, resource contention, and process progress. Furthermore, such a scheduler can be seamlessly adapted to cluster and grid environments if its considerations include out-of-box synchronization as well.

Going back to the composite workload example from Section 4.3, a unified scheduler might classify job 3 as parallel, given its communication pattern, and coschedule its processes to run in a dedicated time slot. Using classification or by testing different resource allocations it might determine that jobs 1 and 2 make the most progress when coscheduled together on different processors: job 1 enjoys high rates of cache sharing among its threads, while job 2 benefits from nearly-exclusive access to the memory bus. The scheduler can then adaptively change the timeslice quantum, to reflect different priorities, reduce noisiness, and enable interactivity.

Admittedly, such a unified scheduler is likely more complex than most of today's schedulers. Much of this complexity merely reflects the fact that the underlying hardware is becoming more complex. Furthermore, complex scheduling can greatly increase the symbiosis of processes and overall system performance [34], so it should not necessarily be regard as a disadvantage.

Explicitly addressing parallelism in the OS is an orthogonal discussion to any OS architectural layout (e.g. monolithic- vs. micro- vs. exo-kernel). The main role of any operating system, regardless of the design philosophy it follows, is to multiplex physical resources, including the CPU [7]. Scheduling thus remains at the core of the OS, and therefore all but the most specialized OSs can benefit from any advances made to better support contemporary hardware and workloads. The guidelines described here will probably evolve as prototypes are built. The key concept remains regardless of the underlying OS architecture: the scheduler's mission is to maximize cooperation between those that benefit from it, while minimizing interference between those that do not.

Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)
1087-4089/05 $20.00 © 2005 IEEE

6 Conclusion

As commodity computers and their workloads continue to evolve, the schedulers that manage them will grow increasingly inadequate. The aging schedulers' limitations attract little attention because they are good enough for single-processor computers. But with the move to parallel desktops, we can no longer afford to ignore these limitations. Applying parallel scheduling techniques alone to schedulers will also not suffice, because these techniques were developed for much more homogeneous workloads.

Some revitalization efforts have been concentrated on specific scheduling domains or on developing a new OS altogether. This paper advocates neither. Instead, it proposes that new holistic scheduling techniques be studied and integrated into existing commodity OSs. These policies should be dynamic and handle a wide range of workloads while remaining transparent to users.

Better scheduling is achieved when the OS has intimate understanding of the hardware's capabilities and the software's requirements. With regard to hardware, the OS should arbitrate between multiple and possibly heterogeneous resources, while considering cache and memory-pressure factors. With regard to applications, the OS needs to be cognizant of all levels of parallel execution: thread, process, and parallel program, in addition to sequential and interactive programs. Schedulers can manage these workloads by applying principles from such fields as parallel and multimedia scheduling. Of these principles, cooperation, adaptivity, and classification, in particular, can play a decisive role in achieving optimal user experience and utilization on next-generation computers.

Acknowledgements

I wish to thank Kei Davis, Yoav Etsion, Scott Pakin and Dan Tsafrir for many useful comments on early versions of this paper.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *27th International Symposium on Computer Architecture (ISCA)*, pages 248–259, June 2000. Available from `citeseer.ist.psu.edu/agarwal00clock.html`.

[2] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *32nd International Conference on Parallel Processing (ICPP)*, Kaohsiung, Taiwan, October 2003. Available from `www.cs.wm.edu/~dsn/papers/icpp03.pdf`.

[3] Andrea C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, August 2001. Available from `portal.acm.org/ft_gateway.cfm?id=380764&type=pdf`.

[4] James R. Bulpin and Ian A. Pratt. Multiprogramming performance of the Pentium 4 with hyper-threading. In *Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD)*, pages 53–62, Munchen, Germany, June 2004. Available from `www.ece.wisc.edu/~wddd/2004/06_bulpin.pdf`.

[5] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, Andy B. Yoo, and Chita R. Das. Coscheduling in clusters: is it a viable alternative? In *16th IEEE/ACM Supercomputing*, Pittsburgh, PA, November 2004. Available from `www.sc-conference.org/sc2004/schedule/pdfs/pap125.pdf`.

[6] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, Charleston, SC, December 1999. Available from `citeseer.ist.psu.edu/duda99borrowedvirtualtime.html`.

[7] Dawson. R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 78–85, Orcas Island, WA, May 1995. Available from `citeseer.ist.psu.edu/457928.html`.

[8] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Desktop scheduling: How can we know what the user wants? In *14th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 110–115, County Cork, Ireland, June 2004. Available from `www.cs.huji.ac.il/~feit/papers/HuCpri04NOSSDAV.pdf`.

[9] Dror G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994. Revised version August 1997 available from `www.cs.huji.ac.il/~feit/papers/SchedSurvey97TR.ps.gz`.

[10] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992. Available from `www.cs.huji.ac.il/~feit/papers/GangPerf92JPDC.ps.gz`.

[11] Dror G. Feitelson, Larry Rudolph, and Uwe Schwigelshohn. Parallel job scheduling – A status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2004. Available from `www.cs.huji.ac.il/~feit/parsched/`.

[12] Krisztián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism and interactive performance of desktop applications. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, number 9, pages 129–138, November 2000. Available from `www.eecs.umich.edu/~tnm/papers/asplos00.pdf`.

[13] Krisztián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism of desktop applications. In *Workshop on Multi-Threaded Execution, Architecture, and Compilers (MTEAC)*, Toulouse, France,

January 2000. Available from www-cse.ucsd.edu/users/tullsen/mteac2000/flautner.pdf.gz.

[14] Eitan Frachtenberg. Process coordination for commodity systems. Technical Report LAUR 04-7256, Los Alamos National Laboratory, December 2004. Available from www.cs.huji.ac.il/~etcs/pubs/.

[15] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003. Available from www.cs.huji.ac.il/~etcs/pubs/.

[16] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel and Distributed Systems*, To appear. Available from www.cs.huji.ac.il/~etcs/pubs/.

[17] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-fast resource management. In *14th IEEE/ACM Supercomputing*, Baltimore, MD, November 2002. Available from www.cs.huji.ac.il/~etcs/pubs/.

[18] W. Wayt Gibbs. A split at the core. *Scientific American*, 291(5):96–101, November 2004. Available from www.sciam.com/article.cfm?articleID=00026625-6DF0-1179-ADF083414B7FFE9F.

[19] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity OS. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, Boston, MA, December 2002. Available from citeseer.ist.psu.edu/goel02supporting.html.

[20] Anoop Gupta, Andrew Tucker, and Shigeru 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, San Diego, CA, May 1991. Available from discolab.rutgers.edu/classes/cs519-old/papers/p120-gupta.pdf.

[21] Paul Hales. Intel's Grove warns of the end of Moore's Law. www.theinquirer.net/?article=6677, December 2002.

[22] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, February 2005. Available from www.hpcaconf.org/hpca11/papers/25_hofstee-cellprocessor_final.pdf.

[23] Terry Jones, William Tuel, Larry Brenner, Jeff Fier, Patrick Caffrey, Shawn Dawson, Rob Neely, Robert Blackmore, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *15th IEEE/ACM Supercomputing*, Phoenix, AZ, November 2003. ACM Press and IEEE Computer Society Press. Available from www.sc-conference.org/sc2003/paperpdfs/pap136.pdf.

[24] Michael Kanellos. Designer puts 96 cores on single chip. news.com.com/2100-1006_3-5399128.html, October 2004.

[25] Michael Kanellos. Intel kills plans for 4GHz Pentium. news.com.com/2100-1006_3-5409816.html, October 2004.

[26] Rakesh Kumar, Keith Farkas, Norman Jouppi, Partha Ranganathan, and Dean Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors. In *Workshop on Complexity-Effective Design (WCED)*, June 2003. Available from citeseer.ist.psu.edu/kumar03multicore.html.

[27] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Fourth ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, November 1993. Available from citeseer.ist.psu.edu/443381.html.

[28] John. K. Ousterhout. Scheduling techniques for concurrent systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, Miami, FL, October 1982.

[29] Calton Pu and Robert M. Fuhrer. Feedback-based scheduling: A toolbox approach. In IEEE, editor, *Fourth IEEE Workshop on Workstation Operating Systems (WWOS-IV)*, pages 124–128, Napa, CA, October 1993. IEEE Computer Society Press. Available from ieeexplore.ieee.org/iel2/918/8054/00348177.pdf.

[30] Stefan Rusu. Trends and challenges in high-performance microprocessor design. Presentation available from www.eda.org/edps/edp04/submissions/presentationRusu.pdf, April 2004.

[31] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting operating systems. In *Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 124–129, Cape Cod, MA, May 1997. Available from www.eecs.harvard.edu/vino/vino/papers/monitor.ps.

[32] Stephen Shankland. Intel's dual-core Xeon due in 2006. news.com.com/2100-1006_3-5416330.html, October 2004.

[33] Stephen Shankland. Azul's first-generation Java servers go on sale. news.com.com/2100-1010_3-5673193.html?tag=nl, April 2005.

[34] Allan Snavely, Dean Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pages 66–76, Marina Del Rey, CA, June 2002. Available from citeseer.ist.psu.edu/528307.html.

[35] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, Cambridge, MA, November 2000. Available from citeseer.ist.psu.edu/338334.html.

[36] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. Available from www.gotw.ca/publications/concurrency-ddj.htm.

[37] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. Available from portal.acm.org/citation.cfm?id=79181&dl=ACM&coll=portal.

[38] Haoqiang Zheng and Jason Nieh. SWAP: A scheduler with automatic process dependency detection. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, pages 183–196, San Francisco, CA, March 2004. Available from www.ncl.cs.columbia.edu/publications/nsdi2004_fordist.pdf.