USENIX Association

Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services

San Francisco, CA, USA
May 5-8, 2003

Operating System Modifications for Task-Based Speed and Voltage Scheduling

Jacob R. Lorch
Microsoft Research
1 Microsoft Way
Redmond, WA 98052
lorch@microsoft.com

Alan Jay Smith
University of California, Berkeley
EECS Department, Computer Science Division
Berkeley, CA 94720-1776
smith@eecs.berkeley.edu

Abstract

This paper describes RightSpeed, a task-based speed and voltage scheduler for Windows 2000. It takes advantage of the ability of certain processors, such as those from Transmeta and AMD, to dynamically change speed and voltage and thus to save energy while running more slowly. RightSpeed uses PACE, an algorithm that computes the most energy efficient way to meet task deadlines with high probability. Since most applications do not provide enough data about tasks, such as task deadlines, for PACE to work, RightSpeed uses simple and efficient heuristics to automatically detect task characteristics for such applications. RightSpeed has only 1.2% background overhead and its operations take only a few microseconds each. It even performs PACE calculation, which is quite complicated, in only 4.4 μs on average due to our extensive optimizations. RightSpeed is effective at meeting performance targets set by applications to within 1.5%. Although the PACE calculator does not save energy for the current generation of processors due to their limited range of worthwhile speed and voltage settings, we expect future processors to have greater such ranges, enabling PACE to reduce CPU energy consumption by 6.1–8.7% relative to the best standard algorithm. Furthermore, with PACE, giving a processor the ability to run at additional, higher speeds and voltages *reduces* overall energy consumption.

1 Introduction

Reducing energy consumption is important in portable computers due to their limited battery capacity. Furthermore, rising concerns about energy prices and aggregate energy dissipation in server farms make energy management important for other computers as well. An energy-saving technology that has recently begun appearing in modern portable computers is dynamic voltage scaling (DVS), the ability to change processor voltage without rebooting. This enables reduced energy consumption, as lower voltages mean lower energy consumption. Lower voltages, however, necessitate lower CPU speeds, presenting an interesting operating system issue: how to ensure that performance remains reasonable while sometimes lowering speed to save energy.

Traditionally, systems use *interval-based* strategies. Such strategies divide time into intervals of fixed length and set the speed for each interval based on recent CPU utilization. However, recent CPU utilization is only a rough indicator of the required speed. An interval-based strategy cannot distinguish an urgent task that must run at full speed to meet a tight deadline from a less important task with several milliseconds to complete and little work to do.

A better solution, as suggested by authors such as Pering et al. [18] and Hong et al. [6], is to use *task-based scheduling*. Such scheduling considers the computer's work to consist of tasks with certain CPU requirements and deadlines. It then runs the CPU fast enough to meet those deadlines with reasonable probability. Recently, some researchers have even built such task-based schedulers [19, 4, 3]. In this paper, we describe how we built RightSpeed, a task-based scheduler with several improvements over these existing schedulers.

The key differentiating feature of RightSpeed is its *PACE calculator*, a component that determines the most energy efficient schedule for meeting each task's performance requirements. PACE stands for Processor Acceleration for Conserving Energy, since the optimal way to schedule a task is to start out slowly, increasing speed only as necessary to complete the task on time. In [11], we showed that computing such a schedule requires estimating the probability distribution of the task's CPU requirement, and gave a method called PACE that uses such a distribution to compute such a schedule. For this paper, we extended this method substantially to deal with issues that arise in real systems: limited available speed/voltage settings, nonlinear relationship between speed squared and energy, limited timer granularity, and I/O wait time.

RightSpeed also differs from other schedulers in the heuristic it uses for *automatic task detection*. A task-based scheduler can provide an interface letting applications specify information about their tasks. However, many application writers will not use it, so a task-based scheduler should also have an automatic task detector to let it infer task information from such applications. The schedulers Flautner et al. describe in [4] and [3] have such detectors, but they require a great deal of complex, high-overhead, and Linux-specific system interposition. In [12], we suggested a method for automatic task detection with a more efficient heuristic, but did not demonstrate an implementation. RightSpeed demonstrates an implementation of our heuristic.

Our scheduler also differs from existing schedulers by running on Windows 2000 rather than Linux. This is important because most portable computers sold today run Windows 2000 or its successor Windows XP. Our work demonstrates that task-based scheduling can be done even on a closed-source commodity operating system.

The goal of this paper is to demonstrate that a task-based scheduler with a PACE calculator and an automatic task detector can be implemented on a real machine running Windows 2000. This involves overcoming the challenges of real hardware and software issues, and demonstrating that the resulting scheduler places little overhead on the system.

The structure of this paper is as follows. Section 2 gives background and related work on DVS algorithms. Section 3 describes the characteristics of the processors to which we ported RightSpeed, and evaluates the potential effectiveness of DVS techniques on these processors. Section 4 discusses the design of our task-based scheduler, and Section 5 describes our implementation of it. Section 6 gives results of benchmarks showing the impact of our modifications on performance and energy consumption. Section 7 discusses avenues for future work. Finally, Section 8 concludes.

2 Background and Related Work

2.1 Dynamic voltage scaling

In CMOS circuits, the dominant component of power consumption is proportional to $V^2 f$, where V is voltage and f is frequency. Energy is power times time, and the time to run a certain number of cycles is inversely proportional to frequency, so energy per cycle is proportional to V^2 [22, p. 235]. At a given voltage, the maximum frequency at which the CPU can run safely decreases with decreasing voltage. Thus, the system can reduce processor energy consumption by reducing CPU voltage, but this necessitates running at a slower speed.

However, it is important to not noticeably increase system response time, for two reasons. First, other components, such as the disk drive and backlight, use power. Noticeably increasing response time may cause these components to remain in high-power modes longer than they otherwise would, which can more than offset processor energy savings. Second, the user will object to unduly extended response times.

2.2 Interval-based DVS algorithms

The first researchers to discuss operating system techniques for DVS were Weiser et al. [21] and Chan et al. [2]. They suggested an interval-based approach, meaning that the system divides time into fixed-length intervals and schedules the speed for each interval based on the CPU utilizations of past intervals.

Interval-based strategies are used today in real systems capable of dynamic voltage scaling, such as Transmeta's LongRun$^{\text{TM}}$ [7]. However, such strategies have problems, as Pering et al. [17], and later Grunwald et al. [5], pointed out. The CPU utilization by itself does not provide enough information about system timing requirements to ensure meeting a reasonable number of deadlines while saving energy.

2.3 Task-based voltage schedulers

Recently, researchers have started building task-based schedulers, i.e., schedulers that consider the work of the system to consist of tasks with certain deadlines. The goal of a task-based scheduler is to use speeds just high enough to meet these deadlines with reasonable probability.

Yao et al. [23] described how to compute an optimal schedule when task CPU requirements and deadlines are known. Hong et al. [6] later showed how to compute such schedules more quickly using various heuristics. However, systems do not generally have definite knowledge of task CPU requirements, so these approaches are unrealistic.

Flautner et al. [4] built a task-based voltage scheduler for Linux. This scheduler requires no modification of applications—it infers all information about the system's tasks via heuristics. It infers that an interactive task begins when a user interface event arrives, and uses a complex work-tracking heuristic to decide when such a task completes. It infers that a periodic task begins when a periodic event occurs; it considers an event periodic if the lengths of intervals between the last n events have a small variance. To determine the speed for a task, it essentially computes the average of the speeds that would have completed past similar tasks on time. In later work [3], they refined their period detector (but not their task completion detector) to use a simpler heuristic and they extended their interactive performance-setting algorithm with two other policy layers: one for application-specific policies and one for a per-task interval-based policy.

Pillai et al. [19] built a task-based scheduler for real-time embedded systems that runs on Linux. This scheduler assumes complete knowledge of the deadlines and worst-case CPU requirements of all tasks in the system, and assumes these tasks are periodic. The scheduler uses different algorithms, some of which make provisions for tasks completing before their deadlines. One such algorithm slows down the CPU when a task creates slack in the schedule by completing early. Another algorithm anticipates that tasks will likely complete early and therefore starts tasks as slowly as possible and only uses higher speeds when these become necessary to guarantee on-time completion.

2.4 PACE

One premise of task-based scheduling is that DVS can exploit deadlines to save energy without significantly reducing performance. This is possible since a task's completion time is irrelevant as long as it completes before the deadline. Thus, in evaluating the performance of a DVS algorithm, we can consider all tasks that complete by the deadline to have the same effective performance.

A DVS algorithm essentially chooses a schedule describing how speed will vary with time. In [11], we showed that two schedules that have the same average pre-deadline speed and identical post-deadline parts will give the same effective performance no matter how much work a task requires. This means that one can get the same performance as any existing DVS algorithm by using different, yet performance equivalent, speed schedules; these new schedules may even consume less energy.

We then described an algorithm, PACE, for choosing a speed schedule that minimizes expected energy consumption for a given performance constraint. The PACE algorithm assumes some knowledge of task CPU requirement distribution; we showed how to dynamically and effectively estimate this distribution. One limitation is that PACE assumes the processor speed and voltage are continuously variable and that energy is a linear function of speed squared; in this work, we extend PACE to real DVS systems without these properties.

PACE requires the ability to detect when tasks begin and end. In [12], we showed that there is a simple heuristic for inferring task completion that is nearly as effective as Flautner et al.'s scheduler [4] and requires substantially less operating system modification. Our approach considers a task complete when either all threads in the system are blocked and no I/O is ongoing, or when a new user interface event is delivered to the same application.

In [12], we pointed out that user interface events belonging to different types, categories, and applications differ significantly from each other. This difference is large enough that PACE benefits, rather than worsens, by inferring the probability distribution of a task from a sample of only those recent past tasks that have nearly identical characteristics. Therefore, in RightSpeed, we keep separate samples for tasks triggered by user interface events of different types, categories, and applications.

3 Platforms

In this section, we examine the characteristics of Transmeta and AMD processors to which we ported RightSpeed. As we do so, we will discuss how these characteristics influence how we should use PACE on these processors.

First, we introduce some definitions. A *setting* is a speed and voltage combination at which a processor can properly operate. The *efficiency* of a setting is the amount by which power consumption is reduced by using this setting instead of emulating its speed using the best possible combination of all other settings. For example, suppose there are three settings: 300 MHz consuming 2 W, 500 MHz consuming 3.6 W, and 700 MHz consuming 6 W. We can emulate 500 MHz by running half the time at 300 MHz and half the time at 700 MHz. This consumes 4 W, while the 500 MHz setting consumes only 3.6 W, so the 500 MHz setting has efficiency 10%. We can emulate 300 MHz by running 60% of the time at 500 MHz and turning the CPU off 40% of the time; this emulation has average power consumption 2.16 W, so the 300 MHz setting has efficiency 7.4%. If a setting has efficiency of 0% or less, it is not *worthwhile*, i.e., one should never use it since one can get lower power consumption at the same speed using other settings.

For PACE to be effective, a processor must have at least three worthwhile speed/voltage settings. Furthermore, the more settings, and the higher their efficiency, the more effective PACE will be. This is because PACE works by choosing among speed schedules with identical performance to find the one with least expected energy consumption. If there is little choice in such speed schedules, and/or if there is little difference between choosing one setting versus emulating that setting's speed with other settings, there will likely be little benefit to choosing among them.

Speed	Voltage	Power	Energy/ cycle	Efficiency
297.3 MHz	1.2 V	1.349 W	4.537 nJ	0.5%
396.6 MHz	1.225 V	1.809 W	4.561 nJ	11.0%
497.8 MHz	1.35 V	2.714 W	5.461 nJ	11.8%
598.5 MHz	1.55 V	4.348 W	7.265 nJ	0.4%
631.1 MHz	1.6 V	4.915 W	7.787 nJ	N/A

Table 1: Characteristics of the Transmeta processor at various settings

Speed	Voltage	Power (est.)	Energy/ cyc (est.)	Efficiency (est.)
500 MHz	1.25 V	10.6 W	21.3 nJ	7.6%
600 MHz	1.3 V	13.8 W	23.0 nJ	1.4%
700 MHz	1.35 V	17.4 W	24.8 nJ	-0.9%
800 MHz	1.4 V	21.3 W	26.7 nJ	-3.6%
900 MHz	1.4 V	24.0 W	26.7 nJ	N/A

Table 2: Characteristics of the AMD processor at various settings, with power and energy values approximated

3.1 Transmeta system

Our Transmeta system contains a TM5400-633 Crusoe$^{\mathrm{TM}}$ processor and 128 MB of memory (64 MB of SDRAM and 64 MB of DDRAM). 16 MB of this memory is reserved for the Code-Morphing Software, whose primary function is to dynamically translate x86 code to the underlying machine language of the VLIW chip. This code also implements LongRun$^{\mathrm{TM}}$, the DVS policy Transmeta chips use. Transmeta told us how to override LongRun$^{\mathrm{TM}}$ policies and change the speed ourselves.

The processor can run at 300–633 MHz and 1.2–1.6 V. Table 1 gives the available speeds and voltages, as well as the power the CPU consumes at each level. We measured power consumption by running a tight loop of additions while using hardware monitoring equipment Transmeta provided.

We see that the 300 MHz and 600 MHz settings have very low efficiencies, and are therefore barely worthwhile. With only three reasonably worthwhile settings, we do not expect PACE to be very effective on this machine.

Incidentally, we note that the formula $1.179 \cdot 10^{-9} \cdot s^{3.41} + 3.681$, where s is speed, gives a very close approximation to the energy consumption in nJ/cycle for all but the 300 MHz setting. The power of 3.41 differs substantially from the power 2 predicted by simple scaling models, e.g., in [21].

3.2 AMD system

Our AMD system contains a pre-production version of the 900 MHz Mobile Athlon 4 processor, based on the Palomino core, as well as 128 MB of memory. We were given documentation about PowerNow!$^{\mathrm{TM}}$, the interface the chip uses for dynamically changing speed and voltage.

The chip indicates it is capable of five settings, shown in Table 2. We were unable to directly determine the power consumption of each setting since we lacked the necessary measurement equipment, so we estimate it using $P \propto V^2 f$. We assume a power consumption of

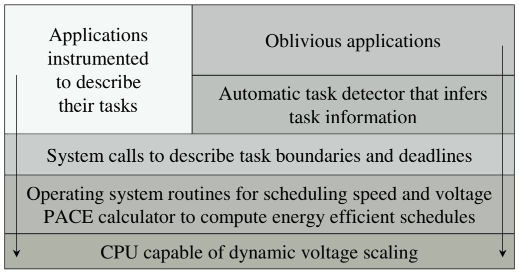
Applications instrumented to describe their tasks
Oblivious applications
Automatic task detector that infers task information
System calls to describe task boundaries and deadlines
Operating system routines for scheduling speed and voltage
PACE calculator to compute energy efficient schedules
CPU capable of dynamic voltage scaling

Figure 1: Overview of RightSpeed

24 W at the maximum speed, as specified in the AMD data sheet [1].

We see that the 700 MHz and 800 MHz settings have negative efficiency, so they are not worthwhile. (It is not surprising that the 800 MHz setting is not worthwhile, since it has the same voltage as the 900 MHz setting, and thus the same energy consumption, but it runs more slowly.) Furthermore, the 600 MHz setting has rather low efficiency. With only three worthwhile settings, one of which is only barely worthwhile, we expect PACE to be largely ineffective.

We suspect that some settings have poor efficiency because AMD made overly conservative choices of maximum stable speed for certain voltages. One reason for this is that their current design requires processor speeds and voltages to attain only a certain set of values. More flexibility in either dimension would let them choose settings closer to the curve of maximum ideal efficiency.

4 Design

4.1 Overview

Figure 1 gives an overview of the RightSpeed design. Applications convey information about their tasks to the operating system using system calls. This information includes when tasks begin and end and what performance targets the application wants for those tasks. Some applications are oblivious to the existence of these system calls, so an automatic task detector infers task information about them and generates task specification system calls on their behalf. The system uses information about ongoing tasks to determine what speed to use at various times, and implements this schedule using timers and special processor instructions that change speed and voltage. The system uses a PACE calculator to compute the most energy efficient schedules that have the performance requested.

In addition to the above functionality, we had three overall goals for RightSpeed. First, we wanted it to be efficient, creating low overhead on the system both when running in the background and when actively invoked. Second, we wanted it to be stable, relying only on documented system interfaces so that it would run even when the operating system was upgraded. Third, we wanted it to be easily portable to different processors despite such processors having different commands dealing with speed and voltage settings.

4.2 Task specification interface

A key piece of information an application must specify about a task is its *type*. An application may define types any way it chooses; there are two reasons applications will want to classify different tasks into different types. First, it may want to specify different performance targets for different types of tasks. For example, an MPEG player may require a faster speed for processing its I-frames than its smaller P-frames. As another example, it may want a short and hard deadline for its frame playback tasks but a longer and soft deadline for its user interface tasks. Second, tasks of different types may have different CPU requirement distributions, so it is helpful to direct PACE to only consider tasks of the same type when estimating the probability distribution of a task's CPU requirement.

RightSpeed uses this notion of task type to simplify its communication with applications. When an application begins a task, it need only tell RightSpeed the type of that task. RightSpeed can figure out all other information about the task, such as its performance requirements, from that type. RightSpeed can give the application a unique identifier to identify this task, so the application can specify when the task completes by merely passing RightSpeed that identifier. RightSpeed can then determine how many CPU cycles that task used and use this datum to compute a new optimal PACE schedule for the next task of that type.

An application specifies performance targets for task types via a separate part of the task specification interface. An application need only specify this data once, when it is installed. Because task type data is persistent, i.e., it is retained even when the application terminates and even when the system shuts down, a logical abstraction to use for this data is a file. Thus, applications create files containing data for their task types.

An application may specify a performance target in two ways. First, it may specify a number of CPU cycles to be completed by a certain deadline. Second, it may specify a deadline and a particular DVS algorithm, such as Transmeta's LongRun^{TM}, and dictate that performance be the same as would be achieved via that algorithm.

4.3 Automatic task detector

Since RightSpeed has not been released, no application currently exists that explicitly communicates its task information to RightSpeed. Furthermore, even when it is released, we expect few application writers will be both willing and able to communicate such information. Therefore, for RightSpeed to be useful, we require an automatic task detector to infer task information from such applications and to call the task specification interface on their behalf.

Our approach focuses on the tasks the user cares about most: those triggered by user interface events. User interface studies have shown that response times under 50–100 ms do not affect user think time [20]; we thus consider 50 ms the soft deadline for handling a user interface event. An exception is mouse movements, whose tracking may require response times of only 25–50 ms [13]; we thus consider 25 ms the soft deadline for handling them.

We consider a task to begin when an application receives a user interface event. We classify tasks into types, and deduce the task type from the event characteristics, i.e., whether it is a keystroke, mouse movement, or mouse click; which key or mouse button was pressed or released; and to what application the event was delivered. As shown in [12], separating tasks into types this way makes estimation of task work distribution more accurate, and enables us to set different policies for, for instance, keystrokes and mouse clicks.

As suggested in [12], we use the minimum speed available as the pre-deadline speed for mouse movement events. Such events require little processing, so this is sufficient to meet practically all task deadlines. We use a default pre-deadline speed of $0.7M$ for keystroke events and $0.85M$ for mouse click events, where M is the maximum speed available on the machine. A better approach might be to compute a variable pre-deadline speed based on the distribution observed and the likelihood of missing deadlines at various pre-deadline speeds, as suggested in [12]. Unfortunately, this requires accurate estimation of the tails of nonstationary distributions, and we do not yet know how to do this; this is future work.

We also need a heuristic to determine when such an inferred task is complete, since it is difficult to determine what CPU activity belongs to a given task. We use the heuristic from [12] described in Section 2.4: we consider a task complete when either (a) all threads in the system above the idle priority level are blocked and no I/O is ongoing, or (b) another user interface event is delivered to the same application. An advantageous side effect of this is that time spent by unrelated threads is considered part of the task. Thus, the speed schedule chosen will automatically account for the work performed by other threads during the task. Without this accounting, the presence of such unrelated activity could interfere with RightSpeed meeting its target deadlines.

4.4 PACE calculator

Computing the optimal speed schedule satisfying certain performance constraints requires knowledge of task CPU use distribution, which typically an application lacks. RightSpeed keeps track of how long tasks of each type have taken, and uses this information to compute such an optimal speed schedule with PACE.

In [11], we described how to compute an optimal schedule assuming a linear relationship between energy and speed squared. Since the processors on which RightSpeed runs do not satisfy this property, we developed a more general formula that does not rely on it. We discovered that the optimal speed schedule satisfies $s^2\mathsf{E}'(s)F^c(w) = K$, where s is the speed to run after completing w cycles of a task, $F^c(w)$ is the probability the task takes more than w cycles, $\mathsf{E}(s)$ is the energy consumption at speed s, and K is a constant chosen to satisfy the performance constraint. For more details about this formula and a proof that it works, see [9, pp. 83–99].

In [11], we assumed that the CPU had arbitrarily variable speed settings that could be changed at arbitrary times. Our real systems have only a limited number of speed settings, and Windows 2000 only allows us to change speed at certain fixed times, once per millisecond. Thus, for RightSpeed we need an algorithm that takes these realities into account yet still computes a near-optimal schedule. Our algorithm uses the following four steps.

1. Create an idealized schedule using the formula above. Apply the granularization techniques of [11] to get a schedule consisting of consecutive *phases*, each having a constant speed.
2. For each phase, round its speed to the closest speed that is available on the CPU and worthwhile.
3. Round the length of each phase to an integer multiple of the scheduling granularity.
4. As the rounding may have altered the schedule's performance characteristics, i.e., changed the pre-deadline speed, adjust the time spent at each speed by multiples of the scheduling granularity to make performance close to, but no less than, requested performance.

As an optimization, we precompute a set of parameterized speed schedules when RightSpeed is installed, based solely on the CPU characteristics. Thus, determining a speed schedule involves only a binary search through the schedules to find the lowest-energy one that nevertheless satisfies the constraint. With this optimization, the algorithm takes time $O(n)$ where n is the number of worthwhile speed settings. For details of this and other optimizations, see [9, pp. 224–226] and the code at the website associated with this paper.

4.5 Dealing with I/O

I/O time, unlike CPU time, is unaffected by changes in CPU speed. The model from which PACE arises accounts only for task CPU time, so PACE does not give optimal results when I/O can occur. Essentially, the occurrence of I/O will delay the completion of a task, possibly causing it to miss its deadline.

We deal with this in the following way. Since the problem is to complete the CPU work *and* the I/O by the deadline, we must complete the CPU work within a period equal to the deadline minus the I/O time. If we knew I/O time in advance, PACE could compute the optimal schedule merely by substituting the deadline minus I/O time for the deadline. Since we do not know I/O time in advance, we initially assume it is 0. If I/O occurs later, we determine how long it took and accelerate the schedule to make up for the lost time.

Theoretically, accelerating the schedule properly requires performing a new complex calculation using the PACE formula. However, we can use a shortcut: we multiply all speeds in the schedule by a constant factor, where we choose that factor such that after rounding all resulting speeds to the nearest worthwhile speed we get a schedule that meets the new deadline constraint. The argument why this works is as follows: The distribution of task work remaining has by assumption not changed, but the deadline has effectively gotten shorter. Thus, all that has changed is the optimal value of K. This means the ratio of the new optimal speed to the old optimal speed is roughly the same for all points in the schedule, assuming that the function of energy versus speed has a reasonable shape.

4.6 Scheduling simultaneous tasks

When multiple tasks are ongoing, the ideal speed is not necessarily the sum of all the speeds for all those tasks. This is because power is not a linear function of speed, so superimposing schedules consumes a different amount of energy than running them sequentially. Unfortunately, computing a reasonable speed schedule that is the conjunction of two is extremely complex, so we avoid the issue by simply running at the maximum speed available when there are multiple tasks, and continue at that speed until no tasks remain.

Fortunately, in a mobile computer (and frequently in a desktop computer) there is only a single user and typically he will only notice the performance of the task with which he is currently actively involved. Therefore, typically there will be only one ongoing task at a time. Evidence supporting this comes from workload analyses we performed in [12] on months-long traces of eight desktop computers. We found that, depending on the user, between 94.7 and 99.3% of all user interface tasks finished before the next one began.

4.7 Scheduling no ongoing tasks

When no tasks are ongoing, nothing of importance is occurring, so the best speed to use is generally the minimum available. However, since our inference of tasks is imperfect, there may be ongoing tasks even when RightSpeed believes there are no such tasks. For instance, a task may have been triggered by a timeout instead of by a user interface event. We deal with this by reverting to a traditional interval-based scheduler when we know of no ongoing tasks. Such a scheduler divides time into intervals of some fixed length and chooses a speed for each interval based on the CPU utilization of recent past intervals. This way, if the CPU becomes busy from working on a task we cannot detect, the interval-based scheduler will nevertheless increase speed to deal with this unknown work.

One caveat is that when the number of known tasks becomes zero, recent past CPU utilization will likely be high because the system just finished working on a task. RightSpeed knows that this recent utilization is a poor predictor of future CPU utilization because it reflects a task that is no longer active. However, an interval-based scheduler has no knowledge of tasks, so it will interpret the high recent utilization as a sign that the next intervals will have high utilization. Accordingly, it will use an unnecessarily high CPU speed. To prevent this problem, when the number of known tasks becomes zero, RightSpeed waits for a short period of time at the minimum CPU speed before initiating the interval-based scheduler.

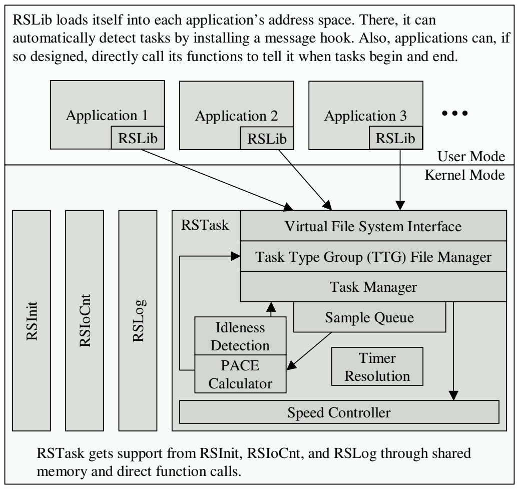
RSLib loads itself into each application's address space. There, it can automatically detect tasks by installing a message hook. Also, applications can, if so designed, directly call its functions to tell it when tasks begin and end.
Application 1
RSLib
Application 2
RSLib
Application 3
RSLib
...
User Mode
Kernel Mode
RSInit
RSIoCnt
RSLog
RSTask
Virtual File System Interface
Task Type Group (TTG) File Manager
Task Manager
Sample Queue
Idleness Detection
PACE Calculator
Timer Resolution
Speed Controller
RSTask gets support from RSInit, RSIoCnt, and RSLog through shared memory and direct function calls.

Figure 2: Architecture of RightSpeed

5 Implementation

In this section, we discuss how we implemented our approach on Windows 2000.

5.1 Architecture

Figure 2 shows the architecture of RightSpeed. The main component is RSTask, a kernel module that receives requests to begin and end tasks and schedules the CPU speed accordingly. Its main components are the speed controller, the task type group file manager, the automatic schedule computer, and the idleness detector, each of which we will discuss later. Alongside RSTask is RSIoCnt, a kernel module that interposes on all file system requests to monitor when any synchronous I/O's are ongoing. The next kernel component is RSLog, a low-overhead logger we use for benchmarking and debugging. The last kernel mode component is RSInit, a driver that starts before all other drivers and facilitates communication between them. In user mode we have RSLib, a user-level library that the system loads into the address space of every application. It interacts with the GUI to interpose the user interface event delivery system and thereby implement the automatic task detector. This library also exports functions that applications can use to communicate with RightSpeed.

5.2 Speed controller

The lowest-level component of RSTask is the speed controller. This component accepts requests to start and stop speed schedules and to transition to idle and maximum speed states. A speed schedule consists of a sequence of phases, each with a speed to use and a duration in multiples of the scheduling granularity. The speed controller internally handles any CPU-specific commands to change the speed. This modularity aided in porting RightSpeed to two different chips with different voltage scaling commands.

The scheduler also exports routines to pause and resume the current schedule when the CPU starts and stops waiting for I/O. A pause changes the speed to the minimum available. A resume determines how long the CPU spent waiting for I/O, accelerates the remaining part of the schedule accordingly, and resumes that schedule.

5.3 Timer resolution controller

The default timer resolution on Windows 2000 machines is about 10 ms. Our timer resolution controller reduces the timer resolution as much as possible using well-documented system calls [16]. On the systems we used, this makes timer resolution, and thus scheduling granularity, 1 ms.

5.4 Task type group file manager

Certain persistent information is associated with each task type: its deadline, its performance target, a sample of recent task CPU requirements, and a schedule to use for the next task. Thus, it makes sense to consider task types to be part of a virtual file system. We could have used one file per task type, but instead a file in this virtual file system is a *task type group file*, containing information about multiple related task types. A task type is uniquely identified by its file and its index within that file.

RSTask thus exposes a virtual file system interface consisting of these files. RSTask stores the information in these virtual files in real files in a reserved directory on the real file system, but RSTask exposes them as existing in the special directory \\.\RSTask. (The Unix analog would be /proc/rstask/.) Subdirectories of this directory are valid and supported; for instance, an application could choose to use a file called \\.\RSTask\AcmeCo\AcmeAppName\MyTasks.ttg. For performance, RSTask caches open files in memory and does not pass along changes to the copy to the on-disk file until the file is closed or until a global hourly timer goes off. (Users may change this period.)

Applications communicate with RSTask by performing I/O control requests on these virtual files. Supported control requests include beginning a task of a certain type and acquiring a task ID for it, ending the task with a given ID, changing the deadline for a task type, resetting the sample of recent work requirements for a task type, and various other minor ones. RSTask supports *fast I/O* control requests [15], a Windows 2000 optimization that speeds up I/O operations. As a further optimization, RSTask has a control request that ends one task and begins another; the automatic task detector in RSLib uses this to quickly signal the end of the previous user interface task when an application receives a new one.

5.5 Task manager and sample queue

RSTask keeps track of ongoing tasks and makes appropriate calls to the scheduler when tasks begin and end. Also, when a task ends, the task manager queues the information about how long this task took in the *sample queue*. It does not immediately invoke the PACE calculator since PACE calculation is best done when the CPU is otherwise idle.

As stated in Section 4.7, when no tasks are ongoing, we wait for a short period then initiate an interval-based scheduler. We do this on the Transmeta system in the following way. When RSTask detects the departure of the last ongoing task, it switches to the lowest available speed and sets a 50 ms timer. When the timer expires, it enters the LongRun$^{\text{TM}}$ automatic speed scheduling mode, which uses an interval-based strategy. We chose 50 ms because this is further backward than LongRun$^{\text{TM}}$'s scheduler ever looks. We have not yet implemented a scheme using an interval-based scheduler on the AMD system.

5.6 Idleness detector and automatic schedule computer

The idleness detector is another major component of RSTask. It is a thread running at priority 5, just above the idle level, so that it can easily detect when no important threads remain unblocked. If it is scheduled when an I/O is ongoing, it tells RSTask to pause the current schedule; RSIoCnt will later tell RSTask to resume the schedule when no synchronous I/O's remain in the system. If the idleness detector runs when no I/O is ongoing, it notifies RSTask that all tasks are complete. The other responsibility of the idleness detector is to invoke the PACE calculator on all unprocessed entries in the sample queue when the system is otherwise idle. Not only does this cause the overhead of PACE calculation to occur only when the system is idle, it also eliminates overhead due to saving and restoring floating-point state, as we will now describe.

The Windows 2000 kernel does not use floating-point instructions, so for performance reasons it does not save floating-point state when entering kernel mode or restore such state when leaving it. If we ran the PACE calculator in the kernel at arbitrary times, e.g., whenever a task completed, it would have to save and restore floating-point state to avoid corrupting the state of whatever thread it interrupted. By doing PACE calculation in the context of its own special thread, we make such save and restore operations unnecessary.

5.7 I/O counter

The other kernel module we will discuss is RSIoCnt. Its job is to count the pending synchronous I/O's and store this count in shared memory where the idleness detector can access it. It must also tell RSTask to resume any paused schedule whenever this count becomes zero.

We implemented RSIoCnt as a file system filter driver. A filter driver implements a filter device, a special kind of device extremely helpful in tracing system events in Windows NT/2000. A filter device can *attach* to an existing device, causing it to intercept any requests destined for that existing device. For more information about them, see [15, 10]. Our filter driver has low overhead because it merely counts the requests as they start and stop and passes them on.

Unfortunately, our approach limits one to filtering only non-network file systems. There are undocumented ways to filter network file systems and network devices, as shown in [10], but we do not do this in our prototype due to our stability goal.

5.8 User-mode library

We use a well-documented method to load RSLib, a user-mode library, into the address space of every process that makes GUI calls [14]. The main activity of this library is interposing on the delivery of user interface events to the application by using a *message hook* [14]. With this mechanism, we tell Windows to call a given function just before it successfully completes an application's request for the next message from the GUI.

RSLib also exports functions that applications can use. Most of these allow applications to specify task information. Applications can interact with RSTask without these calls, but they are helpful to application writers who prefer to use a function call interface rather than make I/O control calls to a virtual file system. RSLib exports other miscellaneous functions letting applications do things like disable automatic detection of their tasks.

6 Results

6.1 General overhead

In this subsection, we evaluate system overhead just from RightSpeed running unused in the background. There are two main sources of this overhead: (a) making the timer interrupt every 1 ms instead of every 10 ms causes interrupt-processing time to increase; and (b) filtering I/O requests to count them increases the time to perform each I/O.

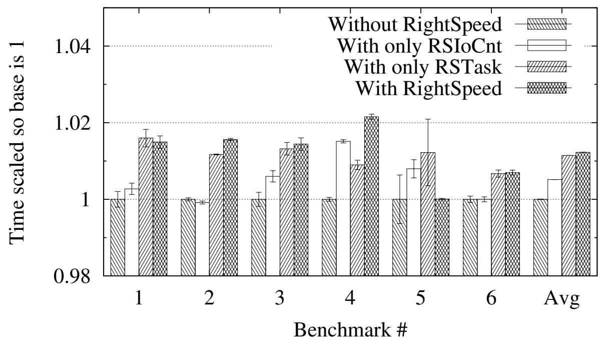

Figure 3: Time to perform various benchmarks without RightSpeed and with various components of RightSpeed enabled, shown with 95% confidence intervals. Note that the Y-axis origin is not zero.

To evaluate these effects, we ran the following benchmarks on a system with a 450 MHz Pentium III:

1. Read an uncached 32 KB file
2. Write a 100 KB file with write-through
3. Read 32 KB directly from the disk
4. Compile the RightSpeed logger device with the Windows DDK
5. Format a Ph.D. dissertation with LaTeX
6. Perform a CPU-intensive mathematical loop

We ran them without any RightSpeed modules loaded, with only the RSIoCnt module loaded, with only the RSTask module loaded, and with both of those two modules loaded. In all cases, we disabled the network to avoid interference from network interrupts. None of these benchmarks *use* RightSpeed at all; indeed, we did not even install RSLib to perform these experiments. We ran each benchmark enough times that the 95% confidence interval about the sample mean included no values more than 0.01% away from the sample mean, or 10,000 runs occurred, or 2,000 seconds passed, whichever came first. Figure 3 shows results.

We see that RSIoCnt adds 0.3–1.5% overhead, with an average of 0.5%, due to filtering I/O operations. If we did not have to use a file system filter to do this, e.g., if Microsoft provided hooks allowing one to simply count ongoing I/O's and be notified when the last I/O leaves the system, this overhead would likely be lower. We also observe that RSTask, by virtue of it reducing timer granularity from 10 ms to 1 ms, increases operation times by 0.7–1.6% with an average of 1.1%, presumably due to the system responding to more frequent timer interrupts. Combined, the overhead is 1.2% on average.

Operation	Time
Load and initialize RSLib for a process	1.401 ms
Install message hook	8.532 μs
Open system auto task type group file	159.777 μs
Get application name	12.583 μs
Open per-app auto task type group file	121.229 μs
Intercept non-user-interface message	2.265 μs
Intercept & handle user interface message	7.605 μs
Evaluate message type	1.013 μs
End task and begin another	3.575 μs
Simple I/O control request to RSTask	1.162 μs
Begin a task	3.450 μs
Kernel-mode component	1.345 μs
End a task	2.530 μs
Kernel-mode component	1.134 μs
End one task and begin another	3.462 μs
Kernel-mode component	1.441 μs

Table 3: Average time RightSpeed takes to perform common operations on the AMD machine at 900 MHz

6.2 Time to perform RightSpeed operations

The next set of results evaluates the time to perform various RightSpeed operations. We performed these measurements on the AMD system, since accurately evaluating performance on the Transmeta system is difficult for two reasons: (a) the dynamic translation of code the chip performs can cause large differences from one run to another, and (b) confidentiality agreements preclude us from publishing certain measurements of the prototype system. In all cases speed changing was disabled to not confound the measurement of durations, so all runs are at 900 MHz. Most of these results we measured directly by making an entry in the log each time an operation started or stopped. However, some of them, such as intercepting a message, involve hidden overhead, so we measured them by running with and without the operation and subtracting. We ran each operation 10,100 times and discarded the first 100. Table 3 shows the mean results.

We see that the overhead of linking RSLib into each application is about 1.4 ms; this occurs only once per application, when it starts. Some of this is RSLib's initialization, including installing the message hook and opening the automatic task type group files, but this accounts for little of it. In these benchmarks, the application task type group file is in the file cache, but even if it were not the time to load would not be significantly more.

The overhead of hooking all messages delivered to applications is also small. For non-user-interface messages, the overhead is 2.3 μs per message. For user interface messages, the message hook must determine the event type and communicate that this task is beginning and the previous task is ending to RightSpeed. The total extra time is small, approximately 7.6 μs per message. Considering that messages arrive on the order of every few milliseconds, and that user interface messages arrive even less often (at worst about every 14 ms in the case of rapid mouse movement, and more typically about once every 150 ms if the user is typing at 40 words per minute), the total overhead is low.

RightSpeed operation microbenchmarks show more detail about the cause of overhead. Each I/O control request takes about 1–2 μs due to the time to trap into kernel mode and to check and copy data from user buffers to kernel buffers. Inside RSTask, the time to begin a task is about 1.3 μs and the time to end a task is about 1.1 μs. The most common operation, beginning one task and ending another that is already considered complete, takes about 1.4 μs of kernel time. Note that this is less than the sum of the time to begin a task and to end a task because of various optimizations for this case. For example, we look up the task type group file only once and we acquire and release the spin lock controlling access to the ongoing tasks list only once.

6.3 Effect on performance

Applications can specify performance targets for tasks. However, since Windows 2000 is not a real-time operating system, scheduling decisions do not necessarily happen precisely when they should, so RightSpeed will not necessarily meet these targets. In this subsection, we evaluate how closely it does.

These evaluations require workloads. We derived these workloads from traces of users performing their normal business on desktop machines running Windows NT or Windows 2000. For more details on the tracing, see [10]. Each workload corresponds to all tasks requiring no I/O that were triggered by keystroke and mouse click events delivered to a particular application for a particular user during the several months that user was traced. Table 4 gives a brief description of each workload; for more details about the users and applications, see [12] and [11]. We inferred when tasks began and ended using the method from [12].

Our traces do not give us sufficient information to precisely recreate the workloads. For instance, we do not collect disk contents and we irreversibly encrypt alphanumeric keystrokes. To simulate RightSpeed, however, we need only know when and for how long each task ran. Thus, we use a simulator that simulates each task by performing additions repeatedly in a tight loop for the same number of cycles as the original task took. Our workload simulator indicates the beginning of each task to RightSpeed with an explicit RSLib call; it sleeps

Workload	User	Application	Key events	Click events
1	1	explorer	17,105	9,549
2	2	explorer	27,972	19,866
3	3	explorer	9,905	2,617
4	4	explorer	11,276	6,301
5	5	explorer	21,297	9,291
6	6	explorer	6,096	5,381
7	7	explorer	6,938	4,443
8	8	explorer	24,208	9,337
9	1	netscape	797,642	22,512
10	2	iexplore	193,823	59,667
11	3	psp	64,229	3,320
12	4	outlook	359,839	14,984
13	5	outlook	109,202	2,633
14	6	grpwise	275,972	13,576
15	7	winword	50,799	2,766
16	8	excel	13,891	2,016

Table 4: Traced application workloads we use in certain simulations

for 2 ms at the end of each task to let RightSpeed automatically detect the end of the task. We run this workload simulator on the AMD machine to measure the performance obtained when RightSpeed schedules the speeds.

We evaluate RightSpeed's performance as follows. We assign performance targets corresponding to an average pre-deadline speed of 630 MHz for keystroke tasks and 765 MHz for mouse click tasks. For each workload, we calculate how many deadlines it theoretically should miss and how much total delay past deadlines it should achieve. We then simulate RightSpeed to see how many deadlines it actually misses and the total delay it actually achieves.

We find that RightSpeed misses 1.5% fewer to 0.3% more deadlines than the target, with an average absolute error of 0.4%. It has delay from 0.5% less to 0.1% more than the target with an average absolute error of 0.2%. Since RightSpeed conservatively rounds speeds for intervals to maximize the probability of making deadlines, it is not surprising that it tends to miss fewer deadlines and have less delay than the target. Nevertheless, the absolute error is very low, showing that RightSpeed is effective at meeting performance targets even though it must use the millisecond-granularity timer of Windows 2000 and even though Windows 2000 makes no guarantees about when speed-changing routines will actually execute.

6.4 Time to perform PACE calculations

We also measured the average time to perform PACE calculations for tasks. We performed this experiment for the user 1 workload running explorer. We found that adding the sample value to the task type group information and recomputing the schedule accordingly took an average of 4.447 μs $\pm$ 0.312 μs, the 95% confidence interval. (The standard deviation is very high, 143.633 μs, because occasionally PACE calculations are interrupted by a context switch and take milliseconds instead of microseconds to complete.) Note that these calculations were always performed at the slowest, 500 MHz setting. So, we see that PACE calculations can be made quite quickly given all our optimizations.

6.5 Effect of overhead on energy consumption

To evaluate the effect of RightSpeed overhead on energy consumption, we ran some workloads on the Transmeta machine both with and without RightSpeed. To equalize performance, we instructed RightSpeed to not use the PACE calculator but instead use an algorithm identical to Transmeta's LongRun^{TM} strategy. Table 5 shows the results for five short workloads derived from VTrace traces. We observe that the performance characteristics (deadlines missed and total delay) of RightSpeed mimicking LongRun^{TM} are very close to that of LongRun^{TM} by itself, so it is meaningful to directly compare the energy consumption of the two. We see that simulating LongRun^{TM} with RightSpeed has little effect on the total energy consumption. In other words, the overhead of signaling the beginnings and ends of tasks, and of implementing the speed schedule in software instead of hardware is insignificant.

6.6 Effect of PACE on future processors

Because the real processors on which we implemented RightSpeed derive little efficiency from using one setting versus another, PACE cannot save sufficient energy on them to make its implementation worthwhile. To evaluate the effectiveness of our PACE calculator, in this section we conduct simulations assuming future processors with better DVS characteristics. Our simulations differ from those in [11] since we do not make the same assumptions about scheduling capabilities. In particular, we consider a finite number of settings and limited timer granularity.

For our simulations, we consider three processors, each with a minimum setting running at 200 MHz and consuming 1 W, and each with power consumption proportional to speed cubed. (This cubic relationship assumes either a very low threshold voltage or a threshold voltage that is varied proportionally to supply voltage using technology like that in [8].) The three processors differ only in their maximum speeds: 600 MHz, 800 MHz,

Workload	Without RightSpeed			With RightSpeed mimicking LongRun^{TM}			Relative energy increase
	Deadlines missed	Delay	Energy	Deadlines missed	Delay	Energy	
A	21 out of 30,635	0.947 s	172.8 J	21 out of 30,635	0.940 s	174.0 J	+0.7%
B	8 out of 19,310	1.172 s	94.58 J	8 out of 19,310	1.172 s	94.86 J	+0.2%
C	1,792 out of 32,288	118.833 s	1007 J	1,796 out of 32,288	118.860 s	1006 J	-0.1%
D	61 out of 19,770	2.364 s	126.8 J	61 out of 19,770	2.366 s	126.8 J	0.0%
E	99 out of 20,641	4.070 s	379.1 J	99 out of 20,641	4.061 s	376.1 J	-0.8%

Table 5: Comparison of using built-in LongRun^{TM} scheduling versus doing this scheduling with RightSpeed

and 1 GHz. We assume the processors can only run at multiples of 50 MHz and the timer granularity is 0.1 ms.

Since our simulations occur on virtual hardware, we can run them much faster than real time. So, we can use longer workloads than those in Table 4, which were restricted to a single application. Instead, we use eight workloads, each corresponding to *all* activity of a traced user.

All the algorithms we simulate, except for the no-DVS algorithm, will use the same performance target, so that we can compare them fairly using only energy consumption. The performance target is to have an average pre-deadline speed of 400 MHz and a post-deadline speed of 600 MHz. The four algorithms we consider are:

- **Flat.** The pre-deadline speed is constant.
- **Stepped.** The pre-deadline speed begins at 200 MHz and is incremented by 50 MHz after each interval. Interval length is chosen to achieve the desired average pre-deadline speed. This models algorithms such as that used by Transmeta's LongRun^{TM} [7].
- **Past/Peg.** The pre-deadline speed is constant at 200 MHz for the first interval, then is pegged to the maximum. Interval length is chosen to achieve the desired average pre-deadline speed. This models the algorithm suggested in [5].
- **PACE.** The pre-deadline speed schedule is computed by PACE using an estimate of task work distribution derived from the most recent tasks of the same type.

Results are in Table 6 and summarized in Figure 4. Note that Flat does not change its behavior for different maximum speeds, so we present its results only for a 600 MHz maximum speed.

One interesting observation is that the greater the range of speeds available on the CPU, the more energy efficient the Stepped and PACE algorithms become. For example, per-task average CPU energy consumption under PACE decreases 19.5% when switching from a CPU with maximum speed 600 MHz to one with maximum speed 1 GHz. This is because the availability of a higher

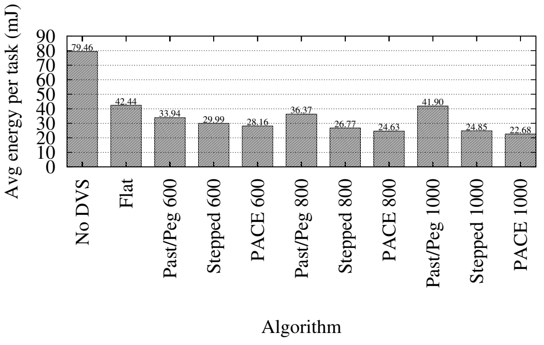

Figure 4: Summary of Table 6, showing average per-task energy consumption averaged over all workloads for various algorithms. Numbers after an algorithm identify the maximum CPU speed made available to that algorithm.

speed on the CPU allows a schedule to begin a task running more slowly, since it can more easily make up for this slowness by running even faster later in the schedule. The ability to run slowly at the beginning saves energy in the common case where the task requires little work, since the schedule never proceeds past the low-energy beginning part. PACE takes advantage of the broader range of speeds to find a better schedule, while Stepped just happens to work better with the larger set of speeds. Past/Peg, on the other hand, does worse with a greater range of speeds. Essentially, Past/Peg ignores all but the two extreme settings of the CPU, and we see that this is costly in terms of energy consumption; we conclude that using intermediate speeds can save energy.

We also see from these results that PACE is always the best algorithm, followed by Stepped, followed by Past/Peg, followed by Flat. This echoes the results from [12], and shows that even when we require PACE to deal with limited settings and timer granularity, it is still an improvement over existing DVS algorithms.

Furthermore, we predicted in Section 3 that the greater the available CPU speed range, the better PACE would do in comparison to other algorithms, and we see this borne out in our simulation results. On the CPU with maximum speed 600 MHz, PACE reduces energy con-

	Maximum speed 600 MHz					*Maximum speed 800 MHz*			*Maximum speed 1 GHz*		
User	**No DVS**	**Flat**	**P/Peg**	**Stepped**	**PACE**	**P/Peg**	**Stepped**	**PACE**	**P/Peg**	**Stepped**	**PACE**
1	44.83	23.29	16.11	14.80	13.67	15.85	13.29	11.87	16.90	12.38	10.92
2	112.36	67.00	64.62	57.07	53.60	69.44	49.37	45.59	81.19	44.75	40.93
3	81.93	39.62	31.19	25.25	23.34	35.78	23.80	21.05	42.44	22.93	20.06
4	48.07	22.20	8.44	9.04	7.78	8.90	8.66	7.39	9.75	8.44	7.18
5	80.24	41.70	25.45	24.59	23.43	25.76	21.86	20.70	28.26	20.23	19.13
6	51.20	23.47	12.02	11.12	10.09	11.71	10.80	9.39	12.39	10.61	9.17
7	132.34	77.22	73.37	64.29	61.26	78.65	55.99	52.48	91.16	51.00	47.47
8	84.75	45.02	40.32	33.74	32.10	44.84	30.43	28.55	53.09	28.44	26.61
Avg	*79.46*	*42.44*	*33.94*	*29.99*	*28.16*	*36.37*	*26.77*	*24.63*	*41.90*	*24.85*	*22.68*

Table 6: Simulation results showing average per-task energy consumption, in mJ, for various algorithms, workloads, and maximum CPU speeds. All algorithms except "No DVS" achieve the same performance target by using a 400 MHz average pre-deadline speed and a 600 MHz constant post-deadline speed. P/Peg stands for Past/Peg.

sumption by 6.1% compared to Stepped; with maximum speed 800 MHz, the reduction is 8.0%; with maximum speed 1 GHz, the reduction is 8.7%.

In conclusion, we find that even when a finite set of speeds are available and the timer granularity is limited, PACE is still an improvement over other algorithms. We find that having higher speeds available on the CPU helps PACE reduce energy consumption, and furthermore PACE does better the greater the range of speeds available on the CPU. This is an important lesson for chip designers, who may think that providing the capability of running at high voltages and therefore high speeds will increase energy consumption. We see here that with proper energy management using PACE, provision of higher speeds can actually *reduce* energy consumption.

7 Future work

7.1 Modifying applications

An important next step in this research is to insert calls to RightSpeed into various applications, such as movie players, to communicate task information to RightSpeed. We have shown that RightSpeed is good at meeting deadline targets, and this will pay off better once we modify applications in this manner.

7.2 User testing

In this paper, we have relied on user interface studies that suggest a connection between making deadlines and user-perceived response time instead of conducting user experiments ourselves. It will be important in future work to make sure that the performance targets RightSpeed assigns to automatically detected tasks ensure a satisfactory user experience.

7.3 PACE calculator

We hope in future to test the PACE calculator on a real system with a large range of worthwhile settings to evaluate its actual effect on the energy consumption of such a system.

7.4 Specification of performance targets

For some applications, the best way to specify the performance target may not be an average pre-deadline speed or an equivalent DVS algorithm, but rather a target fraction of deadlines to make, e.g., to say that 99% of tasks should complete by their deadline. In future, we would like to devise a way for RightSpeed to meet this kind of performance target with high accuracy and energy efficiency.

7.5 Predicting I/O

Our approach to dealing with I/O is somewhat unsatisfactory, as we do not consider the I/O time a task requires until after it actually occurs. A better approach would be to model the probability distribution of task I/O requirements for each task type and use this distribution to compute a more optimal schedule at the outset of the task. This requires a more complicated model of speed and voltage scheduling, and consequently a more complicated solution to computing an optimal schedule than PACE currently uses.

8 Summary and Conclusions

We implemented RightSpeed, a task-based speed and voltage scheduler for systems running Windows 2000 on Transmeta or AMD processors. Unlike traditional DVS schedulers, which use interval-based methods to change speed merely according to recent CPU usage, RightSpeed considers tasks and their performance constraints. RightSpeed is an improvement over other task-based schedulers since it uses PACE to compute optimal speed schedules and uses an efficient heuristic to automatically detect tasks triggered by user interface events. RightSpeed also distinguishes itself by running on Windows, the most popular laptop operating system.

RightSpeed obtains task information in two ways. First, applications can directly indicate when tasks begin and end, what type of task each task is, and the performance targets for each task type. Second, RightSpeed uses an *automatic task detector* to infer task information for applications not using the RightSpeed task specification interface. This detector infers that a task begins whenever it observes a user interface event such as a keystroke.

RightSpeed also features a *PACE calculator*. This allows RightSpeed to automatically monitor the work requirements of tasks as they complete, deduce a probability distribution of work requirements for each task type, and from those to compute optimal schedules for scheduling CPU speed when tasks of those type run. It computes these schedules using the theory of PACE, described in [11].

We demonstrated that RightSpeed can meet performance targets applications specify, despite the fact that Windows 2000 does not provide scheduling guarantees. We also demonstrated that the overhead due to using RightSpeed is small. The overhead due to low-level system modifications, including monitoring I/O and increasing timer resolution, is only 1.2% on average. The overhead due to other aspects of RightSpeed is also modest, on the order of a few microseconds to perform most operations. Even PACE calculation, involving complicated floating-point operations, takes only about 4.4 μs per task on a 500 MHz processor, thanks to several optimizations.

The systems to which we ported RightSpeed have DVS characteristics quite different from the idealized conditions given in [11], since they have limited scheduling granularity, a limited supply of speeds, and a nonlinear relationship between speed squared and energy. We therefore developed techniques to apply PACE to such real systems, and implemented them in RightSpeed.

Unfortunately, these processors derive so little efficiency from using one setting versus another that actual savings from the PACE calculator are minuscule. One system even contains settings that are never worthwhile for PACE schedules to use. This departure from the theoretical model may result from overly conservative speed/voltage settings from the chip manufacturer or poor circuit engineering, or it may reflect problems with voltage scaling not reflected in the standard theoretical model. In any case, assuming that the problems are with the chips and not with the model, we therefore performed simulations on theoretical processors whose settings' efficiencies more closely match those expected from semiconductor theory.

We found in our simulations that our version of PACE, optimized for speed and modified to take into account limits of speed and time granularity on real systems, saves energy compared to other algorithms. Furthermore, PACE is most effective at improving algorithms when the CPU has a large speed range. PACE reduces energy consumption compared to the Stepped algorithm by 6.1% when the speed range is 200 MHz–600 MHz; this relative improvement rises to 8.7% when the speed range expands to 200 MHz–1 GHz.

We also found that as long as one uses the PACE algorithm, CPU energy decreases when the range of available speeds increases. For example, on a CPU with a speed range of 200 MHz–1 GHz, we consume 19.5% less energy than on a CPU with a speed range of only 200 MHz–600 MHz. An important lesson from this is that the current practice of reducing the maximum speed of processors marketed for mobile environments may be misguided. Providing the ability to run at a high speed, even if it can only be for a short time due to thermal constraints, can not only make a processor more attractive to consumers evaluating them in terms of their maximum performance, but can also actually reduce energy consumption by providing DVS algorithms with more options. To take advantage of these options, however, the system needs to use an algorithm like PACE that only uses high speeds when necessary.

The code for RightSpeed is available on the World Wide Web at http://www.cs.berkeley.edu/~lorch/rightspeed/. Although Jacob Lorch is currently affiliated with Microsoft, he performed all implementation work while still a student at UC Berkeley. Thus, the implementation used no internal Microsoft knowledge or documentation.

9 Acknowledgments

This material is based upon work supported by the State of California MICRO program, AT&T Laboratories, Cisco Systems, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sony Research Laboratories, Sun Microsystems, Toshiba Corporation, and Veritas Software.

This work would not have been possible without the cooperation of people from both Transmeta and AMD, who provided not only hardware but also detailed guidance in using it. Those people include Marc Fleischmann and Rowan Hamilton from Transmeta and Richard Russell, Dave Tobias, and Fred Weber from AMD. Also, we offer great thanks to the many users of our tracer whose traces yielded the workloads for this paper. Finally, we thank the anonymous reviewers of this paper and especially our shepherd, Deborah Wallach, for their many helpful comments and suggestions.

References

[1] AMD. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. On the World Wide Web at http://www.amd.com/products/cpg/athlon/techdocs/pdf/24319.pdf, August 2001.

[2] E. Chan, K. Govil, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM 95)*, pages 13–25, November 1995.

[3] K. Flautner and T. Mudge. Vertigo: automatic performance-setting for Linux. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–116, December 2002.

[4] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the Seventh ACM International Conference on Mobile Computing and Networking (MOBICOM 2001)*, July 2001.

[5] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[6] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of the International Conference on Computer Aided Design*, pages 653–656, November 1998.

[7] A. Klaiber. The technology behind Crusoe$^{\text{TM}}$ processors. White paper, Transmeta Corporation, January 2000.

[8] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama. Variable supply-voltage scheme for low-power high-speed CMOS digital design. *IEEE Journal of Solid-State Circuits*, 33(3):454–462, March 1998.

[9] J. R. Lorch. *Operating Systems Techniques for Reducing Processor Energy Consumption*. PhD thesis, Computer Science Division, EECS Department, University of California at Berkeley, 2001. Available online at http://www.cs.berkeley.edu/∼lorch/papers/.

[10] J. R. Lorch and A. J. Smith. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, October 2000.

[11] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the 2001 ACM SIGMETRICS Conference*, pages 50–61, June 2001.

[12] J. R. Lorch and A. J. Smith. Using user interface event information in dynamic voltage scaling algorithms. Technical Report UCB/CSD-02-1190, Computer Science Division, EECS, University of California at Berkeley, August 2002.

[13] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *Proceedings of INTERCHI '93*, pages 24–29, April 1993.

[14] Microsoft Corporation. *Platform SDK Documentation*, 2000.

[15] R. Nagar. *Windows NT File System Internals*. O'Reilly and Associates, Inc., Sebastopol, CA, 1997.

[16] G. Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, Indianapolis, IN, 2000.

[17] T. Pering, T. Burd, and R. W. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[18] T. Pering, T. Burd, and R. W. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 96–101, July 2000.

[19] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102, October 2001.

[20] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, 1998.

[21] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.

[22] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, MA, 1993.

[23] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE 36th Annual Symposium on Foundations of Computer Science*, pages 374–382, October 1995.