Appears in "Proceedings of the First Symposium on Operating Systems Design and Implementation," Usenix Association, November 1994.

Scheduling for Reduced CPU Energy

Mark Weiser, Brent Welch, Alan Demers, Scott Shenker
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
{weiser,welch,demers,shenker}@parc.xerox.com

Abstract

The energy usage of computer systems is becoming more important, especially for battery operated systems. Displays, disks, and cpus, in that order, use the most energy. Reducing the energy used by displays and disks has been studied elsewhere; this paper considers a new method for reducing the energy used by the cpu. We introduce a new metric for cpu energy performance, millions-of-instructions-per-joule (MIPJ). We examine a class of methods to reduce MIPJ that are characterized by dynamic control of system clock speed by the operating system scheduler. Reducing clock speed alone does not reduce MIPJ, since to do the same work the system must run longer. However, a number of methods are available for reducing energy with reduced clock-speed, such as reducing the voltage [Chandrakasan *et al* 1992][Horowitz 1993] or using reversible [Younis and Knight 1993] or adiabatic logic [Athas *et al* 1994].

What are the right scheduling algorithms for taking advantage of reduced clock-speed, especially in the presence of applications demanding ever more instructions-per-second? We consider several methods for varying the clock speed dynamically under control of the operating system, and examine the performance of these methods against workstation traces. The primary result is that by adjusting the clock speed at a fine grain, substantial CPU energy can be saved with a limited impact on performance.

1 Introduction

The energy use of a typical laptop computer is dominated by the backlight and display, and secondarily by the disk. Laptops use a number of techniques to reduce the energy consumed by disk and display, primarily by turning them off after a period of no use [Li 1994][Douglis 1994]. We expect slow but steady progress in the energy consumption of these devices. Smaller computing devices often have no disk at all, and eliminate the display backlight that consumes much of the display-related power. Power consumed by the CPU is significant; the Apple Newton designers sought to maximize MIPS per WATT [Culbert 1994]. This paper considers some methods of reducing the energy used for executing instructions. Our results go beyond the simple power-down-when-idle techniques used in today's laptops.

We consider the opportunities for dynamically varying chip speed and so energy consumption. One would like to give users the appearance of a 100MIPS cpu at peak moments, while drawing much less than 100MIPS energy when users are active but would not notice a reduction in clock rate. Knowing when to use full power and when not requires the cooperation of the operating system scheduler. We consider a number of algorithms by which the operating system scheduler could attempt to optimize system power by monitoring idle time and reducing clock speed to reduce idle time to a minimum. We simulate their performance on some traces of process scheduling and compare these results to the theoretical optimum schedules.

2 An Energy Metric for CPUS

In this paper we use as our measure of the energy performance of a computer system the MIPJ, or millions of instructions per joule. MIPS/WATTS = MIPJ. (Of course MIPS have been superseded by better metrics, such as Specmark: we are using MIPS to stand for any such workload-per-time benchmark). MIPJ is not improving that much for high-end processors. For example, a 1984 2-MIPS 68020 consumed 2.0 watts (at 12.5Mhz), for a MIPJ of 1, and a 1994 200-MIPS Alpha chip consumes 40 watts, so has a MIPJ of 5. However, more recently lower speed processors used in laptops have been optimized to run at low power. For example, the Motorola 68349 is rated at 6 MIPS and consumes 300 mW for 20 MIPJ.

Other things being equal, MIPJ is unchanged by changes in clock speed. Reducing the clock speed causes a linear reduction in energy consumption, but a similar reduction in MIPS. The two effects cancel. Similarly, turning the computer off, or reducing the clock to zero in the “idle-loop”, does not effect MIPJ, since no instructions are being executed. However, a reduced clock speed creates the opportunity for quadratic energy savings; as the clock speed is reduced by n, energy per cycle can be reduced by n^2. Three methods that achieve this are voltage reduction, reversible logic, and adiabatic switching. Our simulations assume n^2 savings, although it is really only important that the energy savings be greater than the amount by which the clock rate is reduced in order to achieve an increase in MIPJ.

Voltage reduction is currently the most promising way to save energy. Already chips are being manufactured to run at 3.3 or 2.2 volts instead of the 5.0 voltage levels commonly used. The intuition behind the power savings comes from the basic energy equation that is proportional to the square of the voltage.

$$E/clock \propto V^2$$

The settling time for a gate is proportional to the voltage; the lower the voltage drop across the gate, the longer the gate takes to stabilize. To lower the voltage and still operate correctly, the cycle time must be lowered first. When raising the clock rate, the voltage must be increased first. Given that the voltage and the cycle time of a chip could be adjusted together, it should be clear now that the lower-voltage, slower-clock chip will dissipate less energy per cycle. If the voltage level can be reduced linearly as the clock rate is reduced, then the energy savings per instruction will be proportional to the square of the voltage reduction. Of course, for a real chip it may not be possible to reduce the voltage linear with the clock reduction. However, if it is possible to reduce the voltage at all by running slower, then there will be a net energy savings per cycle.

Currently manufacturers do not test and rate their chips across a smooth range of voltages. However, some data is available for chips at a set of voltage levels. For example, a Motorola CMOS 6805 microcontroller (cloned by SGS-Thomson) is rated at 6 Mhz at 5.0 Volts, 4.5 Mhz at 3.3 Volts, and 3 Mhz at 2.2 Volts. This is a close to linear relationship between voltage and clock rate.

The other important factor is the time it takes to change the voltage. The frequency for voltage regulators is on the order of 200 KHz, so we speculate that it will take a few tens of microseconds to boost the voltage on the chip.

Finally, why run slower? Suppose a task has a deadline in 100 milliseconds, but it will only take 50 milliseconds of CPU time when running at full speed to complete. A normal system would run at full speed for 50 milliseconds, and then idle for 50 milliseconds (assuming there were no other ready tasks). During the idle time the CPU can be stopped altogether by putting it into a mode that wakes up upon an interrupt, such as from a periodic clock or from an I/O completion. Now, compare this to a system that runs the task at half speed so that it completes just before its deadline. If it can also reduce the voltage by half, then the task will consume 1/4 the energy of the normal system, even taking into account stopping the CPU during the idle time. This is because the same number of cycles are executed in both systems, but the modified system reduces energy use by reducing the operating voltage. Another way to view this is that idle time represents wasted energy, even if the CPU is stopped!

3 Approach of This Paper

This paper evaluates the fine grain control of CPU clock speed and its effect on energy use by means of trace-driven simulation. The trace data shows the context switching activity of the scheduler and the time spent in the idle loop. The goals of the simulation are to evaluate the energy savings possible by running slower (and at reduced voltage), and to measure the adverse affects of running too slow to meet the supplied demand. No simulation is perfect, however, and a true evaluation will require experiments with real hardware.

Trace data was taken from UNIX workstations over many hours of use by a variety of users. The trace data is described in Section 4 of the paper. The assumptions made by the simulations are described in Section 5. The speed adjustment algorithms are presented in Section 6. Section 7 evaluates the different algorithms on the basis of energy savings and a delay penalty function. Section 8 discusses future work, including some things we traced but did not fully utilize in our simulations. Finally, Section 9 provides our conclusions.

4 Trace Data

Trace data from the UNIX scheduler was taken from a number of workstations over periods of up to several hours during the working day. During these times the workloads included software development, documentation, e-mail, simulation, and other typical activities of engineering workstations. In addition, a few short traces were taken during specific workloads such as typing and scrolling through documents. Appendix I has a summary of the different traces we used.

Table 1: Trace Points

SCHED	Context switch away from a process
IDLE_ON	Enter the idle loop
IDLE_OFF	Leave idle loop to run a process
FORK	Create a new process
EXEC	Overlay a (new) process with another program
EXIT	Process termination
SLEEP	Wait on an event
WAKEUP	Notify a sleeping process

The trace points we took are summarized in Table 1. The idle loop events provide a view on how busy the machine is. The process information is used to classify different programs into foreground and background types. The sleep and wakeup events are used to deduce job ordering constraints.

In addition, the program counter of the call to sleep was recorded and kernel sources were examined to determine the nature of the sleep. The sleep events were classified into waits on "hard" and "soft" events. A hard event is something like a disk wait, in which a sleep is done in the kernel's biowait() routine. A soft event is something like a select that is done awaiting user input or a network request packet. The goal of this classification is to distinguish between idle time that can be eliminated by rescheduling (soft idle) and idle that is mandated by a wait on a device (hard idle).

Each trace record has a microsecond resolution time stamp. The trace buffer is periodically copied out of the kernel, compressed, and sent over the network to a central collection site. We used the trace data to measure the tracing overhead, and found it to range from 1.5% to 7% of the traced machine.

5 Assumptions of the Simulations

The basic approach of the simulations was to lengthen the runtime of individually scheduled segments of the trace in order to eliminate idle time. The trace period was divided into intervals of various lengths, and the runtime and idletime during that interval were used to make a speed adjustment decision. If there were excess cycles left over at the end of an interval because the speed was too slow, they were carried over into the next interval. This carry-over is used as a measure of the penalty from using the speed adjustment.

The ability to stretch runtime into idle periods was refined by classifying sleep events into "hard" and "soft" events. The point of the classification is to be fair about what idle time can be squeezed out of the simulated schedule by slowing down the processor. Obviously, running slower should not allow a disk request to be postponed until just before the request completes in the trace. However, it is reasonable to slow down the response to a keystroke in an editor such that the processing of one keystroke finishes just before the next.

Our simulations did not reorder trace data events. We justify this by noting that only if the offered load is far beyond the capacity of the CPU will speed changes affect job ordering significantly. Furthermore, the CPU speed is ramped up to full speed as the offered load increases, so in times of high demand the CPU is running at the speed that matches the trace data.

In addition, we made the following assumptions:

The machine was considered to use no energy when idle, and to use energy/instruction in proportion to n^2 when running at a speed n, where n varies between 1.0 and a minimum relative speed. This is a bit optimistic because a chip will draw a small amount of power while in standby mode, and we might not get a one-to-one reduction in voltage to clock speed. However, the baseline power usages from running at full speed (reported as 1.0 in the graphs) also assume that the CPU is off during idle times.

It takes no time to switch speeds. This is also optimistic. In practice, raising the speed will require a delay to wait for the voltage to rise first, although we speculate that the delay is on the order of 10s of instructions (not 1000s).

After any 30 second period of greater than 90% idle we assumed that any laptop would have been turned off, and skipped simulating until the next 30 second period with less than 90% idle. This models the typical power saving features already present in portables. The energy savings reported below does not count these off periods.

There was assumed to be a lower bound to practical speed, either 0.2, 0.44 or 0.66, where 1.0 represents full speed. In 5V logic using voltage reduction for power savings, these correspond to 1.0 V, 2.2 V and 3.3V minimum voltage levels, respectively. The 1.0 V level is optimistic, while the 2.2 V and 3.3V levels are based on several existing low power chips. In the graphs presented in section 7, the minimum voltage of the system is indicated, meaning that the voltage can vary between 5.0 V and the minimum, and the speed will be adjusted linearly with voltage.

6 Scheduling Algorithms

We simulated three types of scheduling algorithms: unbounded-delay perfect-future (OPT), bounded-delay limited-future (FUTURE), and bounded-delay limited-past (PAST). Each of these algorithms adjust the CPU clock speed at the same time that scheduling decisions are made, with the goal of decreasing time wasted in the idle loop while retaining interactive response.

OPT takes the entire trace, and stretches all the run times to fill all the idle times. Periods when the machine was “off” (more that 90% idle over 30 seconds) were not considered available for stretching runtimes into. This is a kind of batch approach to the work seen in the trace period: as long as all that work is done in that period, any piece can take arbitrarily long. OPT power savings were almost always limited by the minimum speed, achieving the maximum possible savings over the period. This algorithm is both impractical and undesirable. It is impractical because it requires perfect future knowledge of the work to be done over the interval. It also assumes that all idle time can be filled by stretching runlengths and reordering jobs. It is undesirable because it produces large delays in runtimes of individual jobs without regard to the need for effective response to real-time events like user keystrokes or network packets.

FUTURE is like OPT, except it peers into the future only a small window, and optimizes energy over that window, while never delaying work past the window. Again, it is assumed that all idle time in the next interval can be eliminated, unless the minimum speed of the CPU is reached. We simulated windows as small as 1 millisecond, where savings are usually small, and as large as 400 seconds, where FUTURE generally approaches OPT in energy savings. FUTURE is impractical, because it uses future knowledge, but desirable, because no realtime response is ever delayed longer than the window.

By setting a window of 10 to 50 milliseconds, user interactive response will remain high. In addition, a window this size will not substantially reduce a very long idle time, one that would trigger the spin down of a disk or the blanking of a display. Those decisions are based on idle times of many seconds or a few minutes, so stretching a computation out by a few tens of milliseconds will not affect them.

PAST is a practical version of FUTURE. Instead of looking a fixed window into the future it looks a fixed window into the past, and assumes the next window will be like the previous one. The PAST speed setting algorithm is shown at the top of the next column.

There are four parts to the code. The first part computes the percent of time during the interval when the CPU was running. The `run_cycles` come from two sources, the runtime in the trace data for the interval, and the `excess_cycles` from the simulation of the previous interval.

The `excess_cycles` represents a carry over from the previous interval because the CPU speed was set too slow to accommodate all the load that was supplied during the interval. Consider:

```
next_excess = run_cycles -
    speed * (run_cycles + soft_idle)
```

The `run_cycles` is the sum of the cycles presented by the trace data and the previous value of `excess_cycles`. This initial value is reduced by the soft idle time and the number of cycles actually per-

<u>Speed Setting Algorithm (PAST)</u>

`run_cycles` is the number of non-idle CPU cycles in the last interval.

`idle_cycles` is the idle CPU cycles, split between hard and soft idle time.

`excess_cycles` is the cycles left over from the previous interval because we ran too slow. All these cycles are measured in time units.

```
idle_cycles = hard_idle + soft_idle;
run_cycles += excess_cycles;
run_percent = run_cycles /
   (idle_cycles + run_cycles);

next_excess = run_cycles -
   speed * (run_cycles + soft_idle)
IF excess_cycles < 0. THEN
   excess_cycles = 0.

energy = (run_cycles - excess_cycles) *
   speed * speed;

IF excess_cycles > idle_cycles THEN
   newspeed = 1.0;
ELSEIF run_percent > 0.7 THEN
   newspeed = speed + 0.2;
ELSEIF run_percent < 0.5 THEN
   newspeed = speed -
      (0.6 - run_percent);

IF newspeed > 1.0 THEN
   newspeed = 1.0;
IF newspeed < min_speed THEN
   newspeed = min_speed;

speed = newspeed;
excess_cycles = next_excess;
```

formed at the current speed. This calculation represents the ability to squeeze out idle time by lengthening the runtimes in the interval. Only “soft” idle, such as waiting for keyboard events, is available for elimination of idle. As the soft idle time during an interval approaches zero, the excess cycles approach:

```
run_cycles * (1 - oldspeed)
```

The energy used during the interval is computed based on an n^2 relationship between speed and power consumption per cycle. The cycles that could not be serviced during the interval have to be subtracted out first. They will be accounted for in the next interval, probably at a higher CPU speed.

The last section represents the speed setting policy. The adjustment of the clock rate is a simple heuristic that attempts to smooth the transitions from fast to slow processing. If the system was more busy than idle, then the speed is ramped up. If it was mostly idle, then it is slowed down. We simulated several variations on the code shown here to come up with the constants shown here.

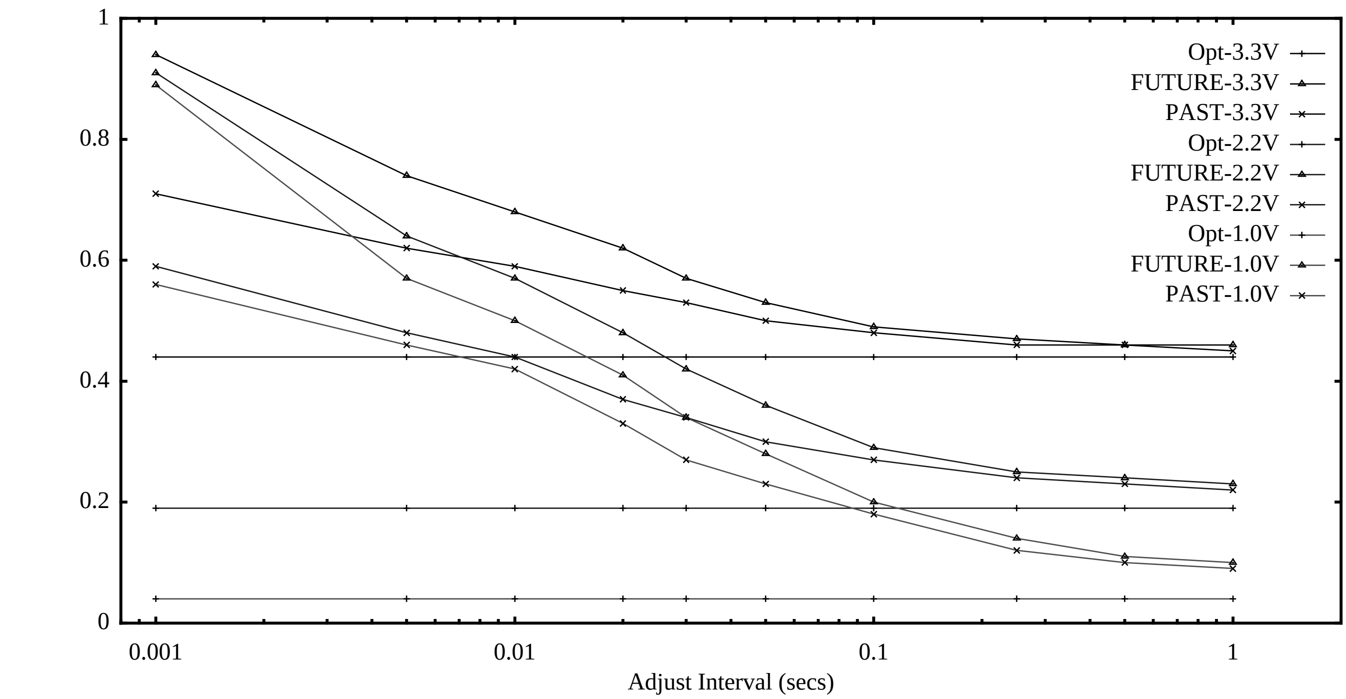
7 Evaluating the Algorithms

Figure 1compares the results of these three algorithms on a single trace (Kestrel March 1) as the adjustment interval is varied. The OPT energy is unaffected by the interval, but is shown for comparison. The vertical access shows relative power used by the scheduling algorithms, with 1.0 being full power. Three sets of three lines are shown, corresponding to three voltage levels which determine the minimum speed, and the three algorithms, OPT, FUTURE, and PAST. The PAST and FUTURE algorithms approach OPT as the interval is lengthened. (Note that the log scale for the X axis.) For the same interval PAST actually does better than FUTURE because it is allowed to defer excess cycles into the next interval, effectively lengthening the interval. The intervals from 10 msec to 50 msec are considered in more detail in other figures.

Figure 2 shows the excess cycles that result from setting the speed too slow in PAST when using a 20 msec adjustment interval and the same trace data as Figure 1. Note that the graph uses log-log scales. Cycles are measured in the time it would take to execute them at full speed. The data was taken as a histogram, so a given point counts all the excess cycles that were less than or equal that point on the X axis, but greater than the previous bucket value in the histogram. Lines are used to connect the points so that the spike at zero is evident. The large spike at zero indicates that most intervals have no excess cycles at all. There is a smaller peak near the interval length, and then the values drop off.

As the minimum speed is lowered, there are more cases where excess cycles build up, and they can accumulate in longer intervals. This is evident Figure 2

Figure 1. Kestrel March 1 All Algorithms



where the points for 1.0 V are above the others, which indicates more frequent intervals with excess cycles, and the peak extends to the right, which indicates longer excess cycle intervals.

Figure 3 shows the relationship between the interval length and the peak in excess cycle length. It compares the excess cycles with the same minimum voltage (2.2 V) while the interval length varies. This is from the same trace data as Figures 1 and 2.The main result here is that the peak in excess cycle lengths shifts right as the interval length increases. All this means is that as a longer scheduling interval is chosen, there can be more excess cycles built up.

Figure 4 compares the energy savings for the bounded delay limited past (PAST) algorithm with a 20 msec adjustment interval and with three different minimum voltage limits. In this plot each position on the X access represents a different set of trace data.The position corresponding to the trace data used in Figures 1 to 3 is indicated with the arrow.

While there is a lot of information in the graph, there are two overall points to get from the figure: the relative savings for picking different minimum voltages, and the overall possible savings across all traces.

The first thing to look for in Figure 4 is that for any given trace the three points show the relative possible energy savings for picking the three different minimum voltages.Interestingly, the 1.0 V minimum does not always result in the minimum energy. This is because it. has more of a tendency to fall behind (more excess cycles), so its speed varies more and the power consumption is less efficient. Even when 1.0 V does

Figure 2. Kestrel March 1 Penalty at 20msec

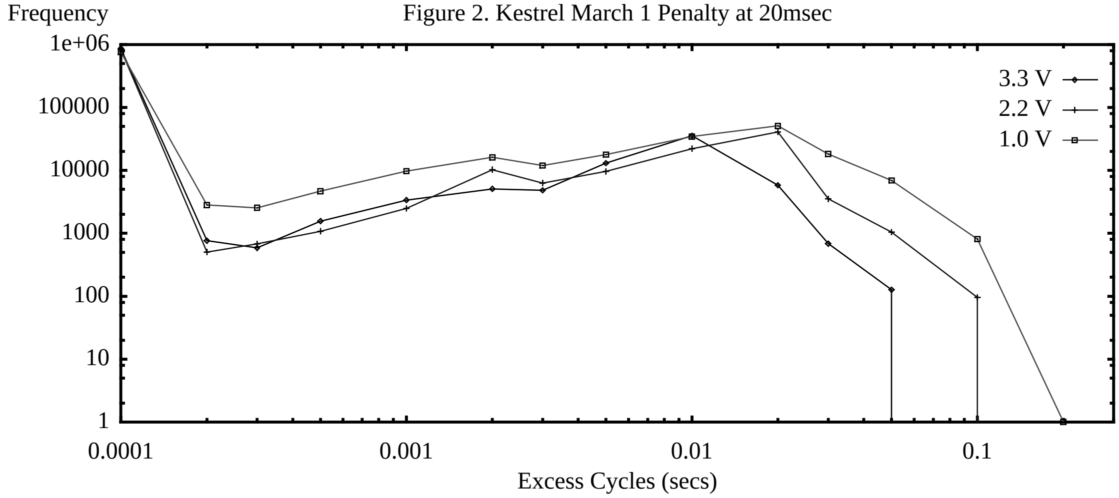
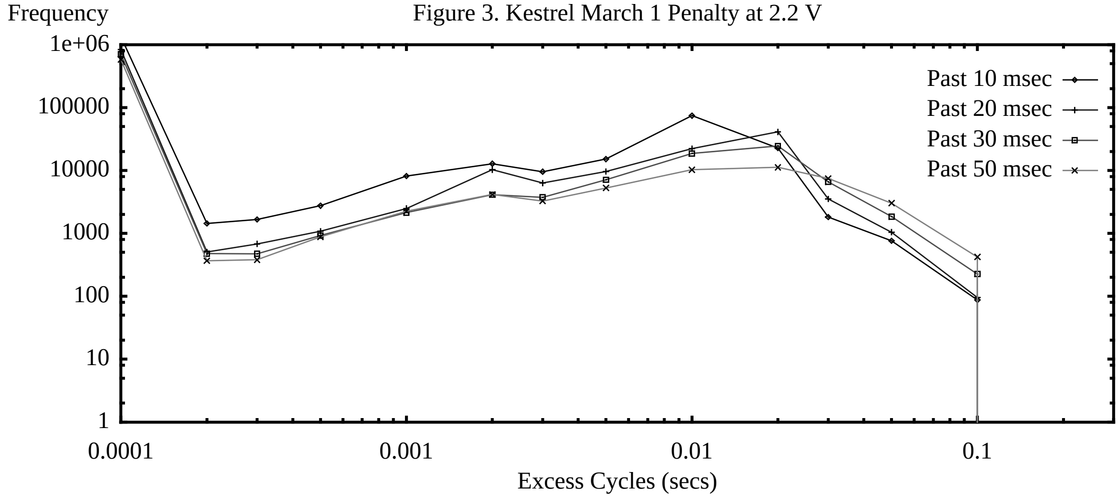


Figure 3. Kestrel March 1 Penalty at 2.2 V



provide the minimum energy, the 2.2 V minimum is almost as good.

The other main point conveyed by Figure 4 is that in most of the traces the potential for energy savings is good. The savings range from about 5% to about 75%, with most data points falling between 25% to 65% savings.

Figure 5 fixes the minimum voltage at 2.2 V and shows the effect of changing the interval length. The OPT energy savings for 2.2 V is plotted for comparision.Again, each position on the X axis represents a different trace. The position corresponding to the trace data used in Figures 1 to 3 is indicated with the arrow.

In this figure the main message to get is the difference in relative savings for a given trace as the interval is varied. This is represented by the spread in the points plotted for each trace. A longer adjustment period results in more savings, which is consistent with Figure 1.

Figures 6 and 7 show the average excess cycles for all trace runs. These averages do not count intervals with zero excess cycles. Figure 6 shows the excess cycles at a given adjustment interval (20 msec) and different minimum voltages. Figure 7 shows the excess cycles at a given minimum voltage (2.2 V) and different intervals. Again, the lower minimum voltage

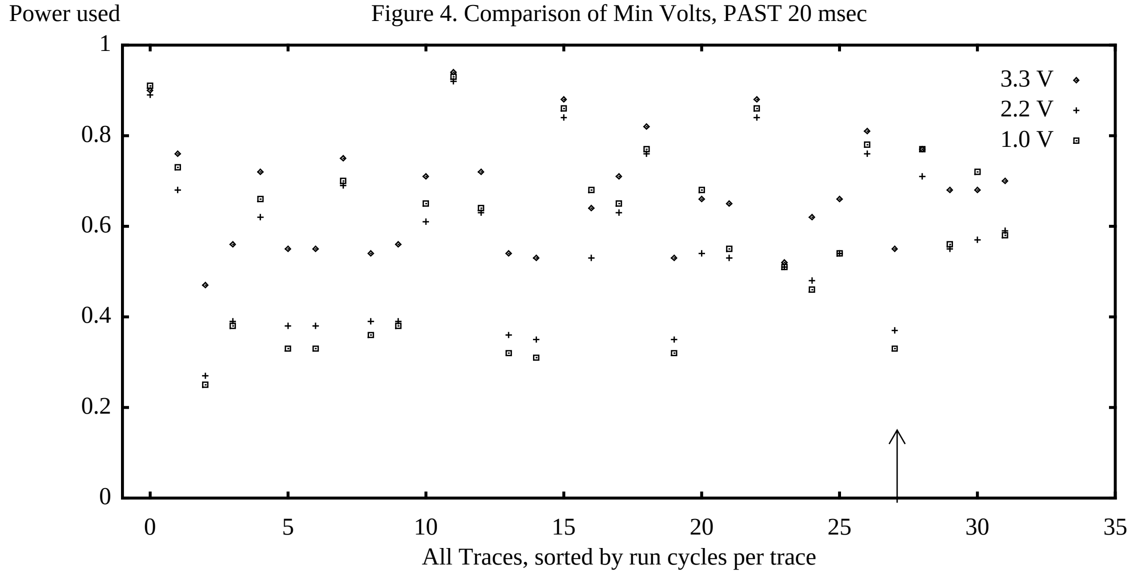


Figure 4. Comparison of Min Volts, PAST 20 msec

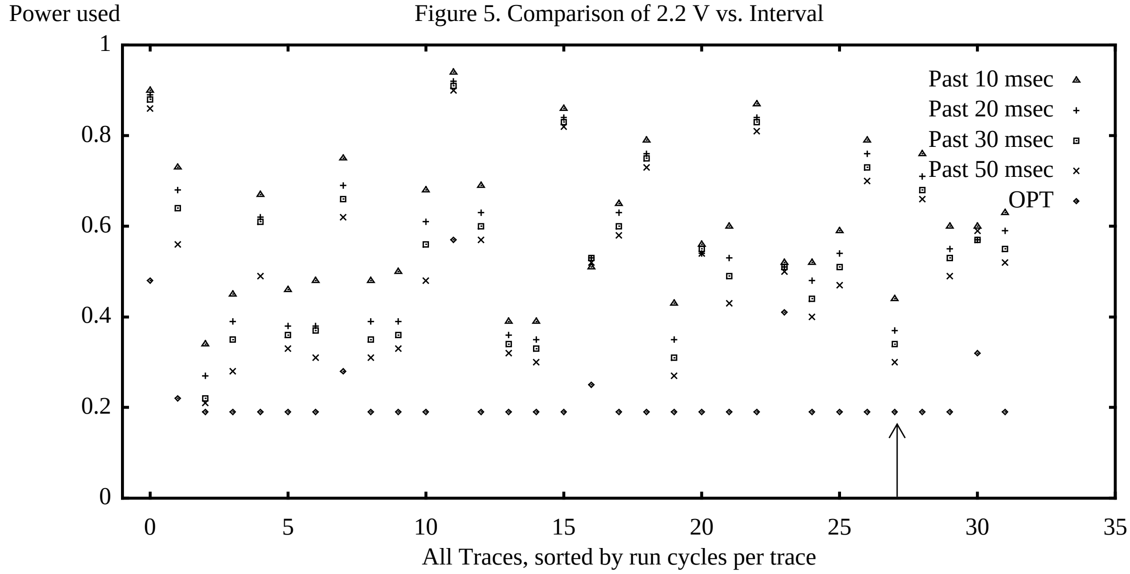


Figure 5. Comparison of 2.2 V vs. Interval

results show more excess cycles, and the longer intervals accumulate more excess cycles.

There is a trade off between the excess cycles penalty and the energy savings that is a function of the interval size. As the interval decreases, the CPU speed is adjusted at a finer grain and so it matches the offered load better. This results in fewer excess cycles, but it also does not save as much energy. This is consistent with the motivating observation that it is better to execute at an average speed than to alternate between full speed and full idle.

8 Discussion and Future Work

The primary source of feedback we used for the speed adjustment was the percent idle time of the system. Another approach is to classify jobs into background, periodic, and foreground classes. This is similar to what Wilkes proposes in his schemes to utilize idle time [Wilkes 95]. With this sort of classification the speed need not be ramped up when executing background tasks. Periodic tasks impose a constant, measurable load. They typically run for a short burst and then sleep for a relatively long time. With these tasks there is a well defined notion of “fast enough”,

Figure 6. Average Excess Cycles, 20 msec Interval

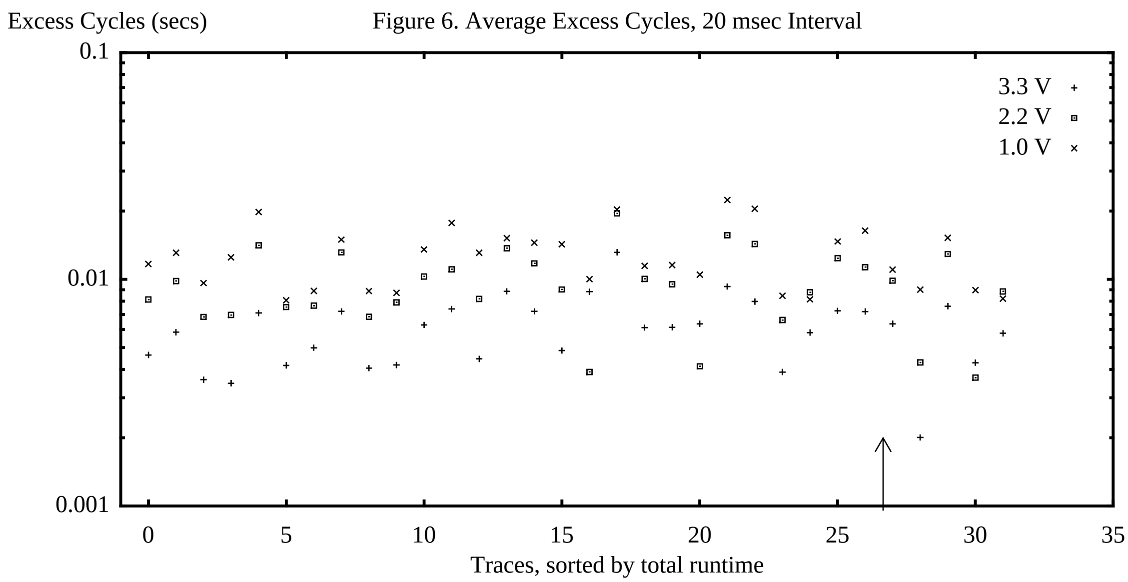
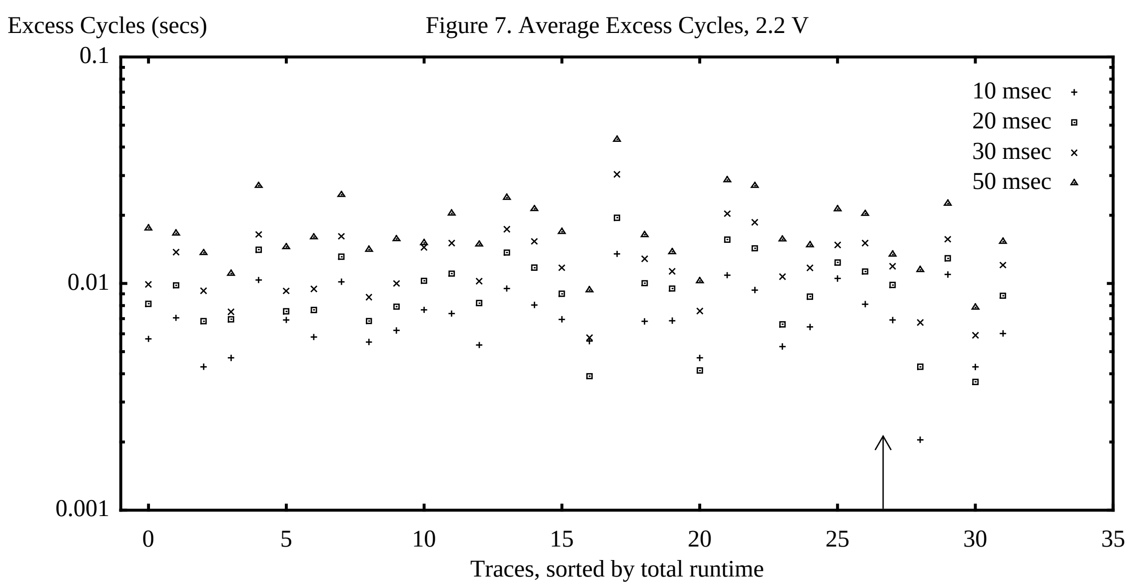


Figure 7. Average Excess Cycles, 2.2 V



and the CPU speed can be adjusted to finish these tasks just in time. When there is a combination of background, periodic, and foreground tasks, then the standard approach is to schedule the periodic tasks first, then fit in the foreground tasks, and lastly fit in the background tasks. In this case there would be a minimum speed that would always execute the periodic tasks on time, and the system would increase the speed in response to the presence of foreground and background tasks.

The simulations we performed are simplified by not reordering scheduling events. In a real rate-adjusting scheduler, the change in processing rates will have an effect on when jobs are preempted due to time slicing and the order that ready jobs are scheduled. We argue that unless there is a large job mix, then the reordering will not be that significant. Our speed adjustment algorithm will ramp up to full speed during heavy loads, and during light loads the reordering should not have a significant effect on energy.

In order to evaluate more realistic scheduling algorithms, it would be interesting to generate an abstract load for the simulation. This load includes CPU runs with preemption points eliminated, pause times due to I/O delays preserved, and causal ordering among jobs preserved. Given an abstract load, it would be possible to simulate a scheduler in more detail, giving us the ability to reorder preemption events while still preserving the semantics of I/O delays and IPC dependencies.

We have attempted to model the I/O waits by classifying idle time into “hard” and “soft” idle. We think this approximation is valid, but it would be good to verify it with a much more detailed simulation.

9 Conclusions

This paper presents preliminary results on CPU scheduling to reduce CPU energy usage, beyond the simple approaches taken by today’s laptops. The metric of interest is how many instructions are executed for a given amount of energy, or MIPJ. The observation that motivates the work is that reducing the cycle time of the CPU allows for power savings, primarily by allowing the CPU to use a lower voltage. We examine the potential for saving power by scheduling jobs at different clock rates.

Trace driven simulation is used to compare three classes of schedules: OPT that spreads computation over the whole trace period to eliminate all idle time (regardless of deadlines), FUTURE that uses a limited future look ahead to determine the minimum clock rate, and PAST that uses the recent past as a predictor of the future. A PAST scheduler with a 50 msec window shows power savings of up to 50% for conservative circuit design assumptions (e.g., 3.3 V), and up to 70% for more aggressive assumptions (2.2 V). These savings are in addition to the obvious savings that come from stopping the processor in the idle loop, and powering off the machine all together after extended idle periods.

The energy savings depends on the interval between speed adjustments. If it is adjusted at too fine a grain, then less power is saved because CPU usage is bursty. If it is adjusted at too coarse a grain, then the excess cycles built up during a slow interval will adversely affect interactive response. An adjustment interval of 20 or 30 milliseconds seems to represent a good compromise between power savings and interactive response.

Interestingly, having too low a minimum speed results in less efficient schedules because there is more of a tendency to have excess cycles and therefore the need to speed up to catch up. In particular, a minimum voltage of 2.2 V seems to provide most of the savings of a minimum voltage of 1.0 V. The 1.0 V system, however, tends to have a larger delay penalty as measured by excess cycles.

In general, scheduling algorithms have the potential to provide significant power savings while respecting deadlines that arise from human factors considerations. If an effective way of predicting workload can be found, then significant power can be saved by adjusting the processor speed at a fine grain so it is just fast enough to accommodate the workload. Put simply, the tortoise is more efficient than the hare: it is better to spread work out by reducing cycle time (and voltage) than to run the CPU at full speed for short bursts and then idle. This stems from the non-linear relationship between CPU speed and power consumption.

Acknowledgments

This work was supported in part by Xerox, and by ARPA under contract DABT63-91-C-0027; funding does not imply endorsement. David Wood of the University of Wisconsin helped us get started in this research, and provided substantial assistance in understanding CPU architecture. The authors benefited from the stimulating and open environment of the Computer Science Lab at Xerox PARC.

Appendix I. Description of Trace Data

The table on the next page lists the characteristics of the 32 traces runs that are reported in the figures. The table is sorted from shortest to longest runtime to match the ordering in Figures 4 through 7. The elapsed time of each trace is broken down into time spent running the CPU on behalf of a process (Runtime), time spent in the idle loop (IdleTime), and time when the machine is considered so idle that it would be turned off by a typical laptop power manager (Offtime). The short traces labeled mx, emacs, and fm are of typing (runs 1 and 2) and scrolling (run 3) in various editors. The remaining runs are taken over several hours of everyday use

References

William C. Athas, Jeffrey G. Koller, and Lars "J." Svensson. "An Energy-Efficient CMOS Line Driver Using Adiabatic Switching", 1994 IEEE Fourth Great Lakes Symposium on VLSI, pp. 196-199, March 1994.

A. P. Chandrakasan and S. Sheng and R. W. Brodersen. "Low-Power CMOS Digital Design". JSSC, V27, N4, April 1992, pp 473--484.

Michael Culbert, "Low Power Hardware for a High Performance PDA", *to appear* Proc. of the 1994 Computer Conference, San Francisco.

Fred Douglis, P. Krishnan, Brian Marsh, "Thwarting the Power-Hungry Disk", Proc. of Winter 1994 USENIX Conference, January 1994, pp 293-306

Mark A. Horowitz. "Self-Clocked Structures for Low Power Systems". ARPA semi-annual report, December 1993. Computer Systems Laboratory, Stanford University.

Kester Li, Roger Kumpf, Paul Horton, Thomas Anderson, "A Quantitative Analysis of Disk Drive Power Management in Portable Computers", Proc. of Winter 1994 USENIX Conference, January 1994, pp 279-292.

S. Younis and T. Knight. "Practical Implementation of Charge Recovering Asymptotically Zero Power CMOS." 1993 Symposium on Integrated Systems (C. Ebeling and G. Borriello, eds.), Univ. of Washington, 1993.

Wilkes, John "Idleness is not Sloth", *to appear*, proc. of the 1995 Winter USENIX Conf

.

Table 2: Summary of Trace Data

I	Trace	Runtime	Idle	Elapsed	Offtime
0	feb28klono	0.906	29.094	9H 24M 20S	33828.9
1	idle1	1.509	28.653	39S	9.05
2	heur1	7.043	3.103	10S	0
3	emacs2	7.585	31.719	40S	0
4	emacs1	8.060	32.273	40S	0
5	mx2	8.362	30.916	39S	0
6	mx1	9.508	30.871	41S	0
7	fm1	9.544	10.594	20S	0
8	em3	11.669	27.580	40S	0
9	fm2	16.679	23.770	41S	0
10	mx3	20.738	18.642	39S	0
11	feb28dekanore	30.548	541.045	9H 24M 40S	33307.8
12	fm3	30.626	9.942	41S	0
13	mar1klono	41.822	1011.251	9H 55M 46S	34690.6
14	feb28mezzo	61.940	449.717	9H 24M 20S	33346.1
15	mar1cleonie	214.656	1321.591	9H 50S	30913.0
16	feb28kestrel	510.259	3362.222	1H 4M 33S	0
17	feb28corvina	524.248	768.857	9H 24M 41S	32588.0
18	mar1mezzo	686.340	673.204	9H 55M 36S	34375.7
19	mar1egeus	695.409	4774.911	9H 55M 35S	30263.6
20	feb28ptarmigan	1497.908	2207.005	1H 1M 41S	0
21	feb28fandango	1703.037	3489.760	9H 24M 17S	28665.0
22	feb28zwilnik	4414.429	29448.058	9H 24M 21S	0
23	mar1zwilnik	4914.787	30823.917	9H 55M 38S	0
24	mar1kestrel	5135.297	30599.364	9H 55M 34S	0
25	feb28siria	6714.109	27146.678	9H 24M 20S	0
26	mar1siria	8873.114	26868.738	9H 55M 37S	0
27	feb28egeus	9065.477	13500.028	6H 16M 6S	0
28	mar1corvina	10898.545	24648.883	9H 55M 57S	210.202
29	mar1ptarmigan	12416.924	23319.178	9H 55M 34S	0
30	mar1fandango	20101.182	15638.594	9H 55M 38S	0
31	mar1dekanore	25614.651	14168.562	9H 55M 58S	7191.81