Vertigo: Automatic Performance-Setting for Linux

Krisztián Flautner
krisztian.flautner@arm.com
ARM Limited
110 Fulbourn Road
Cambridge, UK CB1 9NJ

Trevor Mudge
tnm@eecs.umich.edu
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122

Abstract

Combining high performance with low power consumption is becoming one of the primary objectives of processor designs. Instead of relying just on sleep mode for conserving power, an increasing number of processors take advantage of the fact that reducing the performance level and corresponding operating voltage of the CPU can yield quadratic decrease in energy use. However, performance reduction can only be beneficial if it is done transparently, without causing the software to miss its deadlines. In this paper, we describe the implementation and performance-setting algorithms used in Vertigo, our power management extensions for Linux. Vertigo makes its decisions automatically, without any application-specific involvement. We describe how a hierarchy of performance-setting algorithms, each specialized for different workload characteristics, can be used for controlling the processor's performance. The algorithms operate independently from one another and can be dynamically configured. As a basis for comparison with conventional algorithms, we contrast measurements made on a Transmeta Crusoe-based computer using its built-in LongRun power manager with Vertigo running on the same system. We show that unlike conventional interval-based algorithms like LongRun, Vertigo is successful at focusing in on a small range of performance levels that are sufficient to meet an application's deadlines. When playing MPEG movies, this behaviour translates into a 11%-35% reduction of mean performance level over LongRun, without any negative impact on the framerate. The performance reduction can in turn yield significant power savings.

1. Introduction

Power considerations are increasingly driving processor designs from embedded computers to servers. Perhaps the most apparent need for low-power processors is for mobile communication and PDA devices. These devices are battery operated, have small form factors and are increasingly taking up computational tasks that in the past have been performed by desktop computers. The next generation 3G mobile phones promise always-on connections, high-bandwidth mobile data access, video-on-demand services, video conferencing and the convergence of today's multiple standalone devices—MP3 player, game machine, camera, GPS, even the wallet—into a single device. This requires processors that are capable of high performance and modest power consumption. Moreover, to be power efficient, the processors for the next generation communicator need to take advantage of the highly variable performance requirements of the applications they are likely to run. For example an MPEG video player requires about an order of magnitude higher performance than an MP3 audio player but optimizing the processor to always run at the level that accommodates the video player would be wasteful.

Dynamic Voltage Scaling (DVS) exploits the fact that the peak frequency of a processor implemented in CMOS is proportional to the supply voltage, while the amount of dynamic energy required for a given workload is proportional to the square of the processor's supply voltage [12]. Running the processor slower means that the voltage level can also be lowered, yielding a quadratic reduction in energy consumption, at the cost of increased run time. The key to making use of this trade-off are performance-setting algorithms that aim to reduce the processor's performance level only when it is not critical to meeting the software's deadlines. The key observation is that often the processor is running too fast. For example, it is pointless from a quality-of-service perspective to decode the 30 frames of a video in half a second, when the software is only required to display those frames during a 1 second interval. Completing a task before its deadline is inefficient use of energy [6].

While dynamic power currently accounts for the greatest fraction of a processor's power consumption, static power consumption, which results from the leakage current in CMOS devices, is rapidly increasing. Figure 1 shows the power consumption of recent processors, along with the projected trends for the

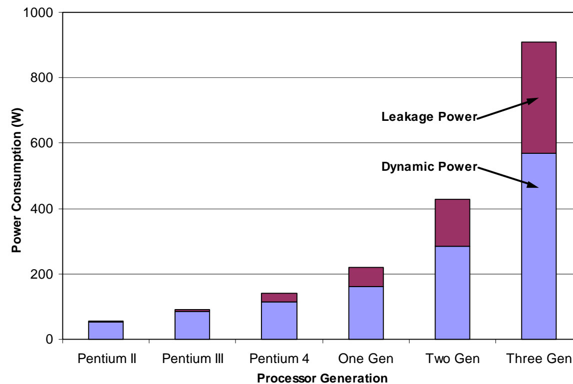


FIGURE 1. Projected increase of processor power consumption given current trends

next three processor generations. The estimates were generated based on projected transistor counts and circuit assumptions based on the ITRS roadmap [9]. If left unchecked, in a 0.07 micron process, leakage power could become comparable to the amount of dynamic power [2]. We expect leakage's proportion of total power to be over 30% in the near future. The processor's share of total power consumption in the system is also increasing, although heat management and power supply considerations provide an eventual limit to its growth. Similarly to dynamic power, leakage can also be substantially reduced if the processor does not always have to operate at its peak performance level. One technique for accomplishing this is adaptive reverse body biasing (ABB) [11], which combined with dynamic voltage scaling can yield substantial reduction in both leakage and dynamic power consumption. The pertinent point for this paper with respect to DVS and ABB is that lowering speed results in better than linear energy savings. Vertigo provides the main lever for controlling both of these techniques by providing an estimate for the necessary performance level of the processor.

Most mobile processors on the market today already support some form of voltage scaling; Intel calls its version of this technology SpeedStep [8]. However, due to the lack of built-in performance-setting policies in current operating systems, the computers based on these chips use a simple approach that is driven not by the workload but by the usage model: when the notebook computer is plugged in a power outlet the processor runs at a higher speed, when running on batteries, it is switched to a more power efficient but slower mode. Transmeta's Crusoe processor sidesteps this problem by building the power management policy into the processor's firmware and consequently not requiring any changes and additions to the operating systems that run on it [20]. Unlike on more conventional processors, the power management policy can be implemented on the Crusoe relatively easily because it already has a hidden software layer that performs dynamic binary translation and optimizations. However, it is currently an open question—one that we address in this paper—how effectively a policy implemented at such a low level in the software hierarchy can perform.

Research into performance-setting algorithms can be broadly divided into two categories: ones that use information about task deadlines in real-time kernels to guide the performance-setting decisions of the processor [11][13][15][19][16][17], and others that seek to derive deadlines automatically [3][6][14][21]. Our work falls into the latter category. Previously, we presented a mechanism for automatically classifying machine utilization into different types of episodes [4] and automatically assigning deadlines to them [3]. Deadline and classification information is derived from communication patterns between the executing tasks based on observations in the OS kernel. Vertigo is built on the ideas that were described in our previous papers and moves these techniques out of the simulator onto actual hardware.

Our performance-setting algorithms, described in Section 2, compare favorably to previous interval-based algorithms. The key difference in our approach is that by moving the algorithms into the OS kernel, they have access to a richer set of data for predictions. Moreover, the multiple concurrently-running algorithms in the system ensure that they do not all have to be optimal in all possible circumstances. This allows at least some of the algorithms to be less concerned about the worst case. Figure 2 illustrates the fraction of time spent at each of the processor's four performance levels (300, 400, 500, and 600 Mhz) using the built-in LongRun power manager in contrast with Vertigo during runs of two MPEG movies. While the playback quality of the different runs were identical, the main difference between the results is that Vertigo spends significantly more time at less than peak performance than LongRun. During the first movie, Vertigo switches mostly between two performance levels: the machine's minimum 300 Mhz and 400 Mhz,

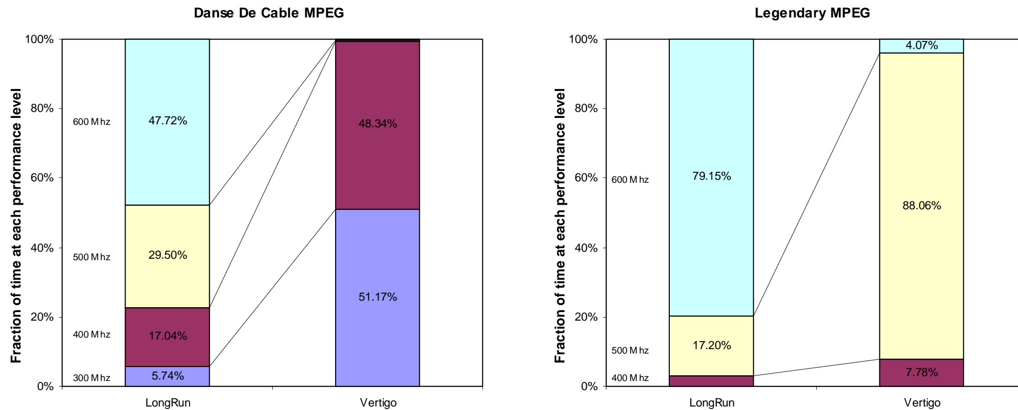


FIGURE 2. MPEG video playback LongRun vs. Vertigo

while during the second, it settles on the processor’s third performance level at 500 Mhz. LongRun, on the other hand, during both movies chooses the machine’s peak performance setting for the dominant portion of execution time.

Vertigo is a set of kernel modules and patches that hook into the Linux kernel to monitor program execution and to control the speed and voltage level of the processor (Figure 3). One of the main design objectives of this system has been to be only minimally intrusive into the host operating system. Vertigo coexists with the existing scheduler, system calls, and power manager (which controls the sleep and awake modes of the processor), however it needs certain hooks within these subsystems to operate. A unique feature of Vertigo is that instead of a single performance-setting algorithm, it allows the composition of multiple algorithms, all specializing in different kinds of run-time situations. The most applicable to a given condition is chosen at run-time. The different performance-setting policies are coordinated by the core module, which connects to the hooks in the kernel and provides shared functionality to the policies. The shared functionality includes an abstraction for setting the processor’s performance level, measuring and estimating work and a low-overhead soft timer implementation built on timestamp counters that provides sub-millisecond resolution. Implementation issues are discussed in Section 3.

Instead of estimating the potential energy savings resulting from our techniques, we use raw performance levels as the metric of interest in this paper. The correlation between performance levels and dynamic power consumption has been clearly established in the literature [3][11][12][13][17]. We believe that performance-setting techniques are applicable more broadly than just for controlling dynamic voltage scaling and that they will also be useful for controlling leakage-power reduction techniques in the near future. However, process details for useful estimates of energy are not yet available, and current predictions are likely to be inaccurate. Evaluations of our algorithms are presented in Section 4.

The main contributions of this paper are a set of kernel-level algorithms for performance-setting under Linux, a technique for coordinating multiple algorithms, a description of the Vertigo performance-setting framework, an evaluation of our algorithms on a Crusoe-based hardware platform, and a technique for measuring and contrasting our results with the processor’s built-in power manager. While Vertigo’s per-

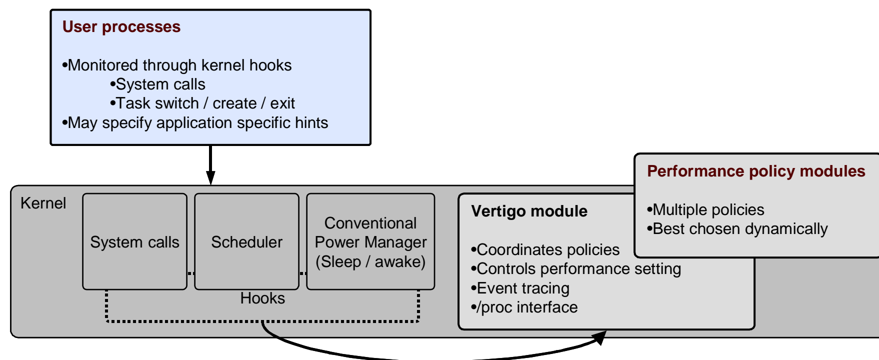


FIGURE 3. Vertigo architecture

spectives-based algorithm is a new addition, the interactive algorithm has been described in our previous work and evaluated on a simulator.

2. Performance-setting algorithms

Unlike previous approaches, Vertigo includes multiple performance-setting algorithms that are coordinated to find the best estimate for the necessary performance level. The various algorithms are organized into a decision hierarchy, where algorithms closer to the top have the right to override the choices made at lower levels. Currently we have three levels on the stack:

- *At the top:* an algorithm for automatically quantifying the performance requirements of interactive applications and which ensures that the user experience does not suffer. This algorithm is based on our previous one described in [3].
- *In the middle:* an application specific layer, where DVS-aware applications can submit information about their performance requirements.
- *At the bottom:* an algorithm that attempts to estimate the future utilization of the processor based on past information. This *perspectives-based* algorithm differs from previous interval-based algorithms in that it derives a utilization estimate for each task separately and adjusts the size of the utilization-history window on a per-task basis. Moreover, since the algorithm in the top layer ensures the high quality of interactive performance, the baseline algorithm does not have to be conservative about the size of the utilization-history window, the consideration of which has led to inefficient algorithms for even simple workloads (e.g. MPEG playback) in the past [7].

In this paper our focus is on the *interactive* algorithm at the top of the stack and the *perspectives-based* algorithm at the bottom. The application-specific layer is currently only used for debugging: we have instrumented certain applications such as the X server and our mpeg player to submit application specific information to Vertigo (through a system call) and then this information can be used to correlate Vertigo's activities with that of the applications.

2.1 Keeping track of work

The main concept used in our performance-setting algorithms is the full-speed equivalent work done during an interval. This measure can be used to estimate how long a given workload would take running at the peak performance of a processor. On the Crusoe, the full-speed equivalent work estimate is computed by the formula in Equation 1:

$$Work_{fse} = \sum_{i=1}^{n} t_i pf_i \qquad \text{(EQ 1)}$$

Where i refers to the n different performance levels during a given interval with the corresponding amount of non-idle time spent at that performance level (t_i) in seconds and frequencies (pf_i) specified as a fraction of peak performance. On a system where the count rate of timestamp counter varies with the speed of the processor, the full-speed equivalent speed would be computed differently. For example, if the timestamp counters count at the current rate of the processor, $Work_{fse}$ would simply be given as the difference between the value of the timestamp counter at the beginning and end of an interval.

2.2 A perspectives-based algorithm

At the lowest level in the policy stack is an algorithm that aims to derive a ballpark guess for the necessary performance level of the processor. It need not be completely accurate, since the assumption is that algorithms at higher positions on the policy stack will override its decisions when necessary. We refer to this algorithm as *perspectives-based*, since it computes performance predictions from the perspective of each task and uses the combined result to control the performance-setting of the processor. This algorithm differs from previous interval-based algorithms in that it derives a utilization estimate for each task separately and adjusts the size of the utilization-history window on a per-task basis.

Our insight is that individual tasks (or groups of tasks) often have discernible utilization periods at the task level, which can be obscured if all tasks are observed in the aggregate. We use each task's observed period for recomputing per-task exponentially decaying averages of the work done during the period and its deadlines. While previous interval-based performance-setting techniques also use exponentially decaying averages, they use it globally and with a fix period—the global utilization prediction is updated every 10ms-50ms—which often causes the predictions to oscillate between two performance levels. Their problem is that since a single algorithm must accurately set the performance level in all cases, they cannot wait

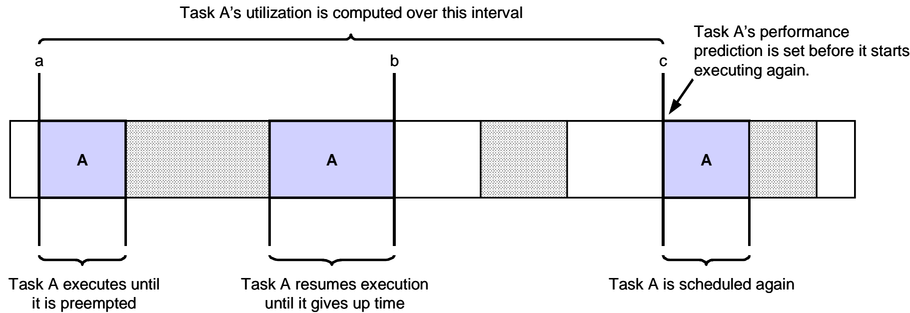


FIGURE 4. Measuring the utilization for task A

long enough to smooth out the performance prediction without unduly impacting the interactive performance. Our current technique uses a simple heuristic for finding a task's period: the algorithm tracks the distance from when a task starts executing, through points when it is preempted and eventually runs out of work (gives up time on its own), until the next time it is rescheduled. We have experimented with more complicated techniques for finding a task's period, such as tracking communications between them and tracking system calls [3], however we found that this simpler strategy works sufficiently well.

Figure 4 illustrates the execution of a hypothetical workload on the processor. At *point a* task A starts execution and the per-task data structures are initialized with four pieces of information: the current state of the work counter, the current state of the idle time counter, the current time, and a *run bit* indicating that the task has started running. The counters are used to compute the task's utilization and subsequently its performance requirements—see Section 3.2 for more information about how these are used. When the task is preempted, the task's run bit is left as-is, indicating that the task still has work left over. When task A gets scheduled again, it runs until it gives up time willingly (runs to completion before its schedule quantum expires or calls a system call that yields the processor to another task) at *point b* and its run bit is cleared. At *point c*, when task A is rescheduled, the cleared state of the run bit indicates that there is enough information for computing the task's performance requirements and setting the processor's performance level accordingly. At point c, $Work_{fse}$ is computed for the range between *point a* and *point c* and a future work estimate is derived based on this value (Equation 2):

$$WorkEstimate_{new} = \frac{k \times WorkEstimate_{old} + Work_{fse}}{k+1} \quad \text{(EQ 2)}$$

A separate exponentially decaying average is maintained to keep track of the deadlines of each interval, where the deadline is computed as $Work_{fse} + Idle$, where *Idle* specifies the amount of time during the interval between *points a* and *c* (Equation 3):

$$DeadlineEstimate_{new} = \frac{k \times DeadlineEstimate_{old} + Work_{fse} + Idle}{k+1} \quad \text{(EQ 3)}$$

Given these two values the performance-level prediction is computed as follows:

$$Perf = \frac{WorkEstimate}{DeadlineEstimate} \quad \text{(EQ 4)}$$

By keeping track of the work and deadline predictions separately, the performance predictions are weighted by the length of the interval over which the work estimates were measured. Note that unlike previous approaches, in this algorithm the performance predictions are used directly to set the machine's performance level, not indirectly to scale the processor's performance level up or down by an arbitrary amount [7]. Similarly to the data presented in [14], we found that small weight values for k work well, and used $k=3$ in our measurements.

As a result of our strategy, work estimates for each task are recomputed on a varying interval with a mean of around 50-150ms (depending on workload), however, as a result of multiple tasks running in the system, there is actually a refinement of the work estimate every 5ms to 10ms. One pitfall of the perspectives-based algorithm is that if there is a new non-interactive, CPU-bound task that gets started on an idle system, and that task utilizes the processor without being preempted for a long duration of time, there might be significant latency incurred in responding to the load. To guard against this situation, we put a limit on the non-preempted duration over which the work estimate is computed. If a task does not yield the processor for 100ms, its work estimate is recomputed. The 100ms value was selected based on two observa-

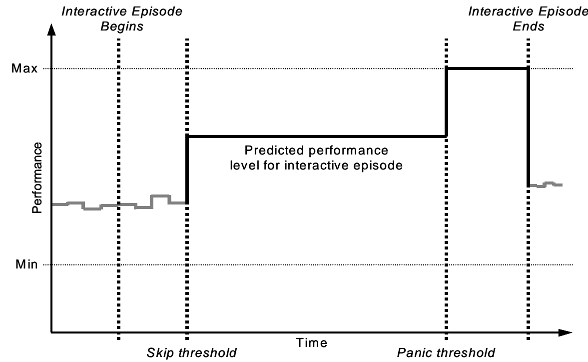


FIGURE 5. Performance-setting for interactive episodes

tions: a separate algorithm for interactive applications ensures that they meet a more stringent deadline, and that the only class of applications affected by the choice of the 100ms limit are the computationally intensive batch jobs (such as compilation) which are likely to run for seconds or minutes, and where an extra tenth of a second of execution time is unlikely to be significant.

2.3 Maintaining the quality of interactive applications

Our strategy for ensuring good interactive performance relies on finding the parts of execution that directly impact the user experience and ensuring that these episodes complete without undue delay. We use a relatively simple technique for automatically isolating interactive episodes that relies on monitoring communication from the X server (or other control task in charge of user interaction) and tracking the execution of the tasks that get triggered as a result. The technique used in Vertigo is based on our previous descriptions in [3] and [4]. A summary follows below:

The beginning of an interactive episode is initiated by the user and is signified by a GUI event, such as pressing a mouse button or a key on the keyboard. As a result of such an event, the GUI controller (X server in our case) dispatches a message to the task that is responsible for handling the event. By monitoring the appropriate system calls (various versions of read, write, and select), Vertigo can automatically detect the beginning of an interactive episode. When the episode starts, both the GUI controller and the task that is the receiver of the message are marked as being in an interactive episode. If tasks of an interactive episode communicate with unmarked tasks, then the as yet unmarked tasks are also marked. During this process, Vertigo keeps track of how many of the marked tasks have been preempted. The end of the episode is reached when that number is zero.

Figure 5 illustrates the strategy for setting the performance level during an interactive episode. At its beginning, the algorithm waits for a specific amount of time, determined by the *skip threshold* before transitioning to the predicted performance level. We observed that the vast majority of interactive episodes are so short (sub millisecond) as to not warrant any special consideration. These short episodes are the results of echoing key presses to the window or moving the mouse across the screen and redrawing small rectangles. We found that a skip threshold of 5ms is good value for filtering short episodes without adversely impacting the worst case. If the episode exceeds the skip threshold, the performance level is switched to the interactive performance prediction. Similarly to the perspectives-based algorithm, the prediction for the interactive episodes is computed as the exponentially decaying average of the correct settings of past interactive episodes. To bound the worst case impact on the user experience, if the interactive episode does not finish before reaching the *panic threshold*, the processor's performance is ramped up to its maximum. At the end of the interactive episode, the algorithm computes what the correct performance-setting for the episode should have been and this value is incorporated into the exponentially moving average for future predictions. An added optimization is that if the panic threshold was reached during an episode, the moving average is rescaled so that the last performance level gets incorporated with a higher weight (k=1 is used instead of k=3).

The performance prediction is computed for all episodes that are longer than the *skip threshold*. If the episode was also longer than *perception threshold*, then the performance requirement is set to 100%. The perception threshold describes a cut-off point, under which events appear to happen instantaneously for the user. Thus, completing these events any faster would not have any perceptible impact on the observer [1]. While the exact value of the perception threshold is dependent on the user and the type of task being accomplished, a value of 50ms is commonly used [1][3][14]. Equation 5 is used for computing the performance requirements of episodes that are shorter than the perception threshold.

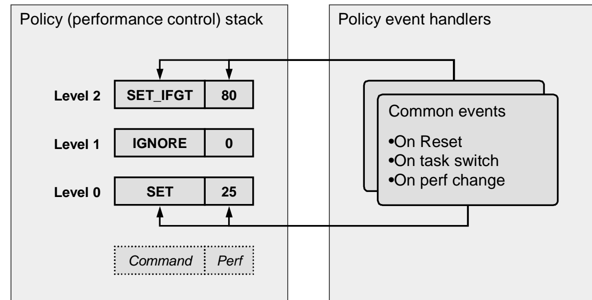


FIGURE 6. Policy stack

$$Perf = \frac{Work_{fse}}{PerceptionThreshold} \quad \text{(EQ 5)}$$

Where the full-speed equivalent work is measured from the beginning of the interactive episode. The algorithm in Vertigo differs on the following points from our previous implementations:

- Finding the end of an interactive episode has been simplified. We found that the higher accuracy inherent in our previous implementations was unnecessary.
- The *panic threshold* has been statically set to 50ms. In our previous implementations the threshold varies dynamically depending on the rate that work is getting done (i.e. the performance level during the interactive episode). While this idea might still prove to be useful on machines with a wider range of performance levels, we saw no perceptible difference on our evaluation machine which has a performance range of 300Mhz to 600Mhz.
- There is only a singe prediction for the necessary performance level for an interactive episode in the system. In our previous technique, we used a per-task value depending on which task initiated the episode.

3. Implementation issues

3.1 Policy stack

The policy stack (Figure 6) is the mechanism for capturing the performance requests from multiple independent performance-setting algorithms and combining them into a single global decision. The different policies are not aware of their positions in the hierarchy and can base their performance decisions on any interesting event in the system. When a policy requests a performance level it submits a command along with its desired performance. The command specifies how the requested performance should be combined with requests from lower levels on the stack: it can specify to ignore (*IGNORE*) the request at the current level, to force (*SET*) a performance level without regard to any requests from below, or to only set a performance level if the request is greater than anything below (*SET_IFGT*). When a new performance level request arrives, then the commands on the stack are evaluated bottom up to compute the new global performance. Using this system performance requests can be submitted any time and a new result computed without having to invoke all the performance-setting policies.

While policies can be triggered by any event in the system and they may submit a new performance request at any time, there are a set of common events that all policies tend to be interested in. On these events, instead of recomputing the global performance level each time a policy modifies its entry, the performance level is computed only once after all interested policies' event handlers have been invoked. Currently the set of common events are: reset, task switch, task create, and performance change. The performance change event is a notification, which allows each policy to be aware of the current performance level of the processor, it does not usually alter the performance requests on the stack.

3.2 Work tracking

Our algorithms use the processor's utilization history over a given interval to estimate the necessary speed of the processor in the future. The idea is to maximize the busy time of the processor by slowing it down to the appropriate performance level. To aid this, Vertigo provides an abstraction for tracking the work done during a given time interval which takes performance changes and idle time into account.

	CPU bound loop			MPEG video		
	400 Mhz	500 Mhz	600 Mhz	400 Mhz	500 Mhz	600 Mhz
300 Mhz	-0.3%	-0.4%	-0.3%	7.1%	13.5%	19.4%
400 Mhz		-0.1%	0.0%		6.9%	13.3%
500 Mhz			0.1%			6.8%

TABLE 1. Scaling error of work predictions

To get a work measurement, a policy needs to allocate a `vertigo_work` struct and call the `vertigo_work_start` function at the beginning, and the `vertigo_work_stop` function at the end of the interval. During the measurements the contents of the structs are updated automatically when idle time and performance changes occur in the system. Aside from the convenience that this abstraction provides for policy writers, it is also designed to simplify porting of Vertigo (and associated policies) to different hardware architectures. One major difference between different platforms is how time is measured. Many architectures provide a low overhead way of counting cycles through timestamp counters, others may only provide externally programmable timer interrupts for the user. Moreover, even when timestamp counters are provided, they do not always measure the same things. On current Intel Pentium and ARM processors the timestamp counters count cycles—the rate varies depending on the speed of the processor—and the counter stops counting when the processor transitions into sleep mode. The Crusoe's implementation of the timestamp counter measures time: it always counts the cycles at the peak rate of the processor and continues to do so even when the processor is asleep. Ideally a system would include both types of counters, however, Vertigo can be made to work with either approach.

One aspect of systems the work estimate does not yet take into account is that a workload running at half of peak performance does not necessarily run twice as long as the original. One reason for this may be that as the core is slowed down, the memory system is not, thus the core to memory performance ratio improves in memory's favour [7]. Table 1 shows our measurements which show the difference between the expected and measured lengths of the workloads based on runs at 300, 400, and 500 Mhz settings of the processor. On the CPU bound loop, the difference between the predictions and actual measurements are in the noise, while on the MPEG workload, there is about a 6%-7% inaccuracy increase per 100 Mhz step. While the maximum inaccuracy on these workloads is less than 20%, as the range of minimum to maximum performance increases, along with a reduction in the range of each performance step, a more accurate work estimator might be necessary. A possible solution could be to take the instruction mix of the workload into account by the use of performance monitoring counters that keep track of significant events such as external memory accesses.

3.3 Monitoring, timers and tracing

One design goal of Vertigo has been to make it as autonomous from other units in the kernel as possible. Another design goal emerged as we selected the platform for our experiments. The Transmeta Crusoe processor includes its own performance-setting algorithm and we wished to compare the two approaches. The first requirement has already yielded a relatively unobtrusive design, the second focused us on turning the existing functionality into a passive observation platform.

An example of how Vertigo has been made unobtrusive is the way timers are handled. Vertigo provides a sub-millisecond resolution timer, without changing the way Linux's built-in 10ms resolution timer works. This is accomplished by piggybacking the timer dispatch routine, which checks for timer events onto often-executed parts of the kernel, such as the scheduler and system calls. Since Vertigo already intercepts certain system calls to find interactive episodes and is also invoked on every task switch, it was straight-forward to add a few instructions to these hooks to manage timer dispatches. Each hook is augmented with a read of the timestamp counter, a comparison against the next timer event's time stamp and a branch to the timer dispatch routine upon success. In practice we found that this strategy yields a timer with sub-millisecond accuracy, while its worst case resolution is bound by the scheduler's time quantum, which is 10ms (see Table 2). However, since the events that Vertigo is interested in measuring usually occur close to the timer triggers, this technique has adequate resolution. Another advantage is that since the soft-timers stop ticking when the processor is in sleep mode, the timer interrupts do not change the sleep characteristics of the running OS and applications.

All these features allowed us to develop, in addition to the active mode where Vertigo is in control, a passive mode, where the built-in LongRun power manager is in charge of performance-setting and Vertigo

Cost of an access to a timestamp counter	30-40 cycles
Mean distance between timer checks	~0.1 ms
Timer accuracy	~1 ms
Avg. timer check and dispatch duration (incl. possible execution of an event handler)	100-150 cycles

TABLE 2. Timer statistics

is simply an observer of the execution and performance changes. Monitoring the performance changes caused by LongRun is accomplished similarly to the timer dispatch routine. Vertigo periodically reads the performance level of the processor through a machine specific register (msr) and compares the result to its previous value. If they are different, then the change is logged in a buffer. Vertigo includes a tracing mechanism that retains a log of significant events in a kernel buffer that is exposed through the proc filesystem. This log includes performance-level requests from the different policies, task preemptions, task ids, and the performance levels of the processor. Another feature of this technique is that it allows us to compare LongRun and Vertigo during the same run: LongRun is in control of performance-setting while Vertigo outputs the decisions that it would have made on the same workload. We use this technique to contrast the differences between unrepeatable runs of interactive benchmarks between the two policies (see Section 4.2).

To get a better feel for the overhead of using our measurement and performance-setting techniques, Vertigo was instrumented with markers that keep track of the time spent in Vertigo code at run-time. While the run-time overhead on a Pentium II is less than 0.1% to 0.5%, on the Transmeta Crusoe it is between 1% and 4%. Further measurements in virtual machines such as VMWare and user-mode-linux (UML) confirmed that the overhead can be significantly higher in virtual machines than on traditional processor architectures. We believe that Vertigo's overhead could be reduced further since we as yet use unoptimized algorithms.

4. Evaluation

Our measurements were performed on a Sony Vaio PCG-C1VN notebook computer using the Transmeta Crusoe 5600 processor running at 300Mhz to 600Mhz with 100Mhz steps. The operating system used is Mandrake 7.2 with a modified version of the Linux 2.4.4-ac18 kernel. The workloads used in the evaluation are the following: Plaympeg SDL MPEG player library [18], Acrobat Reader for rendering PDF files, Emacs for text editing, Netscape Mail and News 4.7 for news reading, Konqueror 1.9.8 for web browsing, and Xwelltris 1.0.0 as a 3D tetris-like game. The interactive shell commands benchmark is a record of a user doing miscellaneous shell operations during a span of about 30 minutes. To avoid variability due to the Crusoe's dynamic translation engine, most benchmarks were run at least twice and data was used from the last run.

4.1 Multimedia

MPEG video playback poses a difficult challenge for performance-setting algorithms. While the algorithm puts a periodic load on the system, the performance requirements can vary depending on the frame's type. Thus, if a performance-setting algorithm looks at too-long of a past history for predicting future requirements, it can miss the deadlines for more computationally intensive frames. On the other hand, if the algorithm looks at only a short interval, then it will not settle on a single performance value but oscillate between multiple settings. This issue is exposed in [7], where the authors show that no heuristic algorithm they looked at could successfully settle on the single performance level that would have been adequate for the entire workload. Our observations of LongRun confirm this behaviour.

Vertigo deals with this problem by relying on the interactive performance-setting algorithm at the top of the hierarchy to bound worst-case responsiveness (in this case frame rate) and allowing the more conventional interval-based algorithm at the bottom of the hierarchy to take a longer-term view. Table 3 shows measurements about the plaympeg video player [18] playing a variety of MPEG videos. Some of the internal variables of the video player have been exposed to provide information about how the player is affected as the result of changing the processor's performance levels during execution. These figures are shown in the *MPEG decode* column of the table. The *Ahead* variable measures how close the end of each frame's decoding is to its deadline. It is expressed as cumulative seconds during the playback of each video. For power efficiency, this number should be as close to zero as possible, although the slowest performance level of the processor puts a limit on how much its value can be reduced. The *Exactly on time*

		Execution statistics			MPEG decode	
		Length (s)	Idle	Sleep	Ahead (s)	Exactly on time
Danse De Cable	**LongRun**	247.1	54%	23%	148.10	6
320x160 +audio	**Vertigo**		27%	4%	68.74	1012
Legendary	**LongRun**	19.4	33%	13%	7.20	19
352x240 +audio	**Vertigo**		24%	7%	4.79	65
Red's Nightmare	**LongRun**	49.1	48%	36%	26.31	5
320x240	**Vertigo**		32%	13%	16.53	74
Red's Nightmare	**LongRun**	49.3	22%	15%	12.48	87
480x360	**Vertigo**		18%	11%	8.17	139
Roadkill Turtle	**LongRun**	121.3	46%	19%	64.93	5
304x240 +audio	**Vertigo**		25%	4%	33.34	237
Sentinel	**LongRun**	35.6	28%	10%	11.05	80
320x240 +audio	**Vertigo**		19%	5%	6.32	231
SpecialOps	**LongRun**	60.8	30%	11%	19.01	129
320x240 +audio	**Vertigo**		20%	5%	12.67	305

TABLE 3. Application-level statistics about the plaympeg benchmark playing various movies

field specifies the number of frames that met their deadlines exactly. The more frames are on time, the closer the performance-setting algorithm is to the theoretical optimum. The data in the *Execution Statistics* column is collected by Vertigo's monitoring subsystem. To collect information about LongRun, Vertigo was used in passive mode to gather a trace of performance changes without controlling the processor's performance level. The difference between the *Idle* and *Sleep* fields are that the first corresponds to the fraction of time spent in the kernel's idle loop—possibly doing housekeeping chores or just spinning—while the latter shows the fraction of time the processor actually spends in a low-power sleep mode.

Table 4 provides statistics about the processor's performance levels during the runs of each workload. The fraction of time at each performance level is computed as a proportion of total non-idle time during the run of the workload. The *Mean perf level* column specifies the average performance levels (as the percentage of peak performance) during the execution of each workload. Since, in all cases, the mean performance level for each workload was lower using Vertigo, the last column specifies the amount of reduction. The playback quality for each pair of workloads was the same: same frame rate and no dropped frames. Our results show that Vertigo is more accurately able to predict the necessary performance level than LongRun. The increased accuracy results in a 11% to 35% reduction of the average performance levels of the processor during the benchmarks' execution. Since the amount of work between runs of a workload stays the same, the lower average performance level implies reduced idle and sleep times when

	LongRun					Vertigo					Mean performance reduction over LongRun
	Fraction of time at each performance level (Mhz)				Mean perf level	Fraction of time at each performance level (Mhz)				Mean perf level	
	300	400	500	600		300	400	500	600		
Danse De Cable	6%	19%	33%	**54%**	89%	**51%**	48%	0%	0%	59%	**34%**
Legendary	0%	3%	17%	**79%**	96%	0%	8%	**88%**	4%	82%	**15%**
Red's Nightmare small	11%	**35%**	**35%**	19%	80%	**95%**	2%	0%	3%	52%	**35%**
Red's Nightmare big	0%	5%	21%	**74%**	95%	0%	0%	**90%**	10%	85%	**11%**
Roadkill Turtle	3%	10%	23%	**64%**	92%	1%	**97%**	1%	0%	66%	**28%**
Sentinel	0%	0%	14%	**86%**	97%	0%	0%	**93%**	7%	84%	**13%**
SpecialOps	1%	2%	14%	**83%**	96%	0%	2%	**93%**	4%	83%	**14%**

TABLE 4. Performance levels during movie playback

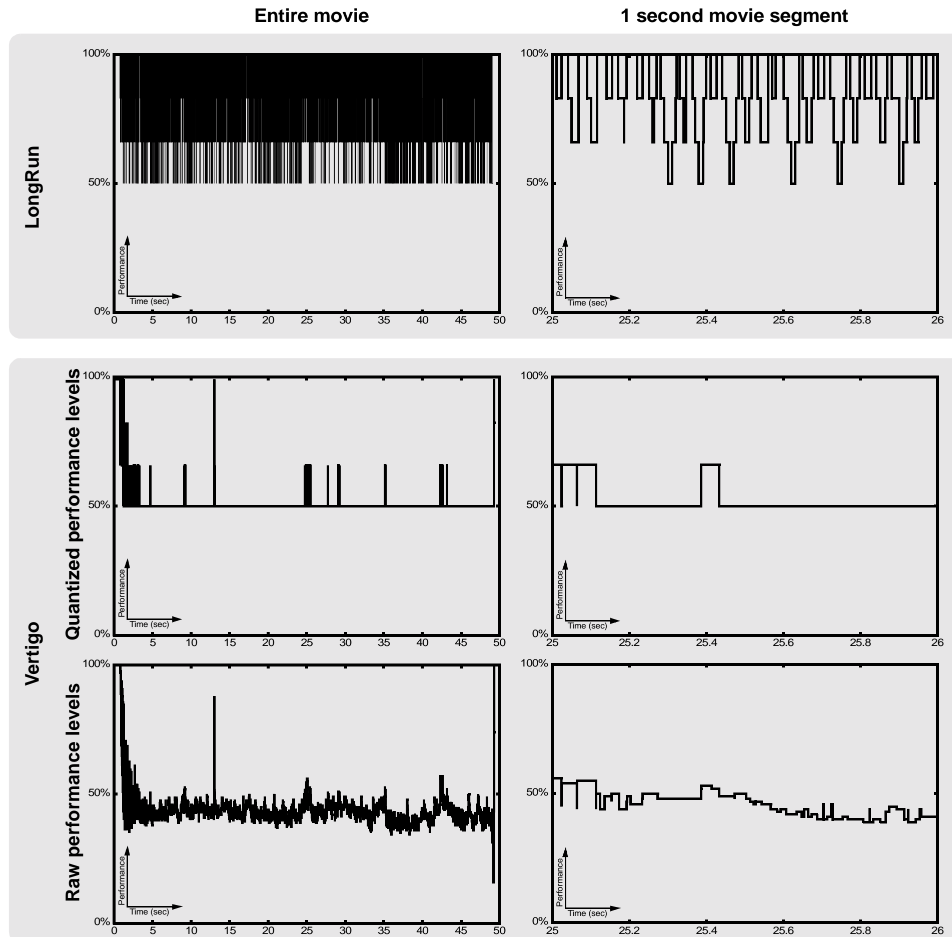
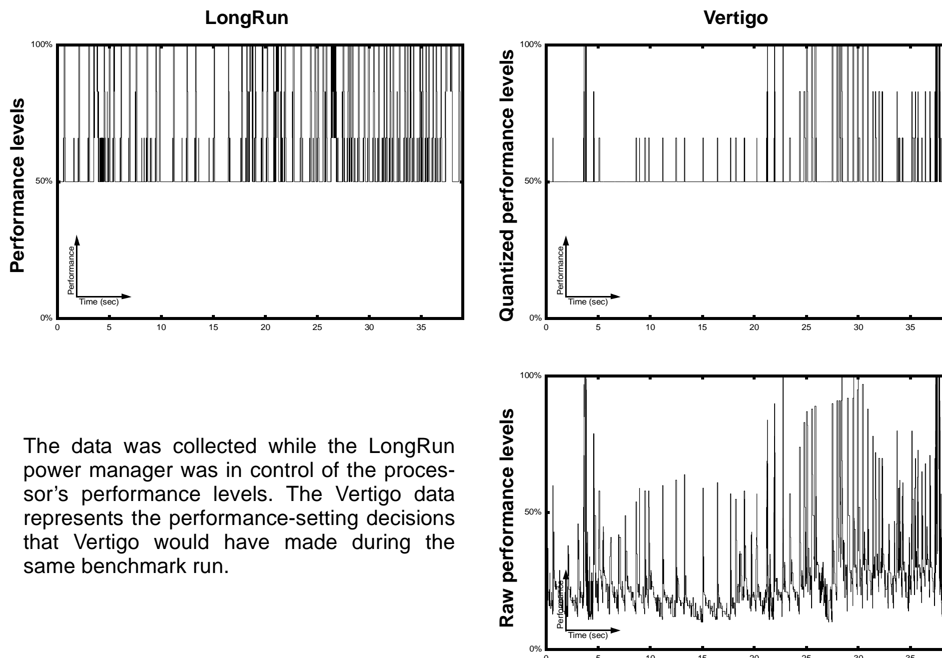


FIGURE 7. Performance-setting during MPEG playback of Red's Nightmare 320x240

Vertigo is enabled. This expectation is affirmed by our results. Similarly, the number of frames that exactly meet their deadlines increases when Vertigo is enabled and the cumulative amount of time when decode is ahead of its deadline is reduced. The median performance level (highlighted with bold in each column) also shows significant reductions. While on most benchmarks Vertigo settles on a single performance level below peak for the greatest fraction of execution time (>88%), LongRun usually chooses to run the processor at full throttle. The exception to this is the Danse De Cable workload, where Vertigo settles on the lowest two performance levels and switches between the two continuously. The reason for this behaviour is due to the specific performance levels on the Crusoe processor; Vertigo would have wanted to select a performance level which is only slightly higher than 300 Mhz and as the prediction fluctuates below and above that value, it is quantized to the closest two performance levels.

The biggest single difference between LongRun and Vertigo is that LongRun appears to be overcautious: it ramps up the performance level very quickly when it detects significant amounts of processor activity. Over all workloads, the average performance level with LongRun never gets below 80%, while Vertigo goes down as low as 52%. Vertigo is less cautious but responds quickly when the quality of service appears to have been compromised. Since LongRun does not have any information about the interactive performance, it is forced to act conservatively on a shorter time frame, which leads to inefficiencies. Figure 7 provides some qualitative insight into the characteristics of the two different performance-setting policies. LongRun keeps on ramping the performance level up and down in fast succession, while Vertigo stays close to a target performance level. The top row shows the processor's performance levels during a benchmark run with LongRun enabled and the bottom two rows show the same benchmark for Vertigo. The middle row shows the actual performance levels during execution, while the bottom row reflects the performance level that Vertigo would request on a processor that could run at arbitrary performance levels (given the same max. performance). Note that in some cases, Vertigo's desired performance levels are actually below the minimum that's achievable on the processor.



The data was collected while the LongRun power manager was in control of the processor's performance levels. The Vertigo data represents the performance-setting decisions that Vertigo would have made during the same benchmark run.

FIGURE 8. Performance-setting decisions during the Konqueror benchmark

4.2 Interactive workloads

Due to the difficulty in making interactive benchmark runs repeatable, interactive workloads are significantly harder to evaluate than the multimedia benchmarks. To get around this problem, we combined empirical measurements with a simple simulation technique. The idea is to run our benchmarks under the control of the native LongRun power manager and only engage Vertigo in passive mode, where it merely records the performance-setting decisions that it would have made but does not actually change the processor's performance levels. Figure 8 shows the performance data that was collected during a run of our measurements. The LongRun graph corresponds to the actual performance levels of the processor during the measurement, while the Vertigo graphs show the quantized (top right) and raw (bottom right) performance levels that Vertigo would have used had it been in control. Note that if Vertigo were in control, its performance-setting decisions would have had a different run-time impact from LongRun, thus the time axes on the Vertigo graphs are only approximations.

To get around the time-skew problem in our statistics, the passive Vertigo performance-level traces were postprocessed to take the impact of the increased execution times that would have resulted from the use of Vertigo instead of LongRun. Instead of looking at the entire performance-level trace, we chose to focus only on the interesting parts: the interactive episodes. As part of the interactive performance-setting algorithm in Vertigo, it includes a technique for finding durations of execution that have a direct impact on the user. This technique gives valid readings regardless of which algorithm is in control and is used to focus our measurements. Once the execution range for an interactive episode has been isolated, the full-speed equivalent work done during the episode is computed for both LongRun and Vertigo. Since during the measurement LongRun is in control of the CPU speed and it runs faster than Vertigo, the latter's episode duration must be lengthened. First, the remaining work is computed for Vertigo (Equation 6). Then, the algorithm computes how much the length of the interactive episode needs to be stretched—assuming that Vertigo continues to run at its predicted speed until reaching the panic threshold, at full-speed after that—and the statistics are adjusted accordingly.

$$Work_{VertigoRemaining} = Work_{LongRun} - Work_{Vertigo} \qquad \text{(EQ 6)}$$

We found that the results using this technique are close to what we observed on similar workloads (same benchmark but with slightly different interactive load) running with Vertigo active. However, when Vertigo is in control, the number of performance-setting decisions are reduced and are more accurate.

Figure 9 shows the statistics gathered using the above technique. Each graph contains two stacked columns, corresponding to the fraction of time spent in interactive episodes at each of the four performance levels supported in our computer. These performance levels—from bottom up—are from 300 Mhz

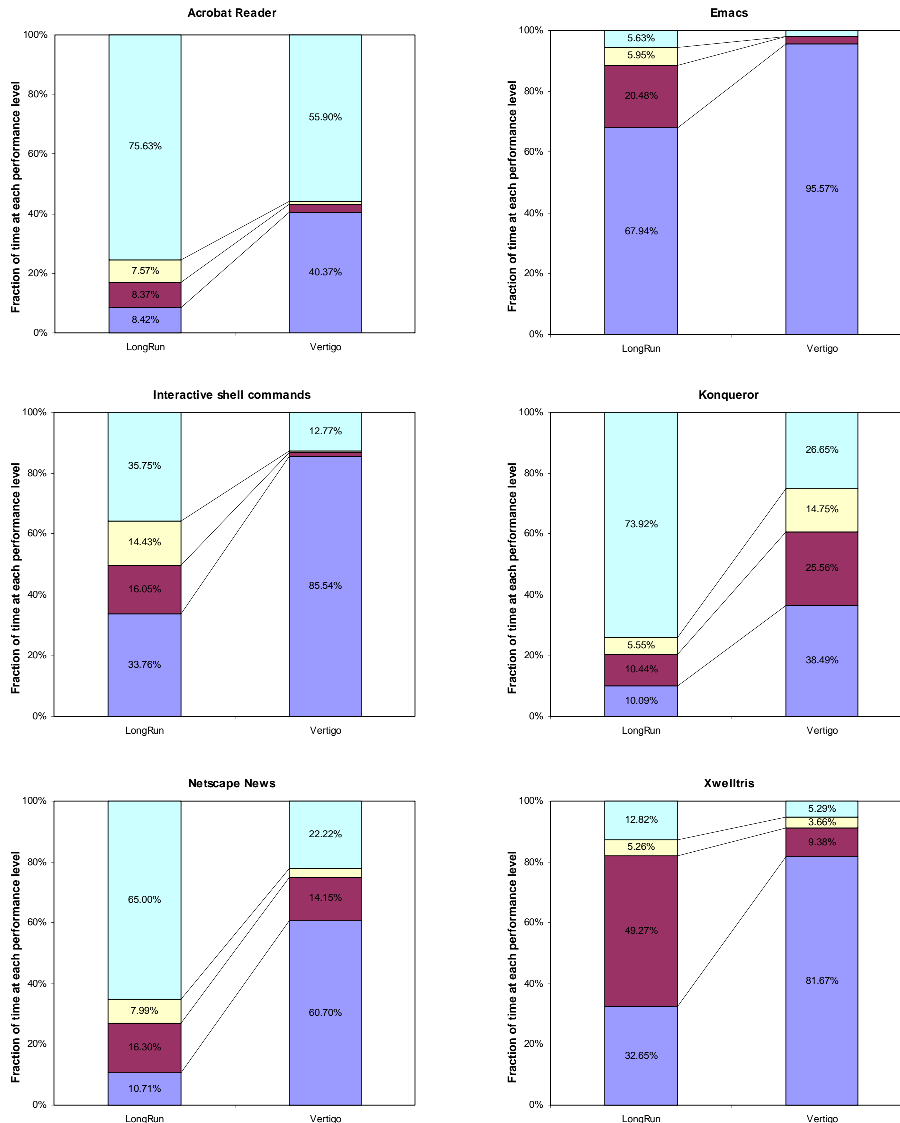


FIGURE 9. Fraction of time at different performance levels

to 600 Mhz at 100 Mhz increments. Even from a high level, it is apparent that Vertigo spends more time at lower performance levels than LongRun. On some benchmarks such as Emacs, there is hardly ever a need to go fast and the interactive deadlines are met while the machine stays at its lowest possible performance level. On the other end of the spectrum is Acrobat Reader, which exhibits bimodal behaviour: the processor either runs at its peak level or at its minimum. Even on this benchmark many of the interactive episodes can complete in time at the machine's minimum performance level, however when it comes to rendering the pages, the peak performance level of the processor is not sufficient to complete its deadlines under the user's perception threshold. Thus, upon encountering a sufficiently long interactive episode, Vertigo switches the machine's performance level to its peak. On the other hand, during the run of the Konqueror benchmark, Vertigo can take advantage of all four performance levels that are available on the machine. This is in contrast with LongRun's strategy which causes the processor to spend most of its time at the peak level.

5. Conclusions and future work

We have shown how two performance-setting policies implemented at different levels in the software hierarchy behave on a variety of multimedia and interactive workloads. We found that Transmeta's LongRun power manager, which is implemented in the processor's firmware, makes more conservative choices than our Vertigo algorithms, which are implemented in the Linux kernel. On a set of multimedia

benchmarks, the different design decisions amount to a 11%-35% average performance level reduction by Vertigo over LongRun. Being higher on the software stack allows Vertigo to make decisions based on a richer set of run-time information, which translates into increased accuracy. While the firmware approach was shown to be less accurate than an algorithm in the kernel, it does not diminish its usefulness. LongRun has the crucial advantage of being operating system agnostic. Perhaps one way to bridge the gap between low and high level implementations is to provide a baseline algorithm in firmware and expose an interface to the operating system to optionally refine performance-setting decisions. The policy stack in Vertigo can be viewed as the beginnings of a mechanism facilitating such design, where the bottom-most policy on the stack could actually be implemented in the processor's firmware.

We believe that aside from dynamic voltage scaling, performance-setting algorithms will be useful for controlling other power reduction techniques, such as adaptive body biasing. These circuit techniques cut down on the processor's leakage power consumption, which is an increasing fraction of total power as the feature sizes of transistors are reduced. While the power consumption of the processor is a significant concern, it only accounts for a fraction of the system's total power consumption. We are working on extending our techniques to managing the power of all the devices in an integrated system.

References

[1] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Publishers, 1983.

[2] A. Chandrakasan, W. Bowhill, F. Fox eds., *Design of High-Performance Microprocessor Circuits*. Piscataway, NJ: IEEE Press, 2001.

[3] K. Flautner, S. Reinhardt, and T. Mudge. Automatic Performance-Setting for Dynamic Voltage Scaling. *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM-7),* July *2001.*

[4] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), November 2000.*

[5] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP-17), December 1999.*

[6] K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. *Proceedings of the First International Conference on Mobile Computing and Networking*, November 1995.

[7] D. Grunwald, P. Levis, K. Farkas, C. B. Morrey III, and M. Neufeld. Policies for Dynamic Clock Scheduling. *Proceedings of the Fourth Symposium on Operating Systems Design & Implementation*, October 2000.

[8] Intel SpeedStep. http://support.intel.com/support/processors/mobile/pentiumiii/ss.htm

[9] ITRS roadmap. http://public.itrs.net

[10] A. Keshavarzi, S. Narendra, et. al., "Effectiveness of reverse body bias for leakage control in scaled dual Vt CMOS ICs," *Intl. Symp. on Low Power Electronics and Design*, 2001.

[11] C. M. Krishna and Y-H Lee. Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power Hard Real-Time Systems. *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000), 2000.*

[12] T. Mudge. Power: A First Class Architectural Design Constraint. IEEE Computer, vol. 34, no. 4, April 2001.

[13] T. Okuma, T. Ishihara, and H. Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. *Proceedings of the International Symposium on System Synthesis*, November 1999.

[14] T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. *Proceedings of International Symposium on Low Power Electronics and Design 1998*, pp. 76-81, June 1998.

[15] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the lpARM Microprocessor System. *Proceedings of the International Symposium on Low Power Electronics and Design 2000*, July 2000.

[16] P. Pillai and K. G. Shin. Real-time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. *Proceedings of the 18th Symposium on Operating System Principles*, October 2001.

[17] J. Pouwelse, K. Langendoen, and H. Sips. Voltage scaling on a low-power microprocessor. *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM-7),* July *2001.*

[18] SDL MPEG player library. http://www.lokigames.com/development/smpeg.php3

[19] Y. Shin and K. Choit. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. *Proceedings of the 36th Annual Design Automation Conference, 1999.*

[20] Transmeta Crusoe. http://www.transmeta.com/technology/index.html

[21] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. *Proceedings of the First Symposium of Operating Systems Design and Implementation*, November 1994.