AutoDVS: An Automatic, General-Purpose, Dynamic Clock Scheduling System for Hand-Held Devices

UCSB Technical Report 2005-04, March 1, 2005

Selim Gurun Chandra Krintz
Computer Science Department
University of California, Santa Barbara
{gurun,ckrintz}@cs.ucsb.edu

ABSTRACT

We present AutoDVS, a dynamic voltage scaling (DVS) system for the HP iPAQ hand-held computer. AutoDVS unifies existing DVS techniques into a single system that significantly reduces the power consumption of popular, general-purpose, iPAQ software. Moreover, it does so without degrading the user's experience perceivably. AutoDVS automatically infers periods of user interactivity and non-interactivity and applies different DVS policies to each period type. We have implemented AutoDVS as a freely-available, kernel-patch for Familiar Linux and the iPAQ Opie Window Manager. We evaluated AutoDVS using real user workloads of iPAQ software running alone and concurrently. AutoDVS decreases energy consumption by 30-70% for interactive programs, by up to 19% for soft real-time applications, and by 12% when these different program types run concurrently.

1. INTRODUCTION

Recent advances in embedded device technology have led to the proliferation of battery-powered devices and, in particular, hand-held personal digital assistants (PDAs) and web-enabled cellular phones. Worldwide, approximately 30 million PDAs are in use, and predictions indicate that PDA sales in the US will increase from 6.9 million to 17.1 million by 2007. Moreover, the market for smartphones is growing even faster, out-pacing sales of PDAs in Europe [16]. As a result of the popularity and improving capability of hand-held devices, users demand increasingly complex software for these devices. Moreover, users expect lighter devices with long battery lives.

Dynamic voltage scaling (DVS) is a technique that attempts to extend the battery life of hand-held devices without compromising system performance. DVS enables the CPU to be scaled dynamically according to varying program or workload demand [24, 19, 7, 8]. Next-generation CPUs for mobile computers, such as those produced by companies like Transmeta and Intel Corporation, will support a range of voltage and frequency levels that can be adjusted at runtime. Using such functionality, a DVS system must determine the best frequency level for optimal performance and battery lifetime.

To enable our empirical evaluation of and investigation into DVS techniques for hand-held devices, we implemented a DVS system called *AutoDVS* for the HP iPAQ hand-held [3] and its operating system (OS), Familiar Linux [4] AutoDVS is a set of open-source, OS and window manager extensions that performs DVS transparently, requiring no application support or programmer effort. AutoDVS is also general-purpose, i.e., it is effective for a range of popular iPAQ software, e.g., games, personal assistance applications, and multimedia programs.

AutoDVS couples a number of important extant approaches to DVS into a unifying system that reduces power consumption without negatively impacting the user's perception of system performance. As originally proposed in [5], and employed and extended in [14, 12], AutoDVS differentiates between different workload types: periods of user interaction with the device, periods when the device is idle, and computationally-intensive periods.

AutoDVS detects these different types of workload behaviors using two light-weight software sensors. AutoDVS then employs multiple DVS techniques to handle each workload type differently. AutoDVS considers both application-specific behaviors (for interactive periods) and system-wide behaviors (for non-interactive periods) to guide its CPU scaling decisions.

AutoDVS captures interactive periods by intercepting GUI events in the iPAQ window manager as is similar to the capture of X-Window mouse and keyboard events in [5]. AutoDVS captures all events including user input events and those that update windows, separately, for each application running in the system. This enables AutoDVS to predict more accurately the length of interactive periods given recent event histories. We employ a light-weight, highly accurate prediction utility called NWSLite that we developed in prior work [9], to predict interactive period length. AutoDVS maximizes the CPU voltage and frequency for the predicted duration of interactive periods – since presumably, the user is most sensitive to device performance during this time.

For non-interactive periods, AutoDVS employs two interval scheduling techniques. AutoDVS implements variations of PAST workload prediction [24] and Pering's hysteresis [19] to capture CPU intensive periods in the workload and to scale the CPU appropriately. AutoDVS also monitors Linux idle process statistics to identify opportunities to scale-down the CPU.

Our empirical evaluation indicates that AutoDVS can reduce power consumption of iPAQ hand-held workloads significantly without degrading system performance. We empirically evaluate AutoDVS for real iPAQ workloads that we collected from actual users. For repeatability, we play-back the workloads *in real time on the iPAQ*, and measure the impact of using AutoDVS and comparative ap-

proaches. We evaluate AutoDVS for single as well as concurrent workloads. As benchmarks, we use applications that are bundled with standard distributions of hand-held operating systems. Our results show that AutoDVS reduces energy consumption by 30-70% for interactive programs, by up to 19% for soft real-time applications, and by 12% when these different program types run concurrently. Moreover, it does so while maintaining very high program execution quality.

In summary, the contributions of this paper include:

- The design and implementation of AutoDVS, a lightweight Linux extension that automatically and transparently reduces power consumption without negatively impacting the user's perception of performance by coupling and extending known CPU scaling techniques effectively.
- A system that accurately predicts periods of interactivity and considers non-interactive periods separately to make CPU scaling decisions.
- An empirical evaluation that shows that AutoDVS successfully balances low power consumption and system performance. We evaluate AutoDVS using real workloads of popular iPAQ software that we collected from users of our devices.
- An open-source framework for the implementation, investigation, and empirical evaluation and comparison of DVS algorithms.

In the following section, we preset background on dynamic voltage scaling and on existing approaches to DVS that we incorporate into AutoDVS. We then describe the design and implementation of AutoDVS in Section 3. In Section 4, we present the empirical evaluation of our system and in Section 5, we conclude.

2. BACKGROUND

In modern CPUs, most energy is dissipated in the form of dynamic power consumption [22, 15], which is a function of CPU voltage and frequency and is approximated by:

$$P \propto V^2 f \tag{1}$$

We can reduce device power consumption by decreasing, i.e., scaling down, the voltage and frequency together. Decreasing frequency alone is not attractive since doing so increases program execution time proportionally, and thus, commonly offsets the energy savings. When scaling the voltage, we must also scale the frequency in the same proportion to meet signal propagation delay requirements [20]. By decreasing the voltage, we have the potential for quadratic power savings with only linear performance loss.

To minimize the effect of voltage scaling on system performance (while saving energy), dynamic voltage scaling (DVS) policies must estimate future workload and choose the most appropriate CPU level. Accurately predicting future workload is very challenging yet vital for maintaining acceptable performance. Misprediction can result in setting the CPU level too high, curtailing power savings, or in setting the CPU level too low, producing an unresponsive system and a very irritated user.

The goal of our work is to develop a system for DVS that effectively balances the tradeoff between performance and power consumption iPAQ hand-helds and their applications. The result, AutoDVS, is general-purpose (unlike other DVS-enabled operating systems such as GraceOS [27] for multimedia devices), and is fully automatic, i.e., it requires no application support or programmer effort (as is required in DVS systems such as Chameleon [12]). To implement AutoDVS, we incorporated and extended existing DVS techniques for interval scheduling and interactive task scheduling.

Interval Scheduling

Interval scheduling techniques [24, 8, 7, 23], divide the workload into fixed time intervals. These techniques use measurement history to estimate the workload in a future interval. For example, the PAST interval scheduler [24] assumes that the workload in the next interval will be same as the workload in last interval; the AVG_N interval scheduler [24, 8] assumes that the next interval is an exponential moving average (using a decay factor) of the N previous intervals. Other interval schedulers use observation heuristics [7] and more sophisticated statistical estimation methods [23] to make the predictions.

Interval schedulers have been shown to be quite limited for DVS for workloads that are commonly run on modern hand-held devices [8]. The primary reasons for this are that extant workload prediction approaches are inaccurate and that it is very difficult to identify a general mapping between different CPU loads and scaling levels that works for all applications. Prediction error causes minimal power savings or poor performance, e.g., missed (soft) deadlines, significant interruption, etc.

Another limitation of interval scheduling in a practical setting is the identification of an ideal, application-independent, interval length. To enable DVS techniques to detect and respond to changes in workload effectively, the interval length should be very short, e.g., 10-50ms. Unfortunately, the minimum interval length in a practical (non-simulated) setting is impacted by external factors, e.g., operating system timer resolution and scheduling quanta, which force a much longer interval length to be used (100-200ms).

Interactive Task Scheduling

Recent DVS studies have focused on classifying tasks into different groups, each with a customized policy. [5] suggests three groups: Interactive, periodic, and background tasks. In this system, each interactive task has an implicit 50 milliseconds deadline which is known as the user perception threshold. The system predicts task execution speed using the cumulative distribution function (CDF) of task CPU demand. The system computes task execution speed for future events using exponential smoothing of previous values. If the event exceeds its pre-determined worst case deadline (i.e. the panic threshold), the system scales the CPU to full speed.

A periodic task in this prior work consists of producer-consumer task pairs. Scheduling a periodic task requires that the system estimate the time period between the completion of a producer and start of a consumer. The system computes CPU speed such that producer ends immediately prior to when the consumer starts. The same speed is used for executing the consumer task. The authors of this work evaluate their approach using trace-driven simulation for workstation applications such as Acrobat Reader, Netscape, and Xemacs.

Lorch et al. suggests an approach that specifically targets user interactivity [14]. The system labels a user event with the type of GUI event that initiates it, such as a key-press, mouse-click, or drag event. Each event type has a separate DVS policy. The authors of this approach, compute the CPU schedule using PACE [13], a heuristic that is proven to compute optimal speed when the CPU can change frequency on a continuous scale and when task deadline and CDF of task CPU demand is known.

These approaches and similar approaches must overcome a significant challenge: determining the completion time of a task without any assistance from the application. The approach in [**?**] requires task completion time to update task execution time; the ap-

proach in [**?**] uses task completion time to compute task CDF and deadline. The solution described in the former work is precise, however, inherently complex; it requires monitoring system calls and communication between threads. The latter study suggests a simpler approach: an event is complete if a new event is posted or the idle thread is running and no I/O is ongoing.

3. AUTODVS

Our goal with this work was to produce a practical, implementable, and efficient DVS scheduling system for iPAQ hand-helds and their applications based on the findings of the previous work [5, 7, 8, 14, 23, 24]. With AutoDVS, we hope to exploit the benefits enabled by these algorithms while limiting any negative impact they might impose, by using them in combination. We implemented AutoDVS as a freely-available, lightweight, flexible extension to Familiar Linux. AutoDVS efficiently monitors executing programs and uses the resulting performance samples to estimate future behavior and to guide CPU scaling decisions.

The primary functionality of AutoDVS lies within two *sensors* and an *arbitrator*. The sensors detect and predict task-level (sub-application) and workload behavior; one sensor is responsible for application-specific user-interactivity and the other is responsible for the overall workload of the system. In addition, each sensor determines when CPU scaling is required. We describe the sensors in greater detail in the subsections that follow.

Each sensor makes scaling requests asynchronously to the arbitrator. The arbitrator is a high-priority kernel thread that implements requests. The sensors interact with the arbitrator via a well-defined API that supports requests for a particular CPU frequency, a one-level increase or decrease, and a value from 1-8 indicating the CPU level (1 being slowest). The arbitrator considers the current CPU level and if necessary, converts the request to an available clock frequency (as our CPU supports 8 levels, each number is mapped to a real level). The arbitrator mediates concurrent CPU scaling requests issued by the sensors; it assigns a higher priority to the interactive sensor and when two requests have the same priority, it schedules the one that will result in the higher speed only.

This division of labor across the system is key to the efficacy of AutoDVS. The interactivity sensor considers GUI events for each application separately and makes predictions of the duration of each interactive session. By considering each application individually, our predictions are very accurate due to regular, repeating patterns within applications. The CPU load sensor takes a global view of the system to identify additional DVS opportunities not made apparent to the fine-grain interactive scheduler. Together the sensors are able to consider a wide range of application behaviors to make accurate and effective CPU scaling decisions that reduce power consumption without negatively impacting performance.

In the subsections that follow, we first describe the iPAQ applications that we use to demonstrate various workload behaviors and performance requirements common in popular iPAQ software. We then detail the implementation of the AutoDVS sensors for detecting and predicting user-interactivity and changes in CPU load.

3.1 Example Applications

We use AutoDVS to reduce iPAQ power consumption for common workloads of popular, general-purpose hand-held applications with significant growth potential, e.g., personal assistance programs, games, and multimedia programs [16]. Such software poses many challenges to DVS systems because they require significant computation, heavy user interaction, and soft real-time constraints. Performance loss due to DVS for such applications can severely degrade the user's perception of program and device performance.

We describe all of the iPAQ software that we used in the empirical evaluation of AutoDVS in Section 4.2. However, we use a subset of these benchmarks as examples throughout the paper. Three of these programs are games with different performance characteristics: *Solitaire*, *Checkers* and *Tetrix*. Among these, Checkers has the highest computational requirements; Tetrix has less, but as the level of play increases, users tend to play very fast, generating many graphical user interface (GUI) events. Solitaire commonly consumes the fewest resources.

We use *Opieplayer* to demonstrate the user-interactivity, CPU resource consumption, and the soft real-time requirements common in multimedia applications. Opieplayer is a front-end to a multimedia library which includes decoders for many popular music and video formats. All of our applications are available from [18].

3.2 Interactivity Sensor

The first AutoDVS sensor targets per-application user-interactivity. It incorporates and extends the DVS strategies described by Flautner et al. [5] for interactive tasks. AutoDVS monitors GUI events to identify *interactive sessions* in arbitrary programs. Our system employs no notion of tasks, but instead automatically infers task-like behavior, i.e., periods of time, in which the user is interacting with the device. We refer to non-interactive sessions as *think times*. In addition, we do not distinguish event types (as is done in [14]) i.e, we consider only interactive sessions regardless of which events occur within them. The interactive sensor manages CPU scaling during interactive sessions. In particular, this sensor monitors events, detects and predicts interactive sessions, and scales the CPU for each interactive session.

3.2.1 Monitoring GUI Events

The interactive sensor collects event statistics about each application. The GUI events are triggered either by user input or by the applications themselves. In our hand-held platform, the user input events are generated by touch-buttons, the joypad, and the touch-screen. The GUI events provide a communication path between the applications and the window manager. The window update, focus, selection events are typical examples to GUI events.

In our platform, *all* GUI events are routed to the Window Manager (which implemented as a library) that then re-routes these events to the appropriate target applications. Each GUI application instance has its own event handler with which the sensor maintains separate event sessions for each application. We implemented the interactive sensor in the Window Manager library by extending an extant, call-back function without an implementation (a null function) called *event_filter*.

The event_filter in AutoDVS is an interface to the prediction library used to forecast interactive session lengths (we detail this in the next subsection) and as a policy maker for frequency scaling. The function timestamps event arrivals, determines the start of a think period, extracts session length predictions from the prediction library, and requests CPU scaling from the arbitrator. We implemented a new system call in the Linux kernel to serve as the interface to event_filter.

3.2.2 Detecting and Predicting Interactive Session Lengths

An interactive session starts with the arrival of an event and ends if no event is received for a period of t_p. Identifying the interactive sessions correctly is important, since presumably, the user is most sensitive to any performance loss during these periods. The value of t_p impacts the system in two ways. If t_p is too low, the algorithm might end an interactive session prematurely while the

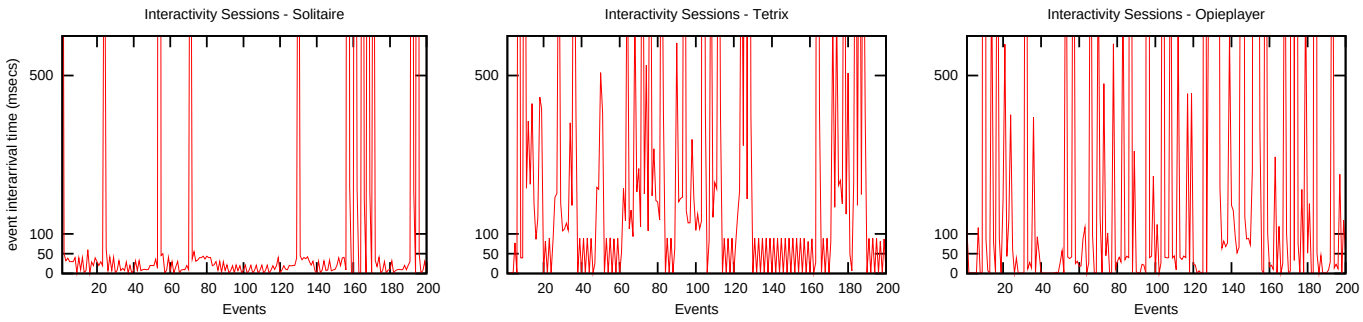

Figure 1: GUI event inter-arrival times for three typical applications -Solitaire, Tetrix and Opieplayer (a multimedia player). Solitaire has bursts of events separated with long idle times. Tetrix has much shorter bursts with smaller idle times due to higher interactivity. Opieplayer has largest event inter-arrival times with interactive sessions of almost a few events at a time.

application is still processing a GUI event. If t_p is too high, the sensor will maintain a high CPU speed and miss opportunities for reducing energy consumption. We determined t_p empirically to be 500 milliseconds (ms) using a large set of GUI programs. We found that out of 177561 GUI events, 99.3% of them were processed in less than 500ms.

Given the start of an interactive session, the sensor predicts its duration. The length of each session is very application-specific. To confirm this, Figure 1 shows the event arrival times for three typical cases using a cross section of an event trace from Solitaire, Tetrix, and Opieplayer. Solitaire receives long bursts of user input events bounded by large think times. The event burst is generated by the touch screen, during the frequent drag-and-drop operations involved in this game. In contrast, Tetrix receives the most user events through the keypad; this results in very short bursts of events. However, due to the nature of game, the event bursts are separated by smaller think times. For these examples, the median think time for Solitaire and Tetrix is 2.2 and 1.0 seconds, respectively. Opieplayer is much less interactive than both of those games with almost no burst of user interface events.

To predict the duration of an interactive session, we integrated NWSLite [9] into AutoDVS. NWSLite is an open-source prediction utility that we developed in prior work; it is an extension of the Network Weather Service [25] for Computational Grid Computing [6]. NWSLite is a non-parametric forecasting tool that targets embedded devices. NWSLite uses a mixture-of-experts approach for prediction rather than relying on a single model, i.e. it implements a set of time-series models, each having its own parameterization. Given a history of observed performance values, NWSLite generates a forecast for each measurement using each prediction model. NWSLite ranks each predictor by computing the cumulative prediction errors. Each time a forecast is requested, NWSLite chooses the predictor with the highest rank (lowest cumulative error). We have shown that NWSLite produces very accurate predictions for a wide range of resource types, including TCP latency, wired and wireless network bandwidth, CPU load, and CPU demand. Moreover, it does so with very low overhead: NWSLite uses 55 floating operations and 592 integer and miscellaneous operations per forecast. As such, we employ NWSLite to predict interactive session length for the interactive sensor in AutoDVS.

NWSLite implements five predictors currently; last value, exponential smoothing with %5 and %20 gain factors, running mean, and median with a window size of 5. The first two of these are particularly popular in research community and used extensively [17, 21, 24, 8]. Table 1 shows the first nine predictions of session length for Opieplayer. The first column is observed session length, the second column is the session length predicted by NWSLite, and the final column is the predictor chosen by NWSLite to make the prediction. The session lengths are highly variable, causing NWSLite to switch back and forth between various predictors in the beginning. Eventually, NWSLite stabilizes on exponential smoothing, with a gain factor of 5%, and uses it through the rest of dataset. NWSLite has a clear advantage over commonly used parametric predictors, especially when the dataset is non-stationary. Moreover, when the dataset is highly predictible, NWSLite will perform similarly to any predictor it incorporates. NWSLite can be easily extended to incorporate any time-series prediction technique.

	Real Value	Predicted Value	Winner Forecaster
1	94.44	243.88	Last Value
2	80.46	236.41	5% Exp Smooth
3	280.95	187.28	20% Exp Smooth
4	685.83	206.02	20% Exp Smooth
5	687.64	685.83	Last Value
6	121.66	687.64	Last Value
7	325.75	267.94	5% Exp Smooth
8	98.05	270.83	5% Exp Smooth
9	773.57	262.19	5% Exp Smooth

Table 1: The first nine session length predictions for Opieplayer using NWSLite. NWSLite always starts with the Last Value predictor. Due to the high variance in this particular dataset, NWSLite switches between multiple predictors before reaching a steady state.

3.2.3 *CPU Scaling for Interactive Sessions*

The interactive sensor is also responsible for CPU scaling for interactive sessions. We empirically evaluated a number of different policies within AutoDVS using a real iPAQ and popular iPAQ GUI software. We found that by far, the best policy is the most simple one: switching to the maximum clock speed during interactive sessions. The reason for this was the LCD driver in our evaluation platform: Each frequency switch results in a short but noticeable disruption in the display. Thus, even a small number of switches degrades the user's experience. As such, techniques that change CPU speeds on this platform during interactive sessions, e.g., PACE [13, 26] are not feasible.

The interactive sensor does not consider interactive sessions less than 500 μs to account for voltage settling time. The authors in [8] measured the voltage settling time to be less than 250 μs on the Itsy [1] platform which uses the same microprocessor as our iPAQs. We use 500 μs as a conservative estimate.

3.3 CPU Load Sensors

AutoDVS must also account for periods of time during workload execution that are not interactive. Most programs, even those that are primarily interactive, execute think (non-interactive or computationally intensive) periods. The CPU load sensor is responsible for these sessions. This sensor takes a global view of the system and workload, i.e., it does not consider task-level and application-specific details. The sensor employs two interval-schedulers to perform clock scheduling of non-interactive periods.

The authors in [8] concluded that interval schedulers do not perform well for real implementations on a hardware/software platform that is similar to our iPAQ platform. The reasons for this include the difficulty with which an appropriate, general-purpose, yet short, interval length can be identified, and the inaccuracy of extant approaches for prediction of future CPU load given past history. Past approaches to interval scheduling use very short, fixed-size intervals to ensure that the systems respond quickly to changes in the workload. Given short time intervals and the highly variable CPU load that is typical of hand-held applications, statistical and last-value prediction results in large forecasting errors and thus, mis-scaling of the CPU.

We have been able to employ interval schedulers effectively in AutoDVS. The key reasons for this are that (1) we couple their use for non-interactive sessions with completely different CPU scaling techniques for interactive sessions (described previously), (2) we use very long time intervals, and (3) we employ two different interval schedulers at once.

The two schedulers are called the *CPU Load Monitor* and the *Idle Process Monitor*. We implemented both monitors as small extensions to the Linux kernel timer interrupt handler. We execute each monitor when their particular interval periods expire.

The CPU load monitor considers very large intervals (10 second) and averages the measured CPU load across intervals. By averaging, we are able to eliminate noise in the data and distribute *slack time* more efficiently. Slack time consists of the idle cycles during an interval when CPU utilization is less than 1. The monitor predicts that the CPU load for the next interval will be the same as it is for the current interval (this is the PAST policy used in [24, 8]). The limitation of the CPU load monitor is that with a very long interval, our system response time is long – we only consider scaling the CPU every 10 seconds. The interactive sensor addresses this problem for interactive sessions since it responds immediately to such behavior. However, we need to respond quickly when the system becomes idle also, in order to conserve as much power as possible. For this we use the idle process monitor.

The idle process is process that the OS scheduler runs whenever no other process in the system is runnable. When the OS schedules the idle process, it puts the CPU into a low-power, hardware mode if one is available. The StrongArm CPU shuts down most clocks when entering the idle process, effectively stalling itself. The interrupt handler remains live during this time and wakes the CPU when an IRQ request occurs. The idle process monitor evaluates (then resets) idle process statistics every 500 milliseconds. The monitor considers the number of times the idle process was scheduled by the OS and its execution duration during the previous interval. We modified the Linux scheduler (sched.c) to collect and export this information as a kernel symbol.

Both monitors make CPU scaling requests to the arbitrator. The CPU load monitor uses an extension to Pering's hysteresis pair [19] to decide when to request a speed change. These values, (50,70) in prior work, indicate the available CPU load levels that are used to scale the CPU. When the load is below 50%, Pering stepped-down the CPU level; when the load was above 70%, he stepped-up the CPU level. We found empirically that the pair (60,80) works best in our real implementation for iPAQ software. The idle process monitor requests a step increase from the arbitrator when it detects a period (500ms) in which the idle process is never scheduled. If the monitor detects that the idle process executes for over half the interval time, it requests a step decrease.

Both monitors request only single step (CPU level) changes. The authors in [5] computed excess cycles during a period of time which they then used to compute new speeds to which to scale the CPU. We investigated such an approach in AutoDVS but found that doing so caused a large number of arbitrator requests and thrashing between CPU levels. Moreover, when we set the CPU to the minimum level when the idle process monitor detects an interval during which only the idle process executed, the system also thrashed. The thrashing is caused by the changing workload and the interaction between the sensors. Single-step changes avoid this problem. The disadvantage to single-stepping is a delayed response time if one of the level extremes (minimum or maximum) is most appropriate. We discuss this effect in our results analysis for soft real-time programs in Section *4.4.2*.

4. EVALUATION

We empirically evaluated the efficacy of our approach by running a large number of very different workloads on an iPAQ device with AutoDVS and comparative techniques. We collected a number of different performance metrics while doing so. In the subsections that follow, we describe our experimental setup and the benchmark workloads. We then define the metrics that we use in our empirical evaluation and present our results.

4.1 Experimental Platform

Our device infrastructure included five Compaq H3800 hand-held computers running Familiar Linux version 0.7.2 [4]. The H3800 is a very typical hand-held computer, with a 206MHz StrongArm CPU, 64 Mbytes of main memory and 32 Mbytes of Flash RAM. It is capable of dynamic frequency scaling, however, it *does not yet support dynamic voltage scaling*.

To estimate power savings, we use the technique defined in prior work [5] for a similar study. We assume that the StrongArm and XScale [11] processor exhibit similar power characteristics and use published data for the XScale XSA (the system most similar to the StrongArm in terms of maximum voltage and supported frequency range), in the estimation. We approximate the voltage levels of the XScale CPU using the available frequency levels and a second degree polynomial parameterized by the XSA data:

$$v = -4 \times 10^{-7} f^2 + 0.0015 f + 0.5324 \quad (2)$$

To compute the corresponding StrongArm voltage levels, we use Equation 3 as a mapping function. That is, we linearly scale StrongArm frequency range to the XScale frequency range:

$$f' = \frac{773 - 150.0}{206.4 - 59.0} \times (f - 59.0) + 150.0 \quad (3)$$

We present the estimated voltage levels in Table 2.

The StrongArm architecture requires that all of the primary peripherals be synchronous to the CPU clock [10]. This implies that all CPU scaling will impact the performance of memory, the I/O controller, DMA, the LCD controller, etc. The dependency between the CPU clock and external devices can cause significant differences between theoretical expectations (and simulated results) and practical results. For example, Grunwald et.al. found that the CPU utilization changes non-linearly with respect to clock frequency, possibly due to variations in memory access cycles [8].

Level	Freq. (MHz)	Estimated Voltage (mV)
1	59.0	748
2	73.7	832
3	88.5	914
4	103.2	992
5	118.0	1067
6	132.7	1139
7	147.5	1209
8	162.2	1274
9	176.9	1337
10	191.7	1397
11	206.4	1453

Table 2: SA1100 Clock frequency levels. Since our iPAQs implement only frequency scaling functionality, we estimate the energy savings from voltage scaling using the available iPAQ frequency levels. We estimate the voltage levels using the published XScale XSA parameters and the formula defined by others in prior work for the same computation.

Another obstacle that we encountered was the dependency on the CPU by other peripherals. In particular, the LCD driver limited our evaluation in two ways. First, the display started vibrating making it unreadable for any speed lower than 103MHz. Thus we had to eliminate three lowest frequencies. Second, frequency changes were accompanied by short but noticeable disruptions of display. As a workaround, we were forced to use a flat (unchanging) clock speed during interactive sessions.

The window manager we run on the devices is Opie [18] version 1.0.2. Opie is an open-source graphical user interface designed for Sharp Zaurus and Compaq hand-held computers. It is a full-fledged GUI comparable to commercial versions in both appearance and features. The available Opie applications include Calendar, Contacts, Drawpad, a multimedia player, a wide range of games, etc.

4.2 Benchmarks and Experimental Methodology

We evaluated AutoDVS using three different scenarios. (1) Interactive: Running GUI applications; (2) Soft Real-Time: Running a single multimedia application, such as a video or audio playback; and (3) Concurrent: Interactive and soft-real time applications running together.

To evaluate and compare the performance of interactive applications, we collected a set of usage traces and extracted event and timestamp information. However, we **did not simulate our algorithms on those traces**. Instead, we implemented AutoDVS, and **we replayed the events in real-time**. Moreover, our results include the time for frequency settling and for AutoDVS itself.

To collect the usage traces, we installed Opie on several Compaq H3800 hand-held computers and distributed them to graduate students in our department. We let the students know that we were capturing all the events happening and asked them to use the handhelds as their own as normally as possible and to reboot periodically (to end the session).

Before distribution, we made three modifications to the iPAQ systems: We (1) disabled network connectivity; (2) modified random number generators to use a fixed seed; and (3) mrogrammed each to clear all user state information after every reboot. These changes were necessary to eliminate as much non-determinism as possible so that we could re-generate the user events in the correct order during experimentation. We clear state information by removing saved files and resetting user preferences to default settings; this enables us to start from a known state during replay.

Trace	Event Count (ETime@206MHz)	Description
DrawPad-1	23100 (915.4s)	Drawing random pictures
General-1	3688 (448.1s)	General use including calendar, contacts and games
Solitaire-1	8700 (756.4s)	Multiple Solitaire games
Tetrix-1	6936 (583.8s)	Tetrix
Tetrix-2	1342 (210.1s)	Tetrix - very short and slow
Checkers-1	1238 (205.1s)	Checkers - medium difficulty
Checkers-2	1214 (265.7s)	Checkers - maximum difficulty
Checkers-3	2490 (1076.4s)	Checkers - maximum difficulty

Table 3: Event traces that we used for our experimentation. We gathered the traces using instrumented versions of the system while different users exercised the iPAQs. We named each trace to reflect the application that was dominant during the usage period.

Datasets	Encoding Rate(Kb/s)	File Size (KB)	Play Length (s)
Low.mp3	56	2893	424
Medium.mp3	128	4809	307
High.mp3	214	3304	218
Super.mp3	488	5824	240

Table 4: The Madplay input files (MP3 files) that we used to investigate the efficacy of AutoDVS under soft real-time constraints. The names of the files reflect the sampling rate of the encoders.

To capture the events, we instrumented the Linux kernel at the I/O driver level. We captured all events generated by the touchscreen, the keypad, and the joy-pad using a microsecond timestamp. We saved the identification information for captured events in RAM and copied them to permanent storage immediately prior to shut-down, to prevent any excessive overhead. The time and space overhead for event trace collection is small. Each event requires a total of 20 bytes: 8 bytes for the timestamp and 12 bytes for event type and attributes. Since we capture event in a device driver, we can read the current time directly from Linux kernel data structures and no system calls are required.

To replay the captured events, we developed a Linux kernel module. The module keeps the list of events in memory and sends them using a microsecond resolution timer.

The event traces describe user behavior from boot-up to shutdown. Some of the traces that we captured were not useful; they were either too short, or broken, i.e., dependent on user created files. Also some traces were too similar. Overall, we selected the traces described in Table 3. The second column shows the number of events in the trace and the total time (seconds) for the realtime play-back at maximum performance (206MHz) in parentheses. We named each event trace using the application that was active most of the time. The first four traces describe more general use applications and includes multiple program types. The last four traces are exclusively games.

To evaluate soft real-time applications we chose Madplay, which is an open-source, high quality MP3 decoder [2]. Madplay can decode the applications off-line or interface to the GNOME Enlightened Sound Daemon (ESD), for on-line playback. We use the latter. The input files we use in our experiments are described in Table 4.

4.3 Evaluation Metrics

We evaluated the impact of AutoDVS on both performance and energy using three different evaluation metrics: *percent degredation in performance*, *deadline miss count* and *energy savings ratio*. We compute percent degredation in performance in terms of us-

ing the highest (maximum speed) CPU level (MAX). Specifically, we divide the absolute difference between the execution time using MAX and AutoDVS by the execution time for MAX.

We evaluate the performance of Madplay by counting the number of deadline misses. Conceptually, audio playback happens in three stages: Madplay reads MP3 encoded data from a file, decodes and converts to the data to the PCM format, and forwards the translated data to the ESD sound server. The sound server manages both the control and data path to audio hardware. At the control path, ESD sets the audio parameters such as sampling rate (22050, 44100, etc), stereo/mono, and sample resolution (8/16 bits). At the data path, ESD multiplexes raw PCM data from input sources and transfers it to the audio hardware. For example, playing a stereo audio file at 44.1 kHz and 16 bits resolution requires:

$$44100 \times 2 \times 2 = 176400 \text{ bytes/second}$$

The sound server provides some level of buffering to prevent any interruption to data flow. However, if the application cannot produce data as fast as the consumption rate of the audio hardware, a buffer underrun condition occurs eventually. When this happens, the audio hardware fills the gap by repeating the most recent data or doing nothing (emitting silence). Every buffer underrun is perceivable by the user and degrades the user's experience with the device. To evaluate the performance of multimedia applications, we counted the buffer underrun events at the sound server level.

Finally, we compute the energy savings ratio using the method defined by Flautner et al. in [5] called *Energy Factor*. This value is the ratio of energy used by the scaled workload to energy used when workload is processed at full speed. That is:

$$EnergySavingsRatio = \frac{\sum_{i=1}^{n} v_i^2 f_i t_i}{v_{MAX}^2 f_{MAX} T} \quad (4)$$

v_i and f_i are the voltage and frequency of each period of time (t_i) between two frequency scaling operations. T refers to the execution time when CPU is run at full speed, i.e., using f_{MAX} and v_{MAX}. We use the frequency and voltage levels that are given in Table 2. To describe the savings, we use energy savings ratio which is equal to $1 - EnergyFactor$.

4.4 Results

We compare AutoDVS to two other policies: MAX, in which the CPU is set to the highest level (for maximum performance) and FIXED, in which we employ a fixed CPU speed that trades off performance and battery life – we use 132MHz (level 6) as is used in a similar study [8]. We experimented with three different scenarios. (1) Interactive: Running GUI applications; (2) Soft Real-Time: Running a single multimedia application, such as a video or audio playback; and (3) Concurrent: Interactive and soft-real time applications running together. We describe the results from each of these scenarios in the following subsections.

4.4.1 Interactive Workloads

We first compare the energy savings ratio enabled by the different policies for interactive applications. The FIXED policy has an advantage in this dataset; our empirical evaluations show that most interactive tasks require only a fraction of the maximum CPU power. A flat policy of 132MHz will provide adequate performance. The question we want to answer is this: Can AutoDVS achieve similar energy savings and still maintain a high level of responsiveness?

Figure 2 shows the energy savings ratio enabled by AutoDVS and FIXED (Equation 4, i.e., the energy consumed by AutoDVS and FIXED over that consumed by MAX). The height of each bar

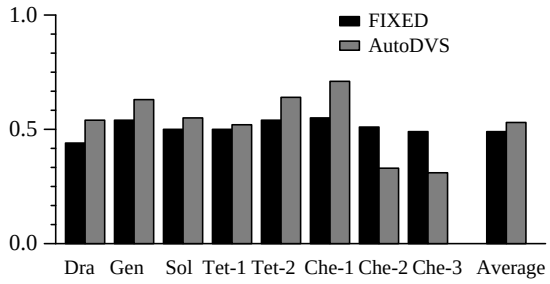

Figure 2: Energy savings ratio (relative to MAX (206MHz), the maximum performance scenario) for *interactive programs*. AutoDVS outperforms FIXED (132MHz) for most benchmarks and achieves an energy savings of 53% on average.

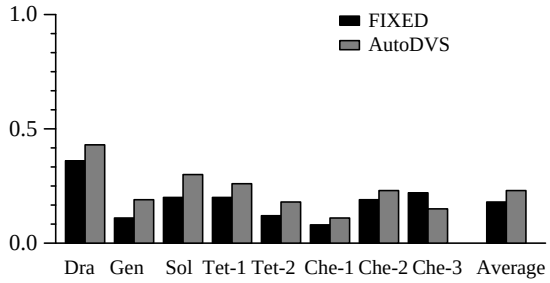

Figure 3: Percent degredation in performance with respect to MAX (maximum performance) for *interactive programs*. AutoDVS and FIXED degraded performance by 23% and 19%, respectively.

shows the percent of energy saved during each event trace, for example for the Drawing trace, AutoDVS saved almost 54% of dynamic CPU energy. We labeled the bars with the first three letter of the event trace, these are **Dra**wing, **Gen**eral, **Sol**itaire, **Tet**rix-**1**, **Tet**rix-**2**, **Che**ckers-**1**, **Che**ckers-**2** and **Che**ckers-**3**, from left to right.

The energy savings enabled by AutoDVS varied from 30% to 70% even beating the FIXED policy in some cases. In general, the lesser the performance requirements of benchmarks, the higher the savings. For example, for Tet-1 and Che-1, which both include game sessions at novice levels, the savings were highest. In contrast, the Che-2 and Che-3 were Checkers game sessions at expert levels. Both traces periodically triggered computationally expensive tasks that overlapped with user-think times. In these cases, there were few opportunities to save power. Even though these two tasks were the worst-case scenarios, the energy savings were approximately 30% for both.

Overall, the average energy savings due to AutoDVS and FIXED are very similar, 53%, and 51%, respectively. These savings come at a cost in performance, however. Figure 3 shows the percent degredation in performance (i.e. the extra time required for processing the workload over time), for the individual traces. Each bar shows the percent degredation with respect to MAX. Higher bars reflect a slower response time. On average, AutoDVS and FIXED degrade performance by 23% and 19%, respectively. Our empirical observations (actually playing the game while using AutoDVS and FIXED) indicated no noticible difference in game performance however since this degredation is distributed over the lifetime of the trace.

We analyzed the two worst cases further: Che-2 and Che-3 in which AutoDVS saves less energy than FIXED *and does not* enable

higher performance by doing so. For Che-2, the reason for this is directly tied to the dynamics of the clock scaling policy: The user, even though the game is set to expert level, played very fast with very short think times. Each interactive session switched the CPU to maximum speed, and right after the interactivity a computationally expensive session followed, leaving almost no opportunity for idle task monitor to kick-in and decrease clock speed.

Che-3 suffered for similar reasons, but to a lesser extent. Che-3 was a very long game session that lasted more than 20 minutes with a large number of interactive events. AutoDVS was particularly effective in determining and scaling clock frequency when interactivity started; our empirical observations showed that both Che-2 and Che-3 appeared as fast as maximum speed under AutoDVS.

4.4.2 *Multimedia Programs with Soft Real-Time Performance Requirements*

We next investigate the efficacy of AutoDVS on multimedia applications, in particular, programs that consist of tasks with soft deadlines (i.e. deadlines that can be missed but doing so results in a degraded user experience). The question we set out to answer is this: Can AutoDVS achieve the performance quality of the MAX policy and yet save a significant amount of energy?

Table 5 compares the soft real-time performance of the three DVS policies, MAX (maximum performance), FIXED (132MHz, unchanging), and AutoDVS. Column 1 is the input file name and columns 2 and 3 are the number of buffer underruns during playback. Column 4 shows the percent reduction in energy consumption that results from AutoDVS. On average, AutoDVS reduces energy consumption by 13% over MAX.

Buffer underruns (which we defined in Subsection 4.3) indicate the number of times that the music being emitted by the device is interrupted. Even a small number of interruptions are perceivable by the user and degrade the user's experience. We omit the buffer underrun data for MAX since it causes no buffer underruns for all datasets (however, it also conserves no energy).

Both policies perform well in terms of underruns for playback of the low quality sample, low.mp3. However, FIXED significantly degraded the sound quality (to the point where we could not determine what song was playing), when we used higher encoding rates - as reflected in the large number of buffer underruns. AutoDVS performed significantly better than FIXED since it was able to adjust the CPU speed to the higher load requirements dynamically.

We analyzed further the underruns produced by AutoDVS. Our investigation indicated that *most* of the underruns occurred at the start of the trace – when AutoDVS had no history with which to make its predictions. Figure 4 shows the distribution over time

Datasets	Buffer Underrun Count FIXED	Buffer Underrun Count AutoDVS	Energy Energy Savings Ratio for AutoDVS
Low.mp3	1	19	19.3%
Medium.mp3	11627	29	11.5%
High.mp3	8094	25	11.5%
Super.mp3	9192	36	10.4%
Average	7229	25	13.2%

Table 5: Buffer underrun counts for the multimedia program (with soft-realtime constraints) for FIXED and AutoDVS. There were no buffer underruns for MAX. The last column is the energy savings ratio for AutoDVS (with respect to MAX). All policies are satisfactory during the playback of the low quality sample. However, the FIXED policy fails for all others. AutoDVS can adapt itself easily for higher resource consumption.

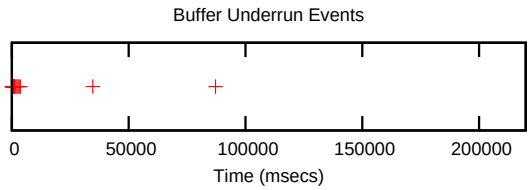

Figure 4: Buffer underrun events during the playback of high.mp3 using AutoDVS. All underruns, except two, happened during the initial three seconds. AutoDVS is unable to capture this interactive session because it has no past history to work from. AutoDVS recovers quickly and avoids all underruns in the steady state.

of buffer underruns during the playback of high.mp3. The same pattern occurs for all other input files. This indicates that there is some overhead imposed by AutoDVS *learning* the behavior of the workload. Learning time is required since our CPU load monitors single-step CPU levels to avoid thrashing (as described in Section 3). The data also indicates that AutoDVS is highly accurate once some workload history has been collected.

This investigation exposed another important interaction between idle task monitor and CPU load monitor. We compared the clock scheduling decisions over time and mapped them to the buffer underrun occurrences (we have omitted the figure due to space constraints). We found that two of the underruns occurred at the moment that load monitor decreased the clock speed, around the 40^{th} and 90^{th} seconds. The idle task monitor immediately increased the speed back to its previous level in the seconds that followed. Without the idle task monitor, there would have been no such recovery and many more buffer underruns would have occurred.

4.4.3 *Concurrent Workloads*

We next investigate how AutoDVS performs when multiple applications are running concurrently on the iPAQ. In particular, we replayed the event trace while running Madplay at the background. For each event trace, we started collecting the measurement statistics when the two tasks began executing events concurrently; we continued measuring until Madplay terminated. There are three short-traces and ended earlier than Madplay, Tet-2, Che-1 and Che-2. For those three cases above, Madplay was the single task for 45%, 42% and 33% of total evaluation time, respectively. As input to Madplay we used low.mp3 as it was the only input file that the FIXED policy was able to play without any noticeable quality degradation.

Running programs concurrently reduces the opportunities of CPU scaling since there are more processing requirements. The question that we are interested in is whether it is possible to extract any energy savings without hurting performance. Figure 5 compares the energy savings ration for FIXED and AutoDVS (using the same methodology as in the previous subsections). AutoDVS is able to saves a moderate amount of energy, 12% on average. The savings are minimal for tasks that have higher computation requirements, e.g., Che-3, Dra, and Sol.

The high energy savings for FIXED are deceiving in this figure. FIXED trades-off power for unacceptably low performance levels. Table 6 shows the number of buffer underruns. We omit the counts for MAX since in all but two cases (discussed below), there were no buffer underruns. Due to the lack of flexibility in clock speed setting, FIXED performed very poorly. For only in three of the benchmarks, Gen-1, Tet-2 and Che-1, FIXED was able to produce

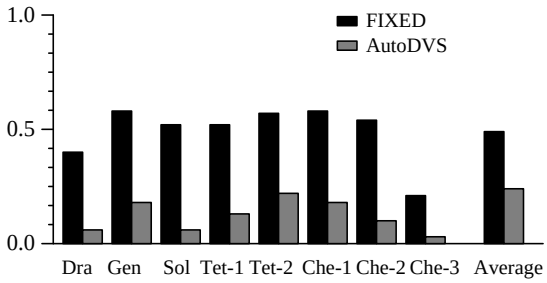

Figure 5: Energy savings ratio (with respect to MAX) when we execute multiple workloads (Madplay and interactive workloads) concurrently. Despite higher resource requirements, AutoDVS enables moderate energy savings (12% on average). FIXED appears to enable significantly more energy savings – however, FIXED severely degrades program performance.

Datasets	FIXED	AutoDVS	Description
DrawPad-1	2543	27	
General-1	8	20	Console input
Solitaire-1	1507	63	
Tetrix-1	1694	97	long game
Tetrix-2	202	24	short, slow game
Checkers-1	11	9	short, slow game
Checkers-2	630	481	very heavy computation, short game
Checkers-3	4936	3541	very heavy computation, long game

Table 6: Buffer underrun counts when we execute Madplay and interactive workloads concurrently . We used the low.mp3 input for playback as it is the only dataset for which FIXED performed well. MAX produced no buffer underruns for all programs except Che-2 and Che-3, for which it imposed 461 and 2073, respectively.

moderately acceptable result. This occurred because each of these workloads required very few computationally intensive sessions.

The buffer underrun numbers for AutoDVS is interesting. AutoDVS performed well in all but two cases, Che-2 and Che-3. In the tests for which AutoDVS performed well, most buffer underruns occurred during the initial a few seconds, due to the same phenomenon we explained in previous section. However, for Che-2 and Che-3, the overall quality was disappointing, with a high number of buffer underruns. This is due to high resource consumption of these two concurrent tasks. *Even MAX did not perform well* for these benchmarks. MAX encountered 461 and 2073 buffer underruns for Che-2 and Che-3, respectively, which resulted in significant degredation in system performance and user experience.

4.4.4 Integrating PACE

As a final experiment, we were interested in whether we could extend AutoDVS to conserve additional energy if it had access to event deadline information. To investigate this, we incorporated the PACE algorithm [13, 14], and in particular, a practical implementation of the algorithm called Practical Pace (PPACE) [26], into AutoDVS.

PACE is a technique that can compute optimal energy savings when *continuous* CPU scaling levels are available. PACE computes CPU speed as a function of work completed and gradually increases the CPU frequency as the task nears its deadline. PPACE handles descrete CPU scaling levels and uses a polynomial time approximation of PACE that is computationally efficient but does not always find the optimal solution.

Parameter	Value	Description
D	50 Msecs	Task deadline
WC	6.192 Mcycles	Worst-case execution cycles
r	6	Number of transition points
f	(103.2 - 206.4) MHz	StrongArm Clock frequency steps
s	[WC-1] / r	Transition period -evenly spaced
ϵ	0.05	Trim error parameter

Table 7: Simulation parameters that we used to investigate the efficacy of integrating PPACE into AutoDVS.

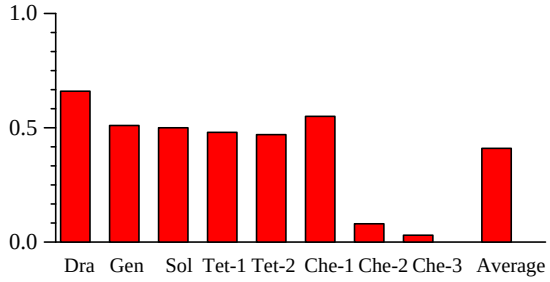

Figure 6: *Simulated* energy savings ratio with respect to AutoDVS when we integrate PPACE. These results are different from all those prior in that we obtained them through simulation. In addition, we consider only GUI events. On average, incorporating PPACE results in a potential 41% decrease in energy consumption of GUI events when the event deadlines and WCETs are known a priori.

We investigated the integration of AutoDVS and PPACE using trace-driven simulation using GUI events in our traces. That is, we only measure the impact of using PACE on GUI events (since they are the only actual tasks in our system that we can provide a priori deadlines for). We do not measure the energy savings on the entire system. Please note that for all prior experiments, we did not use simulation but instead, replayed the complete user activity traces in real time with AutoDVS and the comparative techniques.

We employed simulation to evaluate the *potential* of coupling PACE with AutoDVS for interactive events. An online implementation of PACE is currently not feasible for the following reasons. First, PACE requires frequency switches at a very fine granularity for optimal energy savings (i.e. every few milliseconds given 50ms deadlines). As we described previously, our system thrashes between energy levels and performs poorly when we scale the CPU too often. Second, the computational requirements of PPACE are still too great for an online system on an iPAQ. Third, the computation of cumulative distribution function requires offline information. Finally, AutoDVS requires an additional API through which it can collect the offline information, task deadlines, and task WCETs from the program.

Table 7 shows the parameters we use to evaluate PPACE in AutoDVS. To determine the worst-case execution time (WCET) in cycles, we use the CPU demand of 99 percentile of GUI tasks which is equal to 6.192 Mcycles. We limited the number of clock speed transitions to 6, placing them evenly in the range [1,WCET]. Even though we use a smaller number of transitions than were used in the Practical Pace study, our implementation provides a higher resolution than the original implementation since we use a much smaller WCET. Xu et al. uses $WC = 500$ Mcycles with 100 transition points – this corresponds to a transition point approximately every 5 Mcycles. In contrast, we place a transition point at approximately every 1 Mcycles.

Figure 6 shows the energy savings ratio when CPU speed is

rescheduled using PPACE – relative to AutoDVS (not MAX as in prior graphs). The data indicates that using PPACE with AutoDVS can potentially enable significant energy savings. Our results indicate that for most of our event traces, the energy consumption of GUI events can be decreased by over 50%. Che-2 and Che-3 are exceptions to general trend. For these two cases, the savings are less than 10%. On average, PPACE reduces the energy consumption of GUI events by 41%. As part of future work, we plan to attempt to address the limitations (articulated above) of incorporating PPACE into the real, online, implementation of AutoDVS, in an attempt to further reduce the energy consumption of iPAQ programs.

5. CONCLUSIONS AND FUTURE WORK

In an effort to produce an automatic dynamic voltage scaling system for a popular hand-held device, the HP iPAQ, we developed a set of Linux and Window Manager extensions that implement, couple, and extend a number of extant approaches to dynamic voltage scaling (DVS) to reduce power consumption. Our system is called AutoDVS and is very flexible and extensible. Each of the DVS algorithms used for different workload behaviors can be replaced with others. We intend for it to be used by researchers interested in investigating, empirically evaluating, and comparing DVS algorithms on iPAQ hand-helds using popular and general-purpose hand-held software.

Our results indicate that AutoDVS can reduce power consumption significantly for a wide range of application types executed alone or concurrently. On average, AutoDVS reduces power consumption 53% for interactive tasks, 13% for soft real-time tasks, and by 12% for concurrent workloads that both types of applications. AutoDVS enables these results automatically and transparently for a wide range of real applications. The key to enabling these power reductions is the use of interval schedulers that capture computationally intensive and idle periods in the workload and accurate time series prediction (by NWSLite [9]) to estimate the duration of application-specific interactive sessions. The combination of these techniques enables AutoDVS to infer accurately task-level behavior from applications, workloads, and concurrent workloads, and to adapt the clock speed appropriately.

As part of future work, we plan to study effective and practically efficient ways for implementing PPACE in AutoDVS by inferring and dynamically updating the CDF function and estimates for worst-case execution times for tasks. We plan to employ NWSLite to perform such estimations. In addition, we plan to investigate techniques that reduce the learning time of AutoDVS for soft real-time tasks and that are more aggressive for interactive tasks – in an effort to extract additional savings. Finally, we intend to implement AutoDVS into Familiar Linux for an iPAQ with actual multi-level (> 2 levels), voltage scaling when available (as opposed to only frequency scaling), to verify that our estimated energy savings translate into actual savings.

6. REFERENCES

[1] Itsy – http://research.compaq.com/wrl/projects/itsy/.

[2] Madplay – http://www.underbit.com/products/mad/.

[3] Compaq Computer Corporation. *iPAQ Pocket PC*. http://www.compaq.com/products/handhelds/pocketpc/.

[4] Familiar Web Page – `http://www.handhelds.org`.

[5] Krisztian Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. *Wirel. Netw.*, 8(5):507–520, 2002.

[6] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[7] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *ACM international conference on Mobile Computing and Networking (MoBiCom)*, pages 13–25, 1995.

[8] Dirk Grunwald, Philip Levis, Charles B. Morrey III, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *Fourth Symposium on Operating System Design and Implementation(OSDI 2000)*, pages 73–86, October 2000.

[9] Selim Gurun, Chandra Krintz, and Rich Wolski. Nwslite: a light-weight prediction utility for mobile devices. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 2–11. ACM Press, 2004.

[10] Intel. *StrongARM SA-1110 Microprocessor Developer's Manual*, October 2001. Order Number:278240-004.

[11] Intel Corporation. *Xscale*. www.intel.com/design/intelxscale/.

[12] Xiaotao Liu, Prashant Shenoy, and Mark Corner. Chameloen: Application controlled power management with performance isolation. Technical Report 04-26, Department of Computer Science University of Massachusetts, 2004.

[13] Jacob R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–61. ACM Press, 2001.

[14] Jacob R. Lorch and Alan Jay Smith. Using user interface event information in dynamic voltage scaling algorithms. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation Computer and Telecommunications Systems*, pages 46–55, October 2003.

[15] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 721–725. ACM Press, 2002.

[16] Brad Myers and Michael Beigl. Handheld computing. *IEEE Computer*, pages 27–29, september 2003.

[17] Dushyanth Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *International Conference on Mobile Systems, Applications, and Services*, 2003.

[18] OpenZaurus Web Page – `http://www.openzaurus.org/web`.

[19] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[20] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259. ACM Press, 2001.

[21] A. Rudenko, P. Reiher, G.Popek, and G.Kuenning. The remote processing framework for portable computer power saving. In *ACM Symp. Appl. Comp.*, San Antonio,TX, February 1999.

[22] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. Dynamic power consumption in virtex-ii FPGA family. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 157–164. ACM Press, 2002.

[23] Amit Sinha and Anantha P. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proceedings of the The 14th International Conference on VLSI Design (VLSID '01)*, page 221. IEEE Computer Society, 2001.

[24] Mark Weiser, Brent Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[25] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 1999.

[26] Ruibin Xu, Chenhai Xi, Rami Melhem, and Daniel Moss. Practical pace for embedded systems. In *Proceedings of the fourth ACM international conference on Embedded software*, pages 54–63. ACM Press, 2004.

[27] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.