HAPPE: Human and Application Driven Frequency Scaling for Processor Power Efficiency

Lei Yang, Robert P. Dick, Gokhan Memik, and Peter Dinda

Abstract—Conventional dynamic voltage and frequency scaling techniques use high CPU utilization as a predictor for user dissatisfaction, to which they react by increasing CPU frequency. In this paper, we demonstrate that for many interactive applications, perceived performance is highly-dependent upon the particular user and application, and is not linearly related to CPU utilization. This observation reveals an opportunity for reducing power consumption. We propose HAPPE (Human and Application driven frequency scaling for Processor Power Efficiency), an adaptive user-and-application-aware dynamic CPU frequency scaling technique. HAPPE continuously adapts processor frequency and voltage to the learned performance requirement of the current user and application. Adaptation to user requirements is quick and requires minimal effort from the user (typically a handful of key strokes). Once the system has adapted to the user's performance requirements, the user is not required to provide continued feedback but is permitted to provide additional feedback to adjust the control policy to changes in preferences. HAPPE was implemented on a Linux-based laptop and evaluated in 22 hours of controlled user studies. Compared to the default Linux CPU frequency controller, HAPPE reduces the measured system-wide power consumption of CPU-intensive interactive applications by 25% on average while maintaining user satisfaction.

Index Terms—Power, CPU frequency scaling, user-driven study, mobile systems

1 Introduction

Power efficiency has been a major technology driver for battery-powered mobile systems, such as mobile phones, personal digital assistants, MP3 players, and laptops. Power efficiency has also become a new focus for line-powered desktop systems and data centers because of its impact on power dissipation and chip temperature, which affect performance, reliability, and lifetime.

Processor power consumption is often a substantial portion of system power consumption in mobile systems [1]. Our measurements indicate that reducing the processor power consumption can save up to 40% of the overall system power consumption on a modern laptop (please refer to Section 5.6). Dynamic frequency and voltage scaling (DVFS) is one of the most commonly used power reduction techniques for processors. DVFS changes the frequency and voltage of a processor at run-time to trade off power consumption and performance. Most existing DVFS techniques, such as those used in the Linux [2] and Windows [3] operating systems, determine the appropriate processor frequency based on current CPU utilization. These approaches use CPU utilization as a measure of required performance. Therefore, to maintain adequate performance, they only allow a decrease in frequency when CPU utilization is below a certain threshold, e.g., 80%.

In the process of designing CPU power management techniques, it is possible to lose sight of an important fact: the ultimate goal of any computer system is to satisfy its users, not to execute a particular number of instructions per second. Although CPU utilization is a good indication of processor performance, the actual perceivable system performance depends on individual users and applications. Moreover, user satisfaction is not linearly related to CPU utilization. We conducted a study on 10 users with four interactive applications and found that for some applications, some users are satisfied with system performance when the processor is at the lowest frequency, while other users may not be satisfied even when it operates at the highest frequency. We also found that users may be insensitive to varying processor frequency for one application, but may be very sensitive to such changes for another application. Traditional DVFS policies that consider only CPU utilization or other user-oblivious performance metrics are often too pessimistic about user performance requirements, and use a high frequency in order to satisfy all users, resulting in wasted power. Similar findings were also reported in other studies [4], [5].

In this paper, we propose HAPPE (Human and Application driven frequency scaling for Processor Power Efficiency), a CPU DVFS technique that adapts voltage

- *L. Yang is with Google, Mountain View, CA. She completed much of this work while with Northwestern University, Evanston, IL. E-mail: leiyang@google.com*
- *R. P. Dick is with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI. E-mail: see http://robertdick.org/*
- *G. Memik and P. Dinda are with Electrical Engineering and Computer Science Department, Northwestern University, Evanston, IL. E-mail: memik@eecs.northwestern.edu and see http://pdinda.org/*

and frequency to the performance requirement of the current user and application. HAPPE associates individual users and applications at different CPU utilization levels with the lowest frequency that satisfies the user. HAPPE determines user satisfaction by taking direct input. However, it does not require continuous explicit user feedback, and only needs a short training period the first time a user runs an application. For each user and application, HAPPE saves the required CPU frequency at different utilization levels, and automatically loads this information upon later invocations of the application. After the training period, the user is not required to provide additional feedback, but may occasionally send new inputs to change the control policy, if desired.

We implemented HAPPE as a user-space CPU frequency governor for Linux and compared it to the Linux default *ondemand* frequency governor [2]. To find out whether HAPPE can save more power while still satisfying users, we conducted a study on 24 users with four representative CPU-intensive interactive applications for mobile systems such as smart phones and laptops. Compared to the ondemand governor, HAPPE reduced the overall system power consumption by an average of 25% without degrading user satisfaction. This reduction is significant, considering that the highest possible system-wide power reduction by constantly scaling frequency from the highest level to the lowest level is 40.63%, when the CPU utilization is above 80% (the average for our testing applications).

The rest of this paper is organized as follows. Section 2 introduces related work. Section 3 describes our study of user-perceived performance and its relationship with CPU utilization. Section 4 presents the user and application driven frequency control algorithm used in HAPPE. Section 5 presents our experimental setup for the user studies and power measurements, and shows the comparison results for HAPPE and the Linux ondemand frequency governor. Finally, Section 6 concludes the paper.

2 Related Work

Dynamic voltage and frequency scaling (DVFS) is a commonly-used power reduction technique for processors. Traditional DVFS policies use CPU utilization as the metric to determine when to change frequency [2], [3]. Sasaki et al. [6] used other hardware performance information available to the operating system to make frequency change decisions. Their DVFS algorithm is based on statistical analysis of performance counters. However, their technique needs compiler support to insert code for performance prediction.

Choi, Soma, and Pedram [7] changed CPU frequency based on workload decomposition, which tends to provide power improvements only for memory-bound applications. Wu et al. [8] designed a framework for a run-time DVFS optimizer in a general dynamic compilation system. Xu, Mossé, and Melhem [9] describe a DVFS scheme that captures the variability of workloads using the probability distribution of the computational requirement of each task in the system. To the best of our knowledge, none of these techniques directly used user satisfaction to inform the power management state controller.

Researchers have also described techniques that take user perception into account in DVFS and other related areas. There are in general three types of approaches.

Explicitly obtaining user input, e.g., by monitoring mouse movement or keyboard events. Lorch and Smith [10] found that different types of user interface events such as mouse movements, mouse clicks, and keystrokes trigger tasks with significantly different CPU requirements, suggesting that DVFS algorithms should adjust speeds based on interface events.

Lin et al. [11] describe a DVFS scheme that controls processor frequency by monitoring explicit user input via keyboard events. This technique uses a simple control policy that assigns a time interval for the processor to stay at each frequency level. When the user presses the discomfort key, the frequency is increased and the time interval for the previous frequency level is adjusted accordingly. This simple control policy is completely user-driven and does not make use of any CPU performance information. It does not learn user preferences for particular applications, and requires ongoing (although gradually slowing) explicit feedback from users to maintain adequate performance, which may eventually annoy users.

Implicitly estimating user perception by measuring metrics such as the response times in interactive applications or rate of change of video output as a proxy for the user. Although such approaches use models for user perceivable performance, they cannot distinguish among the requirements of different users, thereby neglecting potential power savings.

Flautner and Mudge [12] proposed Vertigo, which adjusts CPU frequency to different workload characteristics. It monitors application messages to measure user-perceived latency and proposes a layered frequency scaling scheme based on the user-perceived latency. Endo et al. [13] used latency as a performance metric and for detecting performance anomalies in operating systems. Gupta, Lin, and Dinda [4] demonstrated a high variation in user tolerance for restrictions in the availability of CPU, memory, and disk resources. Mallik et al. [14] demonstrated that this variation holds for power management as well.

Yan, Zhong, and Jha [15] defined the delay between user input and computer response as a measure of user-perceived latency, and used it to control CPU voltage scaling. Mallik et al. [16] describe PICSEL, a framework that uses measurements of variations in the rate of change of video output to estimate user-perceived performance, and adapts CPU frequency accordingly. However, these techniques ignore the variation among users and applications.

Shye et al. [5] proposed a DVFS scheme that uses a neural network model to predict user satisfaction based on hardware performance counter readings. This technique requires an off-line training stage for each user and application, which runs the application at different frequency level and asks the user for a verbal satisfaction rating. The user cannot do productive work during the training phase. If the operating conditions or user preferences change, the training must be repeated.

Implicitly inferring user satisfaction by monitoring the status of biometric sensors on the user. This approach requires specially designed biometric sensors to be attached to users. It remains to be seen how easily these sensors can be integrated into computer systems and normal workflows. As such technologies become practical, HAPPE could use them instead of explicit user input.

Shye et al. [17] proposed the addition of new biometric input devices for gathering information about user physiological traits. They considered three biometric devices: eye tracker, galvanic skin response (GSR) sensor, and force sensors. The studies show that there are significant changes in human physiological traits as performance decreases and these changes correlate strongly to the satisfaction levels reported by the users. Based upon these observations, they constructed a DVFS scheme called Physiological Traits-based Power-management (PTP).

3 User-Perceived Performance

CPU utilization has been widely used as a proxy for required performance. In traditional CPU DVFS policies, it is used as the metric to determine CPU frequency. To guarantee high processor performance, these policies only decrease frequency when CPU utilization is below a certain threshold, e.g., 80%. However, processor performance is not identical to user-perceived performance.

Do different users have the same performance requirement for the same application? Does one user have the same performance requirement for different applications? To find out, we conducted a 10-hour user study with 10 users[1] on a Lenovo Thinkpad T61 laptop, which has a Intel Core 2 Duo processor, 2 GB memory, and runs OpenSuse 10.3 and version 2.6.22 of the Linux kernel. Linux supports five frequencies for this processor: 0.8 GHz, 1.2 GHz, 1.6 GHz, 2.2 GHz, and 2.3 GHz[2], and scales voltage automatically with frequency.

We used four Linux CPU-intensive interactive games as our testing applications: Torcs, a 3D car racing game; Quake3, a 3D shooting game; Glest, a 3D real-time strategy game; and Trackballs, a 3D ball maze game. We selected these testing applications as representatives of typical CPU-intensive interactive applications on high-end mobile systems. Section 5 gives additional detail on our reasons for, and implications of, selecting these applications.

In the user study, each user played these games at all five frequency levels. The user studies were double-blind and the order of frequencies was randomized to eliminate any possible "first-time execution" impact. Each application was run for at least 2 minutes, and users were permitted to play as long as they desired to evaluate the responsiveness/performance of the system. After each play period, users were prompted to enter their "*satisfaction level with the system responsiveness/performance on a scale of 1 to 5, where 5 is the most satisfied and 1 is the least satisfied.*" While users were playing the game, a program ran in the background to sample utilization level for both CPUs on the laptop. Because our test platform has two CPUs, in all user studies the testing application being evaluated was scheduled to run on CPU0 using the Linux *taskset* utility.

Figure 1 illustrates (1) the satisfaction ratings of 10 users at five frequency levels, where level 1 represents the lowest frequency (0.8 GHz) and level 5 represents the highest frequency (2.3 GHz), and (2) the average utilization of CPU0 (the CPU that is actually running the task), obtained by sampling the CPU utilization every second during the user study for each user. As shown in Figure 1, all four applications are CPU-intensive. In general, CPU utilization decreases as frequency increases and user satisfaction appears to be a monotonic function of frequency. However, this function is non-linear, and differs greatly among users and applications. There are only a few cases in which user satisfaction decreases when frequency increases. In the 150 test cases (10 users, 3 applications, and 5 frequency levels) presented in Figure 1, there are 12 cases (8%) in which the user rating decreases by one level when frequency is increased by one level, and 4 cases (2.7%) in which the user rating decreases by two levels when frequency is increased by one level. We believe that the most likely explanation for these rare cases is reporting noise. Fortunately, they have limited practical impact on the technique due to their rarity.

These results provide evidence that conventional DVFS policies that only use CPU utilization as the control metric, ignoring variation among users and applications, are likely to either annoy users or waste power.

4 The HAPPE Approach

The objective of HAPPE is to minimize power consumption without degrading user-perceived performance. For user-interactive systems, the optimal frequency for the current application at the current CPU utilization level is the lowest frequency necessary to satisfy the current user. To approximate the optimal frequency, HAPPE learns user preferences and obtains user feedback by monitoring special performance and power keys, which can be mapped to any two keys or key combinations on a regular keyboard. Users may press the performance

1. The scale of this first user study was kept small because each evaluation took an hour. We conducted a larger scale second study with 24 users to evaluate HAPPE, which is presented in Section 5.

2. The Linux *acpi-cpufreq* driver identifies the highest frequency as 2,201 MHz. However, we found the actual frequency is 2,300 MHz using a timing analysis program that operates within cache.

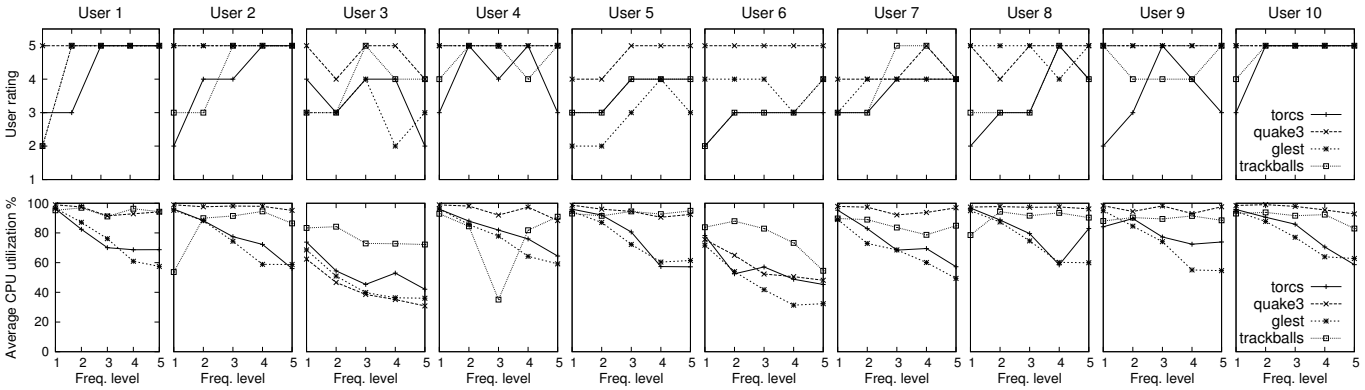


Fig. 1. Satisfaction rating and CPU utilization of 10 users at 5 different frequency levels. Level 1 is the lowest frequency and level 5 is the highest frequency.

Algorithm 1 HAPPE frequency controller (left) and w_map function (right)

for each sample period P_i **do**
  $f_{next} = f_{cur}$
  if performance key pressed **then**
    $f_{next} = f_{cur} + 1$
    $w\_map(user, app, f_{next}, \widetilde{util})$
  else if power key pressed **then**
    $f_{next} = f_{cur} - 1$
    $w\_map(user, app, f_{next}, \widetilde{util})$
  else
    $f_{next} = r\_map(user, app, \widetilde{util})$
  end if
  if $f_{next} \neq f_{cur}$ **then**
    $set\_freq(f_{next})$
  end if
end for

Input: $user$, app, f_{next}, $\widetilde{util}$
$map[user][app][\widetilde{util}] = f_{next}$
for all $u > \widetilde{util}$ **do**
  if $map[user][app][u] < f_{next}$ **then**
    $map[user][app][u] = f_{next}$
  end if
end for
for all $u < \widetilde{util}$ **do**
  if $map[user][app][u] > f_{next}$ **then**
    $map[user][app][u] = f_{next}$
  end if
end for

key when the responsiveness/performance of the system does not satisfy them. They may press the power key when they are satisfied with performance and want to save power.

4.1 User Application Frequency Profile

For each user, HAPPE creates a *user application frequency profile* for every interactive application the user executes. When a different user logs into the computer, HAPPE loads the appropriate profile, i.e., the appropriate power management settings for a user–application combination only start from a blank slate the first time that particular user runs the application. It is important to note that the frequency profile must distinguish between applications as well as users. Consider the example of user 6 in Figure 1. If the frequency profile for Quake3 (for which the lowest frequency satisfies the user) were to be used for Torcs or Trackballs, the user would be very dissatisfied.

We indicate the highest CPU frequency with f_{max}, the current frequency with f_{cur}, and the current CPU utilization with $util$. The *normalized CPU utilization* is defined as follows: $\widetilde{util} = util \cdot f/f_{max}$. The user application frequency profile divides the normalized CPU utilization into ten discrete levels (e.g., 0%–10%) and maps a user-satisfactory frequency to each level. The frequencies at all normalized utilization levels are initialized to the lowest frequency. Then, every sample period (P), e.g., one second, HAPPE refreshes the frequency for the next sample period (f_{next}), and updates the corresponding frequency profile if necessary.

Algorithm 1 describes the technique HAPPE uses to control processor frequency. HAPPE determines f_{next} by checking user feedback in the last sample period and looking up the corresponding frequency profile entry (the r_map function) based on normalized CPU utilization ($\widetilde{util}$), current user ($user$), and current application of focus (app). If neither the performance or power key was pressed in the last sample period, HAPPE uses the frequency profile to determine the frequency that previously satisfied the user at the current normalized utilization level, and adjusts the CPU voltage and frequency appropriately. If HAPPE detects that the performance key was pressed in the last sample period, it increases CPU frequency by one level. Otherwise, if HAPPE detects that the power key was pressed in the last sample period, it decreases CPU frequency by

one level. Then, HAPPE updates the corresponding frequency profile entry using the w_map function presented in Algorithm 1. Based on the data presented in Section 3, we assume that for the same user and application, lower CPU utilization levels require equal or lower CPU frequencies. Therefore, HAPPE not only updates the current normalized utilization level, but also checks to make sure that all utilization levels that are higher than the current level have at least the same frequency, and that all utilization levels that are lower than the current level have at most the same frequency.

The computational overhead of HAPPE is decided by the sample period P in Algorithm 1. This is because HAPPE re-evaluates user satisfaction based on the feedback input in the last sample period and adapts CPU frequency accordingly in every sample period. In our experiments, P is set to 1 second. In practice, using 1 second as sample period is sufficient to reduce power while maintaining user satisfaction, and results in negligible performance overhead. To estimate the performance overhead of HAPPE, we measured the CPU utilization of the experimental platform with the processor at the lowest frequency and without any applications running, establishing that the baseline CPU utilization was 0%. Then with all else being equal, we measured the CPU utilization when HAPPE was running, which was on average 1% and did not exceed 2%.

4.2 Training Phase

When a user first runs an application, HAPPE starts with a blank state and goes through a training period to build the user application frequency profile. Unlike previous work that requires continuous explicit user feedback [11], the training period in HAPPE is short, implicit, and not separated from normal application use: users may actively use the application during training and need not restart the application when training is finished. During the training period, the user needs to use only a few keystrokes to find the satisfactory frequency level. Then, for each user and application, HAPPE learns the required CPU frequencies at different utilization levels using the frequency profile, and automatically loads the profile upon later invocation of the application. After the training period, the user is not required to provide additional input, but is permitted to do so if desired. For example, the user's performance requirements may change each time the same application is invoked, or during different phases of the application. In cases like this, HAPPE starts with the user application frequency profile created previously, and quickly adapts to the new requirements based on user inputs.

4.3 Implementation and Discussions

We implemented HAPPE as a user-space CPU frequency governor for Linux. The governor program uses *Pthreads* to create two threads: T_1 and T_2. The first thread (T_1) polls CPU utilization every second, checks for user input signals from the second thread (T_2), and scales frequency and updates the user application frequency profile when necessary. T_2 monitors keyboard events, and sends a signal to T_1 when the performance key or power key is pressed. There are a few additional details that proved important when implementing HAPPE.

- Some users may send a burst of key presses when they are unsatisfied with performance, without waiting to observe performance improvement. This would result in moving to the highest frequency, which may not be necessary to satisfy the user. To prevent this, we treat all series of key presses within intervals smaller than one second as a single key press.
- For multiprocessor systems, HAPPE manages each CPU individually if its frequency can be changed independently. If the processors must share the same frequency due to hardware limitations, HAPPE manages the frequency of the group based on the highest utilization within it.
- If the user is concurrently running several multitasking applications, HAPPE monitors the current interactive application of focus, e.g., the current active X-window application, and updates its frequency profile. HAPPE maintains a CPU frequency that satisfies the user running the current interactive application. If the active application is the most CPU-intensive process, HAPPE works well as described in Algorithm 1. However, it is possible that the current application of focus is not CPU-intensive, while another background process (or processes) is consuming most of the CPU cycles. For example, the user may be browsing a web site while the system is busy installing updates in the background. In this case, HAPPE still follows Algorithm 1 and tries to decrease frequency, which would most likely not hurt the responsiveness of the application of focus and thereby not cause user dissatisfaction. However, since CPU frequency is decreased, the performance of the other CPU-intensive process will be degraded. A user that indeed cares about the performance of that process, would likely soon switch back to it and send improvement request to HAPPE by pressing the performance key. In this case, HAPPE would eventually adjust the CPU frequency to a desirable level. In the rare event where the user does not switch back to the other process, the key press events would be sent to the process of focus, which would also increase the frequency.
- For portable systems such as laptops and smart phones, it is unlikely that multiple users will run CPU-intensive applications on the system simultaneously. However, if this occurs, HAPPE follows the same policy and creates a frequency profile for each user and application combination. At any given time, HAPPE will use the highest frequency necessary to satisfy all current users.

5 Evaluation Results

To evaluate HAPPE, we conducted a 12-hour, 24-participant user study. Each participant used multiple interactive and CPU-intensive applications in a variety of power management environments. Constraints on the amount of time each study participant can realistically volunteer made it necessary to limit the total number of applications with which HAPPE was tested. We therefore selected applications for which it was most difficult to draw conclusions in the absence of empirical evidence. The analysis used during application selection follows.

Applications run on mobile computers can be broken into categories based on whether they are CPU-intensive and whether they are interactive. For applications that are interactive but not CPU-intensive (e.g., text and image editing applications such as OpenOffice and GIMP), we experimentally determined that conventional DVFS schemes such as the Linux ondemand governor detect low CPU utilization and change DVFS state to the same level as HAPPE. For applications that are non-interactive but CPU-intensive (e.g., compilers or file compression software), user feedback on performance is generally deferred until application termination, increasing the complexity of directly using feedback-based learning of user preferences. In summary, for applications that have low CPU utilization and/or are non-interactive, HAPPE does as well as conventional DVFS control techniques, but does not do better. However, for applications that are both CPU-intensive and interactive, it might be possible to learn user preferences and use this information to reduce power consumption below that of conventional CPU utilization based power management techniques. Our user study therefore focused on answering the following questions for CPU-intensive, interactive applications.

- Can HAPPE reduce power consumption more than the Linux ondemand governor by using the variation among users and applications?
- How close are power consumption improvements brought by HAPPE to the best possible via any DVFS policy?
- Can HAPPE provide similar level of user satisfaction to the Linux ondemand governor?
- How does providing users feedback on the power consumption implications of their decisions influence power consumption and user satisfaction?

5.1 Setup for Power Measurements

All our experiments were performed on the Lenovo Thinkpad T61 laptop described in Section 3. The voltages/frequencies of the two cores in this laptop are scaled together due to hardware constraints. We connected the T61's DC power supply in series with a 100 mΩ Ohmite Lo-Mite 15FR025 molded silicone wire element current sensing resistor. Then we measured the voltage across the resistor to obtain the current of the laptop, using a National Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Linux. This allows us to measure the power consumption of the entire system (including other power consuming components such as memory, graphic card, LCD display, and hard disk). Our power measurement setup is illustrated in Figure 2. During all experiments, the back-light of the LCD display is set to maximum brightness. Our technique focuses on reducing CPU power consumption, because it is a major component in system-wide power consumption. However, our objective is to improve the battery life for portable systems, which is determined by the system-wide power consumption, not the CPU power consumption alone. Therefore, we measured the power consumption of the entire portable system. Note that reducing processor frequency can also reduce the demands on other devices, and therefore their power consumptions.

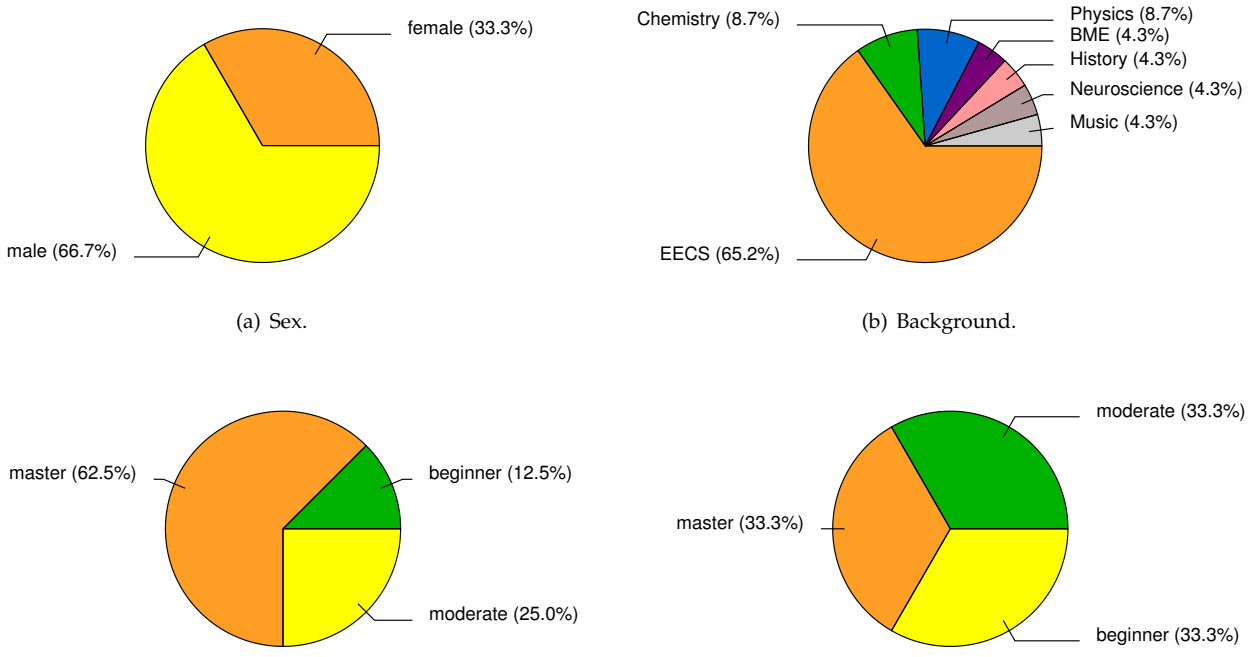
Fig. 2. Setup for power measurements.

5.2 Power Key Feedback

When the performance key is pressed, the user receives almost immediate positive feedback via change in performance. However, when the power key is pressed, the user does not normally receive feedback on the benefits for a long time: the positive effect is increased battery life. In real-world scenarios, users have an incentive to save power when their laptops or other portable devices are running on battery power, but the benefits come later (many hours before the impact on battery life is known) than practical to observe in a user study (2 minutes for each evaluation). Rahmati, Qian, and Zhong found that making battery life information visible to end users has a strong impact on power management for mobile devices [18].

In our user studies, we displayed a bar graph indicating battery life in the bottom-left corner of the screen to provide feedback on the effects of power consumption. We control the indicator with a simulated battery that is designed to last for two hours[3] when the processor

3. We used two hours to approximate the real-world scenario. Experiments with shorter battery life period such as a few minutes showed that users would panic when they saw the timer is reaching zero and would press the keys randomly.



(a) Sex.

(b) Background.

(c) Experience with computer.

(d) Experience with video games.

Fig. 3. Demographics of users.

is at the lowest frequency and 70 minutes when the processor is at the highest frequency. The indicator displays the remaining operating time based on current battery energy and power consumption, obtained from an on-line power model based on the measured power consumption in Table 2.

Note that HAPPE does not require the use of the battery life indicator. In our evaluation, we tested HAPPE both with and without using the battery life indicator to determine how providing feedback on power consumption to users changes their power management decisions.

5.3 Setup for User Study

The 24 user study participants are graduate and undergraduate university students; they span a wide range of professional backgrounds, races, ages, and computer and gaming experience levels. Figure 3 illustrates study participant demographics, which may be of use to the reader in judging whether the diversity in subject background and experience is sufficient for our experimental results to generalize to other populations of interest. Each user evaluation lasted about 40 minutes. Prior to each evaluation, users were asked to fill out questionnaires to rate their level of experience with computers and computer games using one of the following levels: beginner, moderate, and master. Then they read handouts with the instructions for the experiment. The users were also shown how the games are played, and were permitted to practice until they were able to play on their own.

In the user study, each user plays each of the four games four times, denoted as *ondemand*, *T-HAPPE*, *O-HAPPE*, and *B-HAPPE*. Each time, the game is played for two minutes and then exits automatically. Afterwards, the users are prompted to enter their "*satisfaction level with the computer performance/responsiveness on a scale of 1 to 5, where 5 is the most satisfied and 1 is the least satisfied*". Although some users may consistently be biased toward reporting higher or lower levels of satisfaction, this rating system allows us to detect changes in the satisfaction of a particular user when experiencing different levels of application performance during the study.

During the first run, users play the game normally. CPU frequency is controlled by the Linux ondemand frequency controller, with the *sample_rate* set to 80 ms and the *up_threshold* set to 80%. During next three runs, HAPPE is used to control CPU frequency. Users may press a green-colored key to require higher performance or better responsiveness, or press a yellow-colored key to save power when they are satisfied with the current performance; in our experiments, colored labels were attached to otherwise-unused keys near the space bar. Note that users are not required to press either key. The three phases differ as follows.

- *T-HAPPE* is the training phase. During this phase, the frequencies at all utilization levels are initialized to the lowest level and adjusted when the two control keys are pressed. This phase is separated deliberately from the following phase to determine whether more frequent feedback during this phase

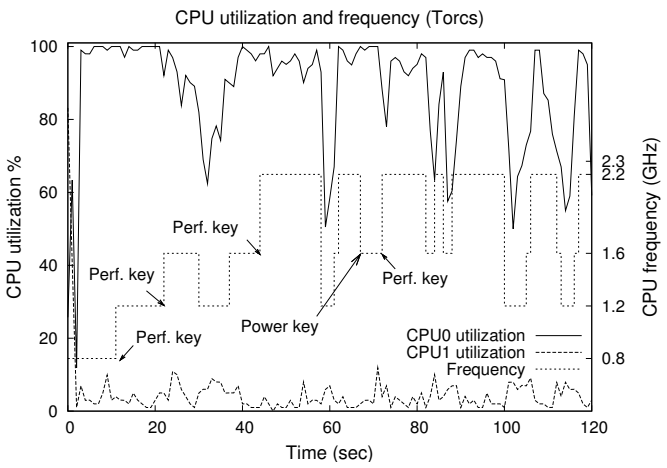


Fig. 4. Example of HAPPE training phase.

annoys users, although there would be no separation of these phases in practical (non-study) use.

- During *O-HAPPE*, HAPPE loads the user application frequency profile created during *T-HAPPE* and controls frequency accordingly. The user may still press the two keys to adjust performance. However, the frequency of user interaction is likely to be lower, and better approximates HAPPE after a brief initial training period.
- During *B-HAPPE*, the user is provided with a battery life indicator that provides feedback on energy use. It is otherwise equivalent to *O-HAPPE*.

5.4 Example of Dynamic Behavior

To illustrate user interaction with HAPPE, Figure 4 shows the time series data of a user playing the Torcs car racing game under the control of HAPPE during the training phase. Figure 4 shows the utilization of both CPUs, the frequency sampled every second, and key press events. When the user started playing the game, frequency was set to the lowest level: 0.8 GHz. After 16 seconds, the user pressed the performance key. HAPPE increased the frequency to 1.2 GHz, and recorded the frequency requirement of the user at this utilization level in the frequency profile. Then, the user pressed the performance key again at 22 seconds and 43 seconds, further increasing the frequency to 2.2 GHz. At 67 seconds, the user pressed the power key and presumably soon realized that the resulting degradation in performance was not tolerable. The user therefore pressed the performance key after 7 seconds. Afterwards, frequency stayed at 2.2 GHz for high CPU utilization levels, and 1.2 GHz for low CPU utilization levels.

5.5 Comparing HAPPE with Linux Ondemand

Figure 5 illustrates the aggregated power consumption and satisfaction ratings across all users and applications, comparing HAPPE with the Linux ondemand frequency controller. For each user, each application, and each technique (*ondemand*, *T-HAPPE*, *O-HAPPE*, and *B-HAPPE*), we obtain the average power consumption during the two minutes the user plays the game. For each application and each technique, the following aggregated results are presented in Table 1 (from left to right): average power consumption of all users (in Watts), standard deviation of average power consumption (in Watts), average satisfaction rating of all users, standard deviation of average satisfaction rating, and the improvement in power consumption compared to the Linux ondemand controller. We make the following observations based on the results presented in Table 1.

- Compared to the Linux ondemand governor, across all four applications, the average reduction in power consumption is 28.50% during the HAPPE training phase, 24.39% during the HAPPE operating phase without the battery indicator, and 26.56% during the HAPPE operating phase with the battery indicator. These results are expected, because (1) during the training phase, the frequency starts low and only increases gradually when the user requires higher performance, and (2) the battery indicator, which provides feedback on the long-term battery life benefits of pressing the power key, gives the user more incentive to save power.
- Across all four applications, the average user satisfaction is 4.61 with the ondemand governor, 4.55 during the HAPPE training phase, 4.69 during the HAPPE operating phase without the battery indicator, and 4.63 during the HAPPE operating phase with the battery indicator. These results indicate that users are in general slightly less satisfied during the training phase, and more satisfied during the operating phase. In addition, the battery indicator can motivate users to save power by pressing the power key, but this appears to very slightly reduce their satisfaction. Nonetheless, compared to the default ondemand governor, during the HAPPE operating phase, users satisfaction actually increases slightly. This improvement could be noise, or might be due to the fact that users are happier when they feel they have control over the computer, i.e., when they press the performance button, they see an instant improvement in the performance/responsiveness[4].

In summary, compared to the Linux ondemand frequency controller, HAPPE reduces system-wide power consumption by 25% on average, without degrading user satisfaction.

5.6 Comparing HAPPE with the Best Possible DVFS

HAPPE produces a substantial full-system power reduction relative to the default ondemand controller. In

4. We considered the possibility of hardware thermal emergency throttling during use of the ondemand governor. However, we ruled out that explanation based on the dynamic frequency, temperature, and power consumption measurements obtained during the user studies.

TABLE 1
Average System-Wide Power Consumption and User Satisfaction Rating

Run	Torcs					Quake3					Glest					Trackballs				
	Pwr.	Stdev	Sat.	Stdev	Imp.	Pwr.	Stdev	Sat.	Stdev	Imp.	Pwr.	Stdev	Sat.	Stdev	Imp.	Pwr.	Stdev	Sat.	Stdev	Imp.
ondemand	39.46	1.18	4.54	0.59	0.00%	38.89	1.54	4.88	0.34	0.00%	37.70	0.77	4.54	0.59	0.00%	42.30	0.51	4.46	0.72	0.00%
T-HAPPE	30.38	2.32	4.42	0.58	23.02%	28.03	0.46	4.88	0.34	27.93%	27.58	1.72	4.50	0.59	26.85%	26.99	2.66	4.38	0.58	36.18%
O-HAPPE	32.72	2.92	4.67	0.64	17.09%	28.38	1.27	4.92	0.28	27.03%	28.45	2.72	4.58	0.58	24.54%	30.07	6.05	4.58	0.50	28.91%
B-HAPPE	31.00	2.31	4.33	0.70	21.45%	27.23	0.88	4.92	0.28	29.97%	28.43	2.52	4.67	0.56	24.60%	29.52	5.69	4.58	0.50	30.21%

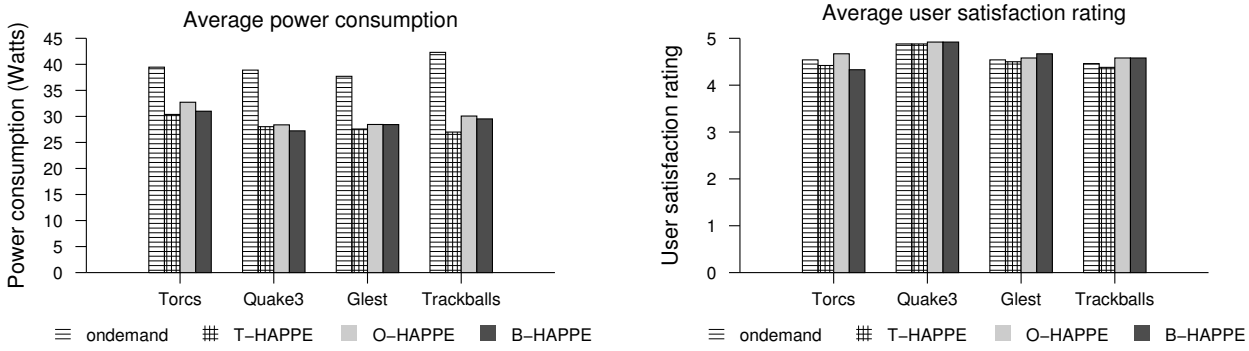


Fig. 5. Comparing HAPPE with Linux ondemand frequency controller.

TABLE 2
System-Wide Power Consumption of T61 at Different CPU Utilization Levels and Frequencies

	Power Consumption (Watts)									
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.8 GHz	24.40	24.80	25.22	25.03	25.37	25.81	25.81	26.41	26.60	26.69	26.74
1.2 GHz	25.73	26.02	26.34	25.95	26.92	27.40	27.80	27.92	27.94	28.15	28.55
1.6 GHz	26.14	26.73	27.30	27.93	28.55	29.51	29.86	30.00	30.50	31.19	32.27
2.2 GHz	29.35	30.01	30.81	31.91	32.77	33.79	34.87	36.00	37.25	38.52	40.18
2.3 GHz	30.72	32.01	33.07	34.75	35.55	36.78	39.06	40.52	42.24	43.62	45.04
	Temperature (C)									
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.8 GHz	41.15	41.45	41.70	42.00	42.00	42.00	42.00	42.00	42.00	42.18	42.80
1.2 GHz	42.05	42.00	42.00	42.48	42.83	43.00	43.43	43.83	44.02	44.78	45.00
1.6 GHz	44.33	44.00	44.65	44.82	45.52	46.05	46.83	47.67	48.13	48.90	49.78
2.2 GHz	47.92	48.45	48.82	50.25	51.37	52.67	53.73	55.02	56.58	58.12	59.70
2.3 GHz	52.68	53.00	53.12	54.40	55.42	57.02	58.52	60.28	61.92	64.05	65.35

Table 2 and Figure 6, we present the measured results of average system power consumption and CPU temperature of the laptop at all five frequencies and ten CPU utilization levels derived by running a testing application that allows fine-grained CPU utilization control [19]. CPU frequency was kept static during the experiments. We provide these results because (1) they suggest the possible power reduction by scaling CPU frequency, and (2) they may be used to derive a power model for the laptop, in which power consumption is a function of frequency (which influences voltage) and CPU utilization. We think this may be helpful to some potential readers.

To approximate the scenarios in the user study, we subjected CPU0 to 10 different load levels at each frequency level, for one minute each. Recall that the testing applications all require 3D graphics acceleration and are all graphic processing unit (GPU) intensive. To approximate the GPU load conditions during the user study, we also ran a 3D screen saver that stresses the GPU but not the CPU. We measured the power consumption of the whole system using the data acquisition card and sampled the temperature of CPU0 every second from the Linux ACPI interface.

As shown in the table, when CPU utilization is above 80%, the highest possible reduction in system-wide power consumption by decreasing CPU frequency from the highest level to the lowest level is 40.63%. However, this brute-force strategy is very likely to annoy the user because it does not consider any performance impact, whereas HAPPE can provide significant power reduction (25% in comparison with Linux ondemand) without degrading user satisfaction.

5.7 Variation Among Users

Figure 7 shows the variation among users by presenting

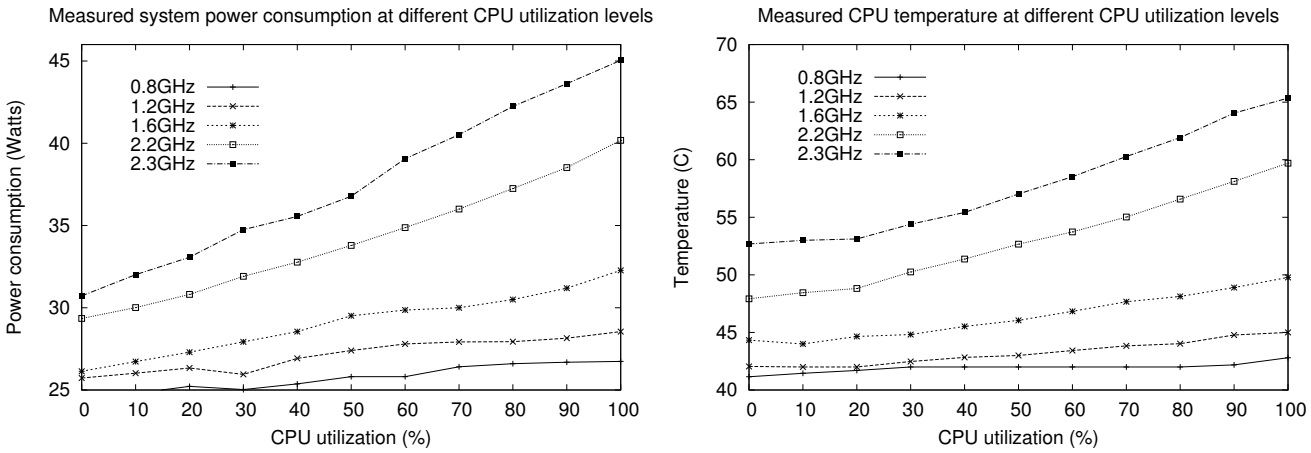


Fig. 6. System-wide power consumption and CPU temperature as a function of CPU utilization and frequency.

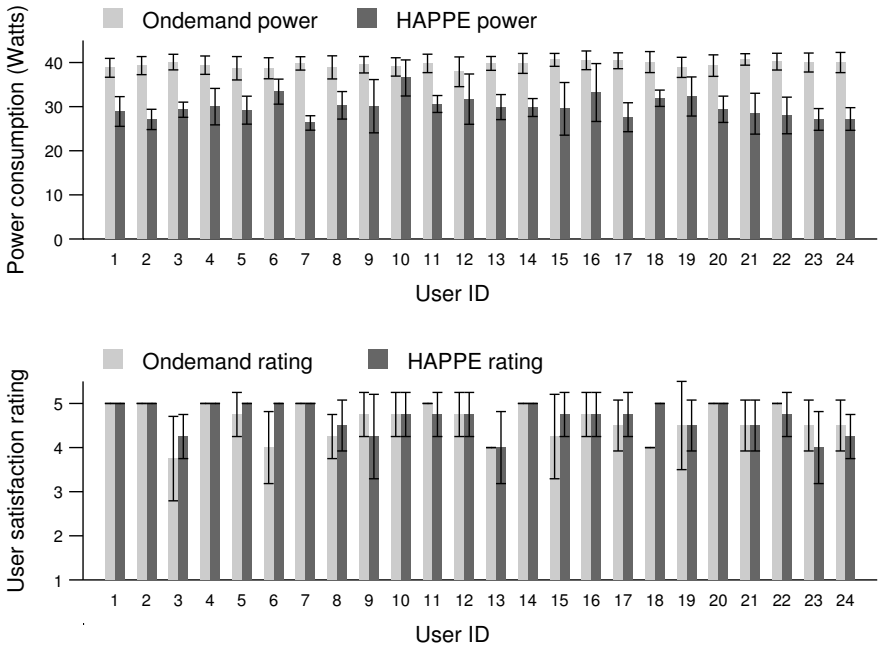


Fig. 7. Variation among users: power consumption and satisfaction.

the power consumptions and satisfaction ratings of all 24 users, comparing *O-HAPPE* to *ondemand*. For each user, we present the average power consumption and average satisfaction ratings for the four test applications, and show the standard deviations using error bars. As shown in the figures, there is significant variation among users. Some users are very sensitive to performance change as a result of varying processor frequency, and require high frequencies to be satisfied, resulting in high power consumption (e.g., user 10); others are less sensitive to the performance difference, and are satisfied at lower frequencies, resulting in lower power consumption (e.g., user 7).

5.8 Variation Among Applications for Same User

In this section, we present the variation among the performance requirements for different applications from the same user. We also provide additional evidence of an important point made in Section 3: performance demands depend strongly on user and application. If one were to consider variation among users but ignore variation among applications for the same user, the resulting DVFS technique would fail to satisfy the user or waste power. In this section, we analyze the importance of adapting dynamically to learned application- and user-dependent requirements.

Figure 8 illustrates a randomly-selected user playing Torcs (car racing game), and Quake3 (shooting game), under the control of *ondemand* and *O-HAPPE*. Under the control of HAPPE, when playing Torcs, the user

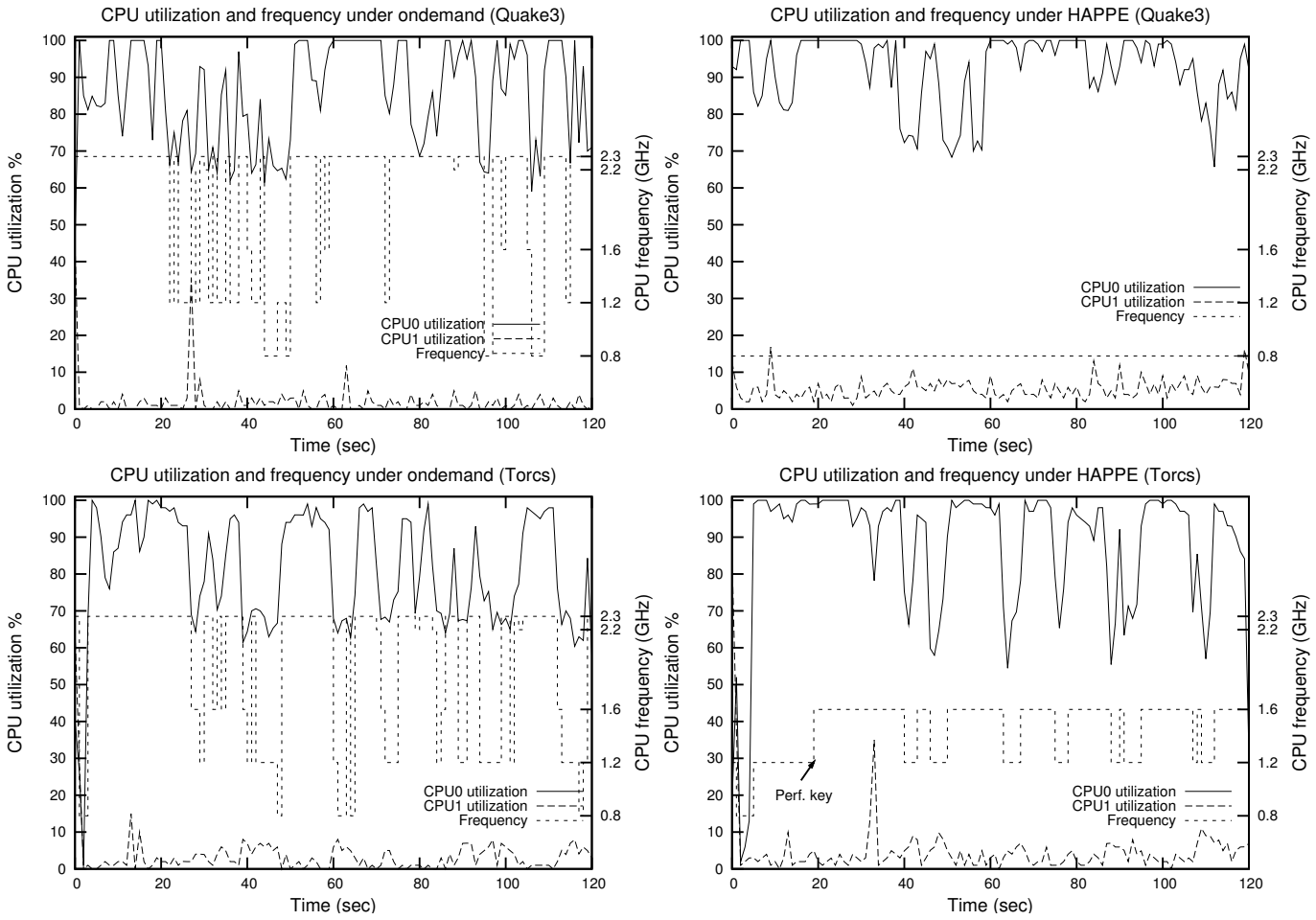


Fig. 8. Variation among different applications for same user.

pressed the performance key once at 20 seconds, and presumably became satisfied with the resulting 1.6 GHz frequency. Therefore, HAPPE set the CPU frequency to 1.6 GHz at high utilization levels and 1.2 GHz at low utilization levels. On the contrary, when playing Quake3, the same user was satisfied with the lowest frequency, and did not press the performance key at all. Therefore, HAPPE set the CPU frequency to 0.8 GHz at all times. In contrast, the ondemand governor did not distinguish between the two applications, and decided frequency using only CPU utilization. Because both applications are CPU-intensive, the frequency stayed at the highest level most of the time, wasting power without an impact on user satisfaction.

5.9 Variation Among Groups

In order to determine whether HAPPE is appropriate for users with varying degrees of familiarity with the experimental workload, we calculated the average power reduction it permitted for self-reported master level game players (23.16%), moderate-level game players (25.91%), and beginners (30.29%). The opportunity for saving power is smaller for users with greater expertise but remains substantial.

5.10 Analysis of User Feedback

A key question remains: "How much user input will HAPPE require to work well?" Put more formally, "How often will the power management parameters appropriate for a particular user–application combination change?" To answer this question, we analyzed the relationship between the frequencies of power and performance key presses and cumulative training/operating time. After the training phase, user input is not required but is still permitted. Therefore, if the user's performance requirements at different CPU utilization levels for this application do not change, the user does not need to press the power and performance keys again. Table 3 presents the number of key presses during the training phase and the operating phase, averaged over all users. There are usually more performance key presses and fewer power key presses during the training phase than the operating phase. On average, less than two performance key presses per minute are necessary for users to adapt to a desired frequency, during the first few minutes of the training phase.

We compare the average number of user inputs of HAPPE to that of UDFS [11]. UDFS requires 5.73 key presses per minute, averaged across all three evaluated

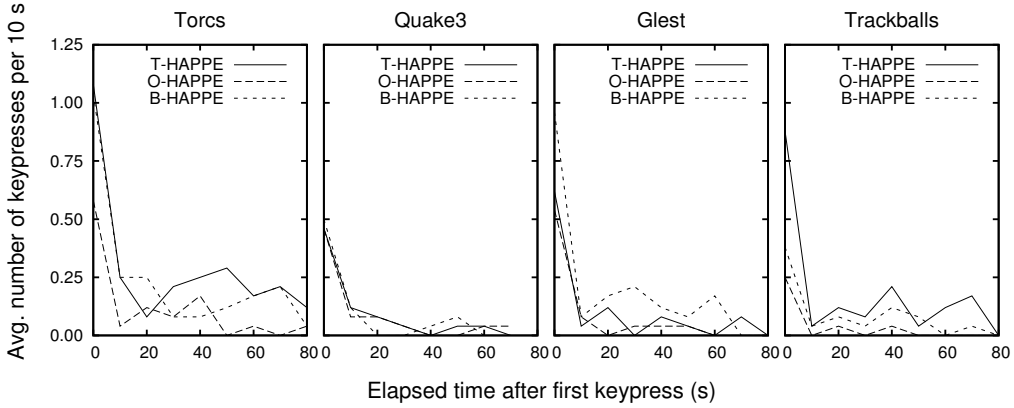


Fig. 9. Aggregated histogram of key presses.

TABLE 3
Average Number of Key Presses Per Minute

App.	Training phase		Operating phase	
	Perf. key	Power key	Perf. key	Power key
Torcs	1.13	0.40	0.46	0.27
Glest	0.36	0.23	0.19	0.34
Trackballs	0.69	0.14	0.14	0.16
Quake3	0.13	0.38	0.11	0.38

applications and all 20 users. In contrast, HAPPE requires 0.86 key presses per minute during the training phase and 0.51 key presses during the operating phase, averaged across all four evaluated applications and all 24 users. Furthermore, UDFS always slowly decreases frequency until the user expresses discomfort via key presses. Although UDFS adapts the rate of frequency decrease to user input, users are never free of key presses, i.e., after a certain time period, they must press the key again to change to a desirable frequency. In contrast, once a user is satisfied with the performance and frequency profile, HAPPE never requires key presses again. Figure 9 illustrates the frequency of the need to provide input via key presses as a function of time, averaged over all users for each of the four testing applications. The x-axis represents elapsed time after the first key press, and the y-axis represents the average number of key presses in every 10-second interval. As shown in the figure, the training phase generally requires more key presses. However, as time goes by, the number of key presses decreases dramatically during both the training phase and the operating phase.

6 Conclusions and Future Work

For CPU-intensive interactive applications, traditional DVFS policies based on fixed CPU utilization thresholds usually select unnecessarily high frequencies and therefore waste power. In this paper, we have presented HAPPE, a dynamic CPU DVFS controller that adapts CPU voltage and frequency to the performance requirements of individual users and applications. HAPPE uses a learning algorithm that creates a profile for each user and application. For each CPU utilization level, HAPPE learns the frequency necessary to satisfy the user. The learning algorithm trains the profile by accepting user key-press inputs during the first few minutes the first time the user runs the application. After the training phase, HAPPE does not require continued user input. We evaluated HAPPE with a study on 24 users and four test applications. Compared to the Linux default ondemand frequency governor, HAPPE reduces system-wide power consumption by 25% on average, without degrading user satisfaction.

Acknowledgments

This work was supported in part by the National Science Foundation under awards CNS-0720691 and CNS-0347941. We would like to acknowledge Xi Chen at University of Michigan for helping us to set up the CPU load test. We would also like to acknowledge Yue Liu at the University of Michigan for helping us to quantify the performance overhead of HAPPE.

References

[1] A. Mahesri and V. Vardhan, "Power consumption breakdown on a modern laptop," in *Proc. Wkshp. on Power Aware Computing Systems, Int. Symp. Microarchitecture*, Dec. 2004.

[2] V. Pallipadi and A. Starikovskiy, "The ondemand governor: Past, present, and future," in *Proc. Linux Symposium*, vol. 2, July 2006.

[3] "Windows native processor performance control," Microsoft Corporation, Tech. Rep., 2002.

[4] A. Gupta, B. Lin, and P. A. Dinda, "Measuring and understanding user comfort with resource borrowing," in *Proc. Int. Symp. on High Performance Distributed Computing*, June 2004, pp. 214–224.

[5] A. Shye, B. Ozisikyilmaz, A. Mallik, G. Memik, P. A. Dinda, R. P. Dick, and A. N. Choudhary, "Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction," in *Proc. Int. Symp. Computer Architecture*, June 2008, pp. 427–438.

[6] H. Sasaki, Y. Ikeda, M. Kondo, and H. Nakamura, "An intra-task DVFS technique based on statistical analysis of hardware events," in *Proc. Int. Conf. Computing Frontiers*, May 2007.

[7] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 2004.

[8] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks, "A dynamic compilation framework for controlling microprocessor energy and performance," in *Proc. Int. Symp. Microarchitecture*, Nov. 2005, pp. 271–282.
[9] R. Xu, D. Mossé, and R. Melhem, "Minimizing expected energy consumption in real-time systems through dynamic voltage scaling," *ACM Trans. on Computer Systems*, Dec. 2007.
[10] J. R. Lorch and A. J. Smith, "Using user interface event information in dynamic voltage scaling," University of California at Berkeley, Tech. Rep., Aug. 2002.
[11] B. Lin, A. Mallik, P. Dinda, G. Memik, and R. P. Dick, "User and process-driven dynamic voltage and frequency scaling," in *Proc. Int. Conf. Performance Analysis of Systems and Software*, Apr. 2009, pp. 11–22.
[12] K. Flautner and T. Mudge, "Vertigo: Automatic performance-setting for Linux," in *Proc. Int. Symp. Operating Systems Design and Implementation*, Dec. 2002.
[13] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer, "Using latency to evaluate interactive system performance," in *Proc. Int. Symp. Operating Systems Design and Implementation*, Oct. 1996.
[14] A. Mallik, B. Lin, P. Dinda, G. Memik, and R. P. Dick, "User driven frequency scaling," *IEEE Computer Architecture Ltrs.*, vol. 5, no. 2, pp. 16–19, Dec. 2006.
[15] L. Yan, L. Zhong, and N. K. Jha, "User-perceived latency driven voltage scaling for interactive applications," in *Proc. Design Automation Conf.*, June 2005.
[16] A. Mallik, J. Cosgrove, R. P. Dick, G. Memik, and P. Dinda, "PICSEL: Measuring user-perceived performance to control dynamic frequency scaling," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2008, pp. 70–79.
[17] A. Shye, Y. Pan, B. Scholbrock, J. S. Miller, G. Memik, P. Dinda, and R. P. Dick, "Power to the people: leveraging human physiological traits to control microprocessor frequency," in *Proc. Int. Symp. Microarchitecture*, Nov. 2008, pp. 188–199.
[18] A. Rahmati, A. Qian, and L. Zhong, "Understanding human-battery interaction on mobile phones," in *Proc. Int. Conf. Human Computer Interaction with Mobility Devices and Services*, Sept. 2007.
[19] P. Dinda and D. O'Hallaron, "Realistic CPU workloads through host load trace playback," in *Proc. Wkshp. on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 2000.

Lei Yang received her Ph.D. degree in 2008 from Northwestern University, Dept. of Electrical Engineering and Computer Science. She also holds a Bachelor's degree from Peking University and a Master's degree from Northwestern University. Her work on online memory compression won a Computerworld Horizon Award for high-impact commercially-used information technology. She has published in a wide range of embedded system design topics including data compression, memory hierarchy design, and user satisfaction driven power management. She is now a software engineer at Google.

Robert Dick (S'95–M'02) is an Associate Professor of Electrical Engineering and Computer Science at the University of Michigan. He received his Ph.D. degree from Princeton University in 2002 and his B.S. degree from Clarkson University in 1996. He worked as a Visiting Professor at Tsinghua University's Department of Electronic Engineering in 2002, as a Visiting Researcher at NEC Labs America in 1999, and was on the faculty of Northwestern University from 2003–2008. Robert received an NSF CAREER award and won his department's Best Teacher of the Year award in 2004. In 2007, his technology won a Computerworld Horizon Award and his paper was selected as one of the 30 in a special collection of DATE papers appearing during the past 10 years. His 2010 work won a Best Paper Award at DATE. He is an Associate Editor of IEEE Trans. on VLSI Systems, a Guest Editor for ACM Trans. on Embedded Computing Systems, was the Technical Program Committee Co-Chair of the 2011 International Conference on Hardware/Software Codesign and System Synthesis, and serves on the technical program committees of several embedded systems and CAD/VLSI conferences.

Gokhan Memik (S'98–M'03) is an Associate Professor at the Electrical Engineering and Computer Science Department of Northwestern University. He received the B.S. degree in Computer Engineering in 1998 from Bogazici University and PhD in Electrical Engineering from University of California at Los Angeles (UCLA) in 2003. He is the author of 2 book chapters and over 100 refereed journal/conference publications. Papers co-authored by him have been nominated for a best paper award at DAC (2005) and MICRO (2008) and won the Best Student Paper Award at Supercomputing (2007). He has served in over 30 program committees, was the co-chair for the Advanced Networking and Communications Hardware Workshop (ANCHOR) and the program co-chair of 2007 International Symposium on Microarchitecture (MICRO-40). He is also an associate editor of International Journal on Reconfigurable Computing. Gokhan Memik is the recipient of the Wissner-Slivka Junior Chair (2006–2009), National Science Foundation CAREER Award (2008–2013), and Department of Energy Early CAREER PI Award (2005–2008).

Peter Dinda (S'92–M'00) is a professor in the Department of Electrical Engineering and Computer Science at Northwestern University, and head of its Computer Engineering and Systems division, which includes 17 faculty members. He holds a B.S. in electrical and computer engineering from the University of Wisconsin and a Ph.D. in computer science from Carnegie Mellon University. He works in experimental computer systems, particularly parallel and distributed systems. His research currently involves virtualization for distributed and parallel computing (v3vee.org and virtuoso.cs.northwestern.edu), programming languages for sensor networks (absynth-project.org), and empathic systems for bridging individual user satisfaction and systems-level decision-making (empathicsystems.org). More information can be found at pdinda.org.