INTEGRATED GLOBAL AND PER-APPLICATION CROSS-LAYER ADAPTATIONS FOR SAVING ENERGY

BY

VIBHORE VARDHAN

B.S., Purdue University, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

ABSTRACT

This thesis concerns the design of energy-efficient multimedia mobile devices, explored in the GRACE project. The GRACE system saves energy by performing a coordinated adaptation in all system layers in response to changing application requirements and resource availability. The GRACE framework proposes a hierarchical solution to the cross-layer coordinated adaptation problem, where the system utilizes global (multiple application) and per-application adaptation to reduce energy. Global adaptation works at a large time granularity of several hundred frames – it uses long-term information about multiple applications to find a long-term system wide optimal configuration. Per-application adaptation works at the granularity of an application frame – it uses frame-level information to respond to short-term variations in resource demand to further save energy.

This thesis focuses on the benefits of per-application adaptation when added to global adaptation. We consider adaptations in the CPU and the application, with the goal of minimizing CPU and network transmission energy, subject to CPU and network bandwidth constraints. Our results on the GRACE prototype system show that the cross-layer hierarchical adaptation strategy in GRACE is effective. Per-application adaptation results in significant energy benefits (up to 50%) in several scenarios, over and above the benefits from global adaptation.

ACKNOWLEDGMENTS

I would like to thank my advisor, Sarita Adve, for her invaluable guidance and support throughout this work, without which this work would not have been possible. I would also like to thank all the members of the GRACE group and the RSIM group for so many insightful discussions and for the feedback that they have given me on several occasions. In particular, I would like to thank Dan Sachs and Wanghong Yuan for building the earlier prototypes of the GRACE system, and for providing answers to all my questions. Last but not the least, I would like to thank my parents and my brother for their never-ending love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Mobile devices running soft real-time multimedia applications are becoming an increasingly important computing platform. Such systems are often limited by their battery life; therefore, minimizing their energy usage has become a primary design goal. A widely used technique to save energy is to adapt the system in response to changes in application demands and resource availability. Researchers have proposed such adaptations in all layers of the system e.g., hardware, application, operating system, and network.

A system with multiple adaptive layers requires careful coordination of these adaptations to reap their full benefit. The Illinois GRACE project (Global Resource Adaptation through CoopEration) has proposed such a coordinated cross-layer adaptation framework [1, 2].

A cross-layer adaptive system must balance the conflicting demands of adaptation scope and time scale. Ideally, it should invoke both *global* and *frequent* adaptation that coordinates all layers in response to all changes in the system. Unfortunately, global adaptation can be expensive and so must be infrequent, but long intervals between adaptation risks inadequate response to intervening changes. To balance this conflict, the GRACE framework has proposed a hierarchical approach, performing expensive global adaptations occasionally, and inexpensive limited-scope adaptations frequently [1, 2].

Our previous work reported results on an initial GRACE protoype, GRACE-1, and demonstrated the benefits of cross-layer coordinated adaptation [1]. This thesis reports on the next generation GRACE-2 system and focuses on the benefits of the *hierarchical approach* for coordinated adaptation.

Specifically, the GRACE framework proposes three levels of adaptation, exploiting the natural

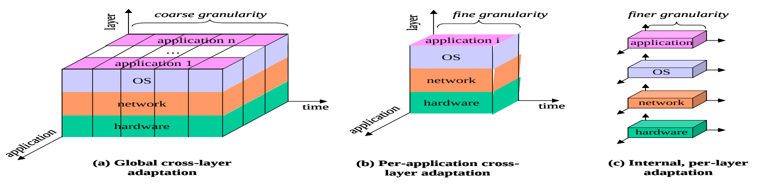


Figure 1.1 GRACE adaptation hierarchy. (This thesis does not implement all adaptations in the figure; specifically, the network layer is minimal and simulated.)

frame boundaries in periodic real-time multimedia applications (Figure 1.1 [2]). *Global* adaptation considers all applications and system layers together, but only occurs at large system changes (e.g., application entry or exit). *Per-application* adaptation considers one application at a time and is invoked every frame, adapting all system layers to that application's current demands. *Internal* adaptation adapts only a single system layer (possibly considering several applications) and may be invoked several times per application frame. All adaptation levels are tightly coupled, ensuring that the resource allocation decisions made through global coordination are respected by the limited-scope adaptations.

The goal of this thesis is to ascertain the benefits of per-application adaptation over global adaptation in a cross-layer adaptive system. Our new implementation, GRACE-2, incorporates global and per-application adaptations in the CPU and application, along with global and internal adaptations in the soft real-time scheduler. It respects the constraints of CPU utilization and network bandwidth, while minimizing CPU and network transmission energy.

GRACE-2's global optimizer chooses, for each application, an application configuration, a CPU configuration, and the corresponding CPU time, network bandwidth, and energy budgets. The optimization objective is to minimize the total system energy, subject to the constraints of total CPU time and network bandwidth available. Since this optimization process is expensive, global adaptation is invoked only when an application enters or leaves the system. Consequently, it needs to make a long-term, usually conservative, prediction of the applications' resource requirements.

GRACE-2's per-application adaptation responds to interframe variations in the application's CPU and network requirement. It is invoked at the start of each application frame, and makes a prediction of the resource requirements for the next frame for different configurations. Based on this prediction and the budgets allocated by the global adaptation, the per-application adaptation chooses the configuration for the application and the CPU for the next frame that would minimize energy within the allocated budgets. The budget allocation is also modulated by the soft real-time scheduler in response to frame-to-frame usage by the applications. Per-application adaptation is less expensive because it only optimizes one application at a time.

We have implemented GRACE-2 on a Pentium-M based laptop system running Linux 2.6.8-1, with a real adaptive CPU, fully implemented adaptive applications and soft real-time CPU scheduler, but a simulated network layer. For CPU adaptation, we use dynamic voltage and frequency scaling on the Pentium-M processor. For applications, we use H.263 video encoders and perform adaptations first proposed in [3]. These adaptations vary the CPU time and network bandwidth requirement of the encoder, to minimize the sum of the CPU and network transmission energy, *without perceptibly changing the video quality.* We perform experiments with several video streams and different network bandwidth availability, to cover four scenarios, where one or both of the CPU and network are lightly or heavily loaded.

To our knowledge, GRACE-2 is the first implemented system that performs adaptations (1) at multiple time scales (global and per-application), (2) in multiple layers (application, hardware, and scheduler), (3) running multiple applications, and (4) in response to multiple constraints (CPU and network bandwidth). Previous work that has considered adaptations in multiple system layers has primarily focused on adaptation at a single time scale (usually global) and/or with one application at a time and/or with one constraint (CPU or network bandwidth) (see Chapter 8 for a full discussion of related work).

This work is the first to comprehensively explore the benefits of per-application adaptation over global adaptation in a cross-layer adaptive system over a variety of scenarios representing different resource constraints. Our results show that per-application adaptation in GRACE-2 provides significant energy savings when added to global adaptation (up to 46%). While the magnitude and source of the benefits depends on the resource constraints in the system, we can expect a typical mobile device to operate under all of the conditions evaluated in this thesis. Further, given the low overhead of per-application adaptation and the relatively low added system implementation complexity over and above a system with global adaptation, the benefits achieved are clearly worthwhile to exploit.

CHAPTER 2

LAYER ADAPTATIONS AND MODELS

2.1 CPU

Adaptations: We study dynamic voltage and frequency scaling or DVFS [4] for CPU adaptation since it is the most widely used software-controlled adaptation technique in current processors. The Pentium-M based system we use supports five frequencies (and corresponding voltages) – 600, 800, 1000, 1200, 1300 MHz.

Energy model: We use the following model for CPU energy with DVFS, where C_{eff} is effective capacitance, f is the frequency, and V is the corresponding voltage.

$$\text{Dynamic power at frequency} f = C_{eff} \times V^2 \times f \tag{2.1}$$

$$\text{Energy at frequency f} = \text{Power} \times \text{Execution Time} \tag{2.2}$$

$$= C_{eff} \times V^2 \times \text{Execution cycles} \tag{2.3}$$

The above model does not include static or leakage power. With technology scaling and increasing frequencies, static power is becoming increasingly important; however, current-leakage models are fairly complex and we did not include them for this study. We derive C_{eff} by using published numbers for the Pentium-M current and voltage at its highest frequency which gives the maximum power ($\sim$ 25 W) – this does not incorporate the impact of application-dependent clock gating. To determine the voltage at each frequency, we assume $V \propto f$, and use the published V, f numbers for the Pentium-M at 1300 MHz to approximate the proportionality constant.

The net effect is that we can derive the energy of an application frame from its execution cycles. All of the above assumptions are commonly made in the literature. We do not believe any of them has an impact on the key focus of the thesis, i.e., the effectiveness of hierarchical adaptation. Nevertheless, in the future, we will use a multimeter to directly measure the average power for each frame.

Execution time: For the multimedia applications studied here, past work has shown that the number of execution cycles for a given frame for a given application configuration is roughly independent of frequency [5]. This is because the applications studied generally hit in the cache and do not see much memory stall time. The execution time therefore scales roughly linearly with frequency, and execution cycles stay roughly constant with changing frequency.

Emulating continuous DVFS (CDVFS): Current processors support a small number of DVFS points. This limits the benefits of DVFS adaptations [6] as the adaptation algorithm rounds up to the next DVFS point in cases where it needs a frequency that lies between two DVFS points. This rounding up results in more energy being used than necessary. We reduce this inefficiency by emulating a continuous set of points, which we refer to as continuous DVFS or CDVFS. The basic idea, based on [7], is that if we want to run at a frequency that is not a DVFS point, we run at the DVFS point below the desired frequency for some time, and the DVFS point above it for the remaining time. Specifically, assume that c cycles need to be executed at the calculated frequency f and the lower and upper DVFS points are f_l and f_h, respectively. This emulation executes c_1 cycles at speed f_l and c_2 cycles at speed f_h, such that

$$c_1 + c_2 = c \tag{2.4}$$

$$\frac{c}{f} = \frac{c_1}{f_l} + \frac{c_2}{f_h} \tag{2.5}$$

2.2 Network (Nonadaptive)

We assume a nonadaptive (simulated) network layer with fixed available network bandwidth. We model network transmission energy using a fixed energy/byte cost:

$$\text{Network Energy} = \text{EnergyPerByte} \times \text{BytesTransmitted} \tag{2.6}$$

Table 2.1 Network bandwidth and energy/byte values.

Bandwidth (Mbps)	2	5.5	11
Energy per byte (e^{-6} J)	4	2	0.8

In our evaluations, we study the effects of different bandwidth values, ranging from 2 Mbps to 11 Mbps. These values represent the bandwidth available in an IEEE 802.11b wireless network. For each bandwidth level, the fixed transmission cost is modeled using the energy consumption of a Cisco Aironet 350 series PC card [8]. Table 2.1 summarizes the specific values we use.

We believe the range of network configurations we study represent reasonable scenarios encountered in practice. Responding to variations in network bandwidth with an adaptive network layer is part of our ongoing work and lies outside the scope of this thesis.

2.3 Application

We consider periodic soft real-time applications or tasks. An application releases a job or a *frame* at the end of each period. We use H.263 video encoders as our base applications and apply adaptations described in [3] and discussed below.

Adaptations: The application adaptations we study enable a tradeoff between the amount of CPU computation (i.e., CPU energy) for the number of bytes transmitted (i.e., network transmission energy), to minimize the total CPU+network transmission energy (referred to as system energy henceforth). The appropriate tradeoff varies dynamically, depending on the video stream, the load on the system, and the relative expense of CPU energy per cycle to network energy per byte. The CPU energy per cycle in our system also depends on the CPU configuration chosen.

The adaptations we consider work at the granularity of a single frame of the input stream. They enable dropping DCT computations and motion searches based on specified thresholds, which can be changed for the next frame by the global or per-application adaptor. For motion search, at each step, we compare the SAD (sum of absolute difference) generated with an externally specified threshold. If the SAD for a candidate motion vector is less than the current threshold, that candidate is declared the best match for the macroblock. Because the search begins with the null motion vector (0,0), motion search is essentially disabled if the specified threshold is sufficiently high.

For DCTs, we extend the H.263 specification by adding a bit before each 8 x 8 DCT block specifying whether or not that block was transformed. If the sum of the absolute values of each element of the 8 x 8 block exceeds the threshold, or the block belongs to an Intracoded macroblock, the block is DCT-transformed and a '1' is emitted into the bitstream. Otherwise, a '0' is emitted, and no DCT is performed. No change is made to the subsequent quantization and VLC (variable length coding) steps.

The net effect of the above adaptations is that, by changing the thresholds, we can vary the bit rate to be transmitted and the number of cycles required to encode each frame by approximately a factor of two. Although these adaptations (particularly dropping DCTs) can reduce the PSNR (pseudo signal to noise ratio) of the encoded stream somewhat, this can be compensated for by adjusting the quantizer step size Q to keep the PSNR of the output stream roughly invariant.

Thus, the adaptive encoder can be scaled between a highly compute-intensive but lower bit rate configuration to a less compute-intensive, higher bit rate configuration, *without affecting the way the stream is decoded or the quality of the video as seen by the user.*

To simplify the configuration space, we chose 16 different configurations to map onto the two thresholds. These 16 configurations include all combinations of four different DCT thresholds (0, 350, 700, and 20 000), and four different motion-search thresholds (0, 750, 2500, and 20 000).

Deadline misses and frame drops: A frame that does not complete encoding or transmission of all its bytes by the end of the ensuing period is said to miss its deadline. If the application misses its deadline for one frame, the encoding/transmission for that frame continues in the next period, borrowing from the budget allocation of the next frame. If the application misses the deadline for two frames in a row, then the next frame is entirely skipped or dropped (i.e., it incurs no CPU computation or network transmission), enabling the application to catch up on its previous frame overruns.

Since this is a soft real-time application, we assume that we may miss the deadline for or drop a total of up to 5% of all frames, without affecting the video quality. Thus, we do not distinguish between deadline misses and frame drops; henceforth, we use the term deadline misses to refer to both frames that miss their deadlines and frames that are completely dropped.

2.4 O.S. Scheduler

We assume an earliest-deadline-first (EDF) soft real-time scheduler for CPU time. The scheduler is responsible for enforcing budget allocations for both CPU time and use of the network bandwidth. To reduce deadline misses due to underruns and overruns, the scheduler performs an internal adaptation called budget sharing as proposed in [9].

Budget sharing allows applications to use unused budget from other applications. This sharing can, to some extent, handle overruns caused by factors such as imperfect prediction of cycle usage. To support this type of sharing, the EDF scheduler maintains a record of all unused budgets, and their expiration times. When an application is scheduled, the scheduler first tries to exhaust any unused budget before charging the elapsed cycles to the application. The unused budget can be given to an application only if the expiration time of the budget is less than the deadline for the application. Also, unused budgets expire with time and may no longer be used. The details of the algorithm can be found in [9]. Similar to the CPU budget sharing, we also exploit unused bandwidth sharing between applications.

CHAPTER 3

GLOBAL AND PER-APPLICATION ADAPTATION ALGORITHMS

We next describe GRACE-2's global and per-application adaptation algorithms. Figure 3.1 summarizes the full system.

3.1 Global Adaptation

The global adaptation algorithm is invoked on large changes in the system, e.g., when an application enters or exits the system. As input, the algorithm receives the resource requirements for each combination of application and CPU configuration. Resources include CPU time (equivalently, CPU utilization), network bandwidth (equivalently, bytes to be transmitted), and energy (the sum of CPU and network transmission energy).

Section 3.3 describes how resource requirements are determined for an application/CPU configuration combination. These requirements should be representative of the behavior of the system until the next global adaptation is invoked. The algorithm (conservatively) assumes that each frame of an application has the same resource requirement for a given combination of application/CPU

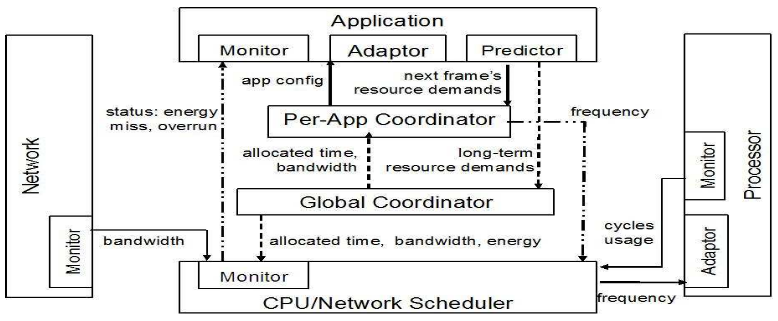


Figure 3.1 The GRACE-2 System.

configuration.

Given the above inputs, the global adaptation algorithm must then choose, for each application, the combination of the application and CPU configuration such that (i) the total system energy usage is minimized, and (ii) the CPU and network resource requirements for all the applications (running with the chosen configurations) can be met.

More formally, for application i, let C_i be a chosen CPU and application configuration combination, Energy_{i,C_i} be the energy consumed by a frame of application i with configuration C_i, Time_{i,C_i} be the time taken by a frame of application i under configuration C_i, Bytes_{i,C_i} be the bytes generated for transmission by a frame of application i under configuration C_i, and Period_i be the period for application i. Let there be a total of N_{apps} applications in the system. Assume an earliest-deadline-first (EDF) real-time scheduling algorithm and assume that the available total network bandwidth is known.

Then the global adaptation algorithm is required to choose the CPU and application configuration C_i for each application i to:

minimize

$$\sum_{i=1}^{N_{apps}} \text{Energy}_{i,C_i} \tag{3.1}$$

subject to the EDF CPU scheduling constraint:

$$\sum_{i=1}^{N_{apps}} \frac{\text{Time}_{i,C_i}}{\text{Period}_i} \leq 1 \tag{3.2}$$

and the network bandwidth constraint:

$$\sum_{i=1}^{N_{apps}} \frac{\text{Bytes}_{i,C_i}}{\text{Period}_i} \leq \text{Available network bandwidth} \tag{3.3}$$

The above optimization problem is essentially a multidimensional multiple-choice knapsack problem (MMKP) [10], and known to be NP-hard. We report results using two techniques to solve this problem. To give global adaptation the best showing, we report results with a brute force approach that performs an exhaustive search for a minimal energy configuration that meets the CPU and network constraint (with one optimization described below). This approach is impractical for a real system. For a more practical, but possibly suboptimal solution, we also report results with a heuristic algorithm for MMKP proposed by Moser et al. [10], based on Lagrangian techniques.

To reduce the complexity of both the exhaustive search and the heuristic algorithm, we make one optimization. We choose the same frequency (CPU configuration) for all applications. This enables solving the MMKP problem separately for each supported frequency – we now need to search only for the best application configuration for each application for each supported frequency (vs. searching over the cross-product of frequency and application configurations). We then pick the frequency that provides the minimum energy with the chosen application configurations at that frequency. We justify the assumption of running all applications at the same frequency by Jensen's inequality [11]: if the energy per unit time is a convex function of frequency, then the best frequency setting will be either a single point for all applications, or, if the CPU does not support a frequency that exactly matches the processor's workload, a combination of adjacent frequencies that match the required computation cycles.

At the end of the above process, the algorithms provide, for each application, the assigned CPU frequency (same for all applications) and the selected application configuration. These configuration allocations imply an allocation of the required resources to the corresponding applications. It is possible that at this point, there is some leftover network bandwidth and CPU utlization. We further divide these leftover resources among the applications in proportion to their current allocation.

To exploit the benefit of emulating continuous DVFS as discussed in Section 2.1, we perform the following final optimization. The leftover allocated CPU utilization above can be further converted into a reduction in frequency. The resulting frequency is not one that is directly supported, but can be emulated using the continuous DVFS emulation discussed in Section 2.1.

A system that runs with only global adaptation uses the frequency and application configurations as chosen by the global algorithm. A system that runs with per-application adaptation uses the resource allocation of the global algorithm to determine the CPU and application configuration each frame, as described next.

3.2 Per-Application Adaptation

The per-application adaptation algorithm is simple, and is derived from [3]. It is invoked at the start of an application frame with the following as input: (1) the resource allocation for each application, as determined by the global adaptation algorithm and (2) the resource requirements for the next frame for each combination of application and CPU configuration. (Section 3.3 discusses how we determine these.)

Given the above input, the algorithm simply chooses the application and CPU configuration combination that has the least energy, and whose CPU time and network bandwidth requirement is within its allocation. If such a combination is not found, then we use the application and CPU configuration of the last frame (likely leading to a deadline miss). The complexity of this algorithm is order of the product of the number of application and CPU configurations.

3.3 Predicting Resource Usage

Both the global and the per-application adaptation algorithms need to predict the per-frame resource requirement for each combination of application and CPU configuration. The resources are CPU time (equivalently CPU utilization), bytes to be transmitted on the network (equivalently network bandwidth), and energy.

Predictions for Global

For the global adaptation, the predictions need to be representative of the resource usage until the next global adaptation is invoked. This could be thousands of frames. Following previous work on resource allocation and scheduling for soft real-time multimedia applications [12], we use profiling of several frames to determine the resource usage for global allocation. Per-frame CPU time and network bytes can be directly estimated during profiling. These estimates also allow estimation of energy using the models in Section 2.

We need to determine per-frame execution cycles and network bytes transmitted that would be representative of the frames executed before the next global adaptation. Based on the models in Section 2, we can then convert these to per-frame execution time, network bandwidth, and energy. Since we assume a 5% deadline miss rate is acceptable (Section 2.3), we use the cycle counts (bytes) from the frame that falls in the 95th percentile of the cycle counts (bytes) from all the profiled frames, to determine the execution time (network bandwidth). However, for energy, we are concerned with minimization and not meeting a constraint. We therefore use the average cycle count and bytes from the profiled frames to estimate the total energy (based on the models in Section 2).

We note that the 95th percentile frame above may be different for cycle counts and for bytes. Therefore, using the 95th percentile value for execution time and network bandwidth is optimistic – in the worst case, 5% of the frames will have higher cycle counts and another 5% will have higher byte counts, potentially leading to 10% deadline miss rate. Furthermore, once a frame is dropped, we find that the compression algorithm is less effective for the next frame, resulting in even further increases in byte counts. Our results show all of these effects in the form of large deadline misses for global adaptation. Nevertheless, we use the 95th percentile to give global adaptation a good showing; we use budget sharing in conjunction with global adaptation to bring down the deadline misses to an acceptable level.

Naively, for a given application, we would need a separate profile for each combination of application and CPU configuration. However, as mentioned earlier, for the multimedia applications studied here, the number of execution cycles for a given frame for a given application configuration is roughly independent of frequency [5]. We therefore profile each application configuration at a single CPU frequency to determine both the number of execution cycles and the number of bytes (which is also clearly independent of frequency).

In practice, we expect to use on-line profiling of a few hundred frames to measure the above execution cycles and bytes, as recommended in previous work [13]. For long streams, this profiling poses a negligible overhead. In our experiments, since our streams are short and since we would like to give global adaptation the best showing, we profiled the entire stream off-line to determine the 95th percentile values above.

Predictions for Per-Application

Per-application adaptation is invoked each frame and therefore needs to make its resource predictions only for the next frame. Conceptually, this should be easier than the long-term predictions required by the global adaptation. In prior work, researchers have proposed the use of various history-based techniques, where the behavior of the last few frames is used to predict that of the next frame (e.g., the maximum execution time of the last five frames is used to predict the time for the next frame [14]). However, one key difference for our work is that our application itself is adaptive. The history of the past frames may be for different configurations of the application, and it is unclear how it can be used to predict the behavior of the next frame for yet other configurations.

Sachs et al. [3] proposed the following prediction strategy using off-line profiling. They generate the execution cycle predictor off-line by repeatedly encoding one or more sequences (for a fixed hardware frequency), randomly changing the encoder configuration at each frame. This off-line run generates several points for every pair of (previous, next) encoder configurations, mapping the number of cycles in the previous frame to the those in the next frame. The predictor is generated by fitting a function in the least-squared sense, for every pair of (previous, next) configurations.

The same scheme is followed to generate a predictor for the number of bytes. To avoid deadline misses, we conservatively add an adaptive leeway into the predicted values for both execution cycles and bytes. In particular, we start with a leeway of 1.1 (10%) for the CPU and decay it by 0.005 (5%) every frame until we miss a deadline, in which case we reset it to 1.1. The leeway added into the bytes starts of at 1.2 (20%), and decay it by 0.1 (50%). We decay the byte leeway in two frames as the sole purpose of the byte leeway is to force the per-application coordinator to pick an application configuration that does more compression.

When the per-application adaptation is invoked, it determines the cycle count and byte count for each application configuration for the next frame by using the appropriate predictor, given the knowledge of the previous frame's application configuration, actual cycle count, and actual byte count. The cycle and byte count then directly give CPU time, network bandwidth, and total (CPU+network) energy.

CHAPTER 4

IMPLEMENTATION

We have implemented the GRACE-2 prototype on an IBM ThinkPad R40 laptop system. This system has a single Intel Pentium-M processor, which features Intel's Enhanced SpeedStep technology [15], and supports five DVFS points: 600, 800, 1000, 1200, and 1300 MHz. The processor can be made to transition between DVFS points at runtime by the operating system. We use the Linux kernel 2.6.8-1, modified as described below.

The GRACE-2 operating system components are implemented as a set of modules and patches that hook into the Linux kernel 2.6.8-1. Figure 4.1 gives an overview of the software architecture. We discuss the main components next.

4.1 Global Coordinator

The global coordinator implements the global adaptation algorithm described in Section 3.1. It is implemented as a separate user-level process. This decision was based on two constraints. First, the coordination computation involves floating-point type, which currently is not supported in the Linux kernel module. Second, a user-level global coordinator runs at a lower priority than the

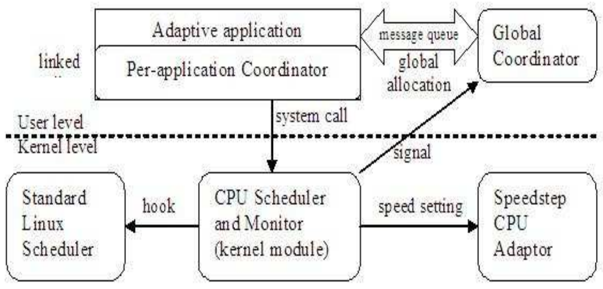


Figure 4.1 GRACE-2 software architecture.

applications, thereby ensuring that the expensive process of global optimization does not supercede applications. The global coordinator communicates with the per-application coordinator via a message queue.

4.2 Per-Application Coordinator

The per-application coordinator implements the per-application adaptation algorithm described in Section 3.2. It is designed as a generic function that can be linked with the application at compile time. Linking the per-application coordinator with the application implies that the CPU scheduler sees the per-application adaptation as part of the application. This has two main advantages over having the per-application coordinator be part of (for example) the global coordinator module: (1) the application can be charged for the cycles that were used by its per-application coordinator, and (2) we can ensure that per-application adaptation occurs at the start of every frame. The advantage over implementing it in the kernel is in the reduced number of system calls. However, linking the per-application coordinator with the application could make the coordinator vulnerable to malicious applications, and thus nontrustworthy. We can work around this problem by sending the global allocations for each application to the CPU scheduler, and having the CPU scheduler enforce that each application is using only its allocation of resources.

4.3 CPU Scheduler

We use an EDF scheduling policy based Soft Real-Time (SRT) CPU scheduler (Section 2.4).
Invocation of the scheduler and new system calls: This scheduler is invoked either when a timer started by the scheduler itself expires, or when an application makes a system call.

The scheduler may set the timer for several reasons. For example, before starting a new application frame, the scheduler sets a timer to expire when the application's budget runs out, to enable handling overruns correctly. At the end of an application frame, the scheduler sets a timer to expire at the start of a new period for the application, to schedule its next frame.

GRACE-1 used the standard 10 ms resoulution Linux kernel timer, but to implement per-application adaptation we need a timer with higher resolution. In GRACE-2, we use the High Res

Posix timers [16] to get a low-overhead, high-resolution timer.

The application may also invoke the scheduler for various reasons. We have added five new system calls to enable the application to interact with the CPU scheduler:

1. *EnterSrt*: This is invoked when the application first joins the system. It registers the application with the CPU scheduler as a new SRT task. The scheduler initializes its data structures for the new application and inserts it into the SRT task list. It also signals the global coordinator to perform the global optimization.

2. *BeginJob*: This is invoked at the start of a new frame. The per-application coordinator passes its chosen CPU frequency to the scheduler. The scheduler refreshes the budget available for the application's new frame (based on the time allocation made by the global coordinator) and invokes the CPU adaptor to change the CPU frequency (by performing a write to a special CPU register MSR_IA32_PERF_CTL).

3. *FinishJob*: This is invoked when the application has finished encoding its frame. The CPU scheduler gets the resource usage (elapsed cycles, energy) from the CPU monitor, checks for deadline miss, and sends the resource usage and miss status information back to the application. The monitor checks the cycle usage by using the *rdtscll* function in the Linux Kernel to read the current processor cycle count. It estimates the CPU energy using the model described in Section 2.

4. *WaitNextPeriod*: The application invokes this when it is done with all of the book-keeping for its past frame, notifying the scheduler that it is ready to give up the CPU. The scheduler sets the suspend flag associated with the application, sets a timer to wake up the application at the start of its next period, and invokes the Linux scheduler to give the CPU to the next application with the next highest priority. When the timer expires at the start of the next period, the scheduler updates the deadline of the application, recalculates the priority of all applications based on the EDF policy, and invokes the Linux scheduler to let the application with the highest priority to proceed.

5. *ExitSrt*: This is invoked when the application is done (with all its frames). The scheduler removes the application from the SRT list, cleans up related data structures, and signals the

global coordinator to perform a global adaptation.

Accounting and Overrun Monitoring: At every timer expiration, the CPU scheduler invokes the CPU monitor to get the elapsed cycles since the last timer expiration and charges it to the application that is currently running. It also compares the cycles used by this application with the cycle budget allocated to the application. If it detects that the application has used up its entire budget, then the scheduler decreases the priority of the application and preempts it. If the preempted application does not finish the job by its deadline, then the scheduler replenishes the budget available to the application and allows it to finish. This extra budget given to the application is deducted from the application's new frame that will run during that period, if this is the first deadline miss in a sequence. If this is the second miss in a sequence, then the extra budget is compensated by asking the application to skip its next job. This is done by sending the miss status information via the FinishJob system call.

CDVFS: The CPU scheduler handles most of the CDVFS related implementation. When an application makes the BeginJob system call, in addition to refreshing its budget, the CPU scheduler calculates the CDVFS values based on the allocated time and frequency. It also invokes the CPU adaptor to set the CPU speed to the lower CDVFS frequency, and sets a timer to expire at the end of the low-frequency interval. When the timer expires, the CPU scheduler invokes the CPU monitor to get the resource usage, and the CPU adaptor to set the frequency to the higher CDVFS frequency.

Budget Sharing: When an application makes the FinishJob call, the CPU scheduler adds any unused budget to the *budget queue*. Later, when a timer expires because of a frame's overrun and the scheduler has to charge the frame for the elapsed cycles, it first checks whether it can charge any of the elapsed cycles to the budget queue. If it can, then the unused budget in the budget queue is adjusted accordingly, and a lower time is charged to the application. The scheduler also removes any expired budget from the budget queue.

4.4 Network Scheduling and Budget Sharing

The CPU scheduler in our system meets the added responsibility of a network bandwidth scheduler. This network scheduler also does network budget sharing, which works exatly like the CPU budget sharing. When an application has finished encoding a frame, and makes a FinishJob call, it also sends information regarding the number of bytes it has generated and its allocated bandwidth to the scheduler. The scheduler then checks whether the bandwidth requirement can be met by the allocated bandwidth and any available residual budget. If the requirement is not met, then the scheduler passes this information back to the application, as in the case of a CPU deadline miss.

4.5 Oracular Predictions

To determine the impact of the execution cycles and network bytes predictors in the per-application adaptation, we also implement the ability to use an oracular predictor in "real-time." At the end of a frame of an application, we "freeze" the real-time clock used by the CPU scheduler and run all the configurations of that application for the next frame. Using this, we can determine the actual execution cycles and byte counts to feed to the per-application predictor. We then restart the real-time clock and continue.

CHAPTER 5

EXPERIMENTAL METHODOLOGY

We run all of our experiments on the system described in Section 4. Except for the energy measurements and the actual network transmission, all parts of the system have been implemented.

Table 5.1 summarizes the input video streams we use and their characteristics. The streams are obtained from standard web sites hosting such sequences. They have been chosen to represent a spectrum in interframe computation variability and encoding complexity. All the streams process QCIF size frames.

In each experiment, we run two applications concurrently, each running a different stream. We use a total of four combinations of streams, shown in Table 5.2, to represent different combinations of variability/complexity. To study the effect of different types of resource constraints (i.e., system load), we use different periods (frame rates) for our workloads and different values of the available network bandwidth. We create four scenarios of resource constraints (depending on whether the CPU or network is constrained or not) as described below, studying four workloads under each scenario. Table 5.2 summarizes the workloads, their periods, and available bandwidth for each scenario. For simplicity, for a given scenario and workload, we use the same period for both applications, but for generality, the applications start with an arbitrary lag between them.

The four resource constraint (or system load) scenarios we study are:

Table 5.1 Video streams used and their characteristics.

Stream	**# Frames**	**Description**
salesman	450	Very low variability sequence of a talking head
foreman	450	Medium variability sequence of a talking head, has a scene change toward the end
buggy	450	High variability sequence of a buggy race, has a scene change in the middle
mobile	300	High complexity sequence of a toy train and tumbling blocks

Table 5.2 Scenarios evaluated.

Constraint	Number (Scenario-Workload)	Streams	Period (ms)	Bandwidth (Mbps)
CPU	1-1	salesman, mobile	30	11
	1-2	buggy, foreman	30	11
	1-3	foreman, mobile	30	11
	1-4	buggy, mobile	30	11
Network	2-1	salesman, mobile	35	2
	2-2	buggy, foreman	50	2
	2-3	foreman, mobile	50	2
	2-4	buggy, mobile	60	2
Both	3-1	salesman, mobile	30	5
	3-2	buggy, foreman	30	5
	3-3	foreman, mobile	30	5
	3-4	buggy, mobile	30	5
None	4-1	salesman, mobile	45	5
	4-2	buggy, foreman	45	5
	4-3	foreman, mobile	45	5
	4-4	buggy, mobile	45	5

Scenario 1, CPU constrained: We set the application period (frame rate) so that the application configurations that do the most computation (i.e., the most compression) are unable to run on our system (i.e., they would require a higher frequency than that supported). The network does not pose a constraint in this scenario – we set enough available bandwidth to send the bytes produced by the application configuration that does the least compression.

Scenario 2, network constrained: We set the application period and available network bandwidth so that the bandwidth requirement of the application configurations that perform the least compression exceeds the available bandwidth. The CPU does not pose a constraint in this scenario – the application period is set so that even the highest computation application configuration can complete in the available time.

Scenario 3, both CPU and network constrained: This is a combination of the above two constraints. In particular, we set the period and bandwidth such that the application configurations that perform the most or least compression are constrained.

Scenario 4, neither CPU nor network constrained: In this case, we pick the period and bandwidth such that none of the application configurations are resource-constrained.

In creating the above scenarios, we attempted to set the application frame rates as close to 30 frames/second as possible, while achieving the other goals of the constrained scenarios.

CHAPTER 6

OVERHEADS

We next report the overheads from various parts of our implementation. The overheads are reported in terms of the number of CPU cycles (which is virtually independent of frequency). For comparison, note that the number of CPU cycles for encoding a typical frame is of the order 10 to 25 million cycles.

Global Optimizer vs. Per-Application Adaptation: Figure 6.1 compares the cost for global optimization with that for per-application adaptation. For global, we measured the elapsed CPU cycles for the global optimization algorithm by Moser et al. (Section 3.1). To study how the optimizer scales with the number of applications, we report results for systems containing one to ten applications, where each application runs one of the four video streams used in this thesis. The system with ten applications may represent, for example, a teleconference system involving five sites (a video and an audio decoder for each of the four remote sites, and a video encoder and an audio encoder for the local site). Note that our numbers do not include any profiling cost incurred for making predictions for long-term resource usage for the global optimizer (discussed further below).

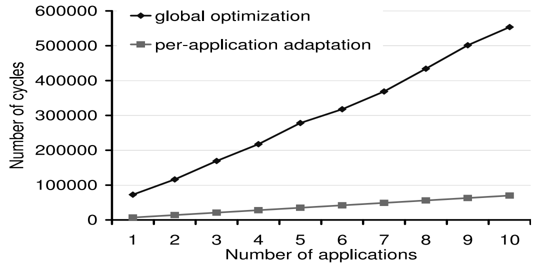


Figure 6.1 Overhead for global optimization and per-application adaptation.

To measure the cost of per-application adaptation, for each frame of the foreman sequence, we measured the elapsed cycles for the per-application adaptation algorithm (Section 3.2). We report the elapsed cycles averaged over the entire sequence. Note that this measurement includes the full cost of the adaptation, including the cost of predicting the resource usage for the next frame.

We find that the cost of per-application adaptation is significantly cheaper than that for the global optimizer (e.g., factor of 8 lower for ten tasks). In absolute terms, the global optimization cost with ten tasks is 5.5×10^5 cycles, which is 0.92 ms at 600 MHz. This is 3.7% of an average encoder frame computation time, and is thus a nonnegligible cost in terms of both time and energy. Per-application adaptation, on the other hand, takes 7.0×10^3 for each application, which corresponds to 0.117 ms for 10 tasks at 600 MHz, and is clearly feasible at the frequency of once every frame. We further discuss below why we expect the total overhead for global adaptation to be larger than reported here.

First, we note that the global algorithm used here is optimized for the system we study. Specifically, recall that we do not explore the full cross-product of the space of CPU and application configurations – we are able to assume a common frequency for all applications because of the special nature of the frequency-energy curve. However, this relationship is not true for other adaptations such as architecture adaptations that are becoming increasingly common in hardware [14]. Further, we also do not consider an adaptive network layer, which will further increase the complexity of the search space that needs to be considered by the global optimizer. As the number of possible adaptive layers, adaptive components within each layer, and the number of adaptive states within each component increases, the overhead of the global optimizer will continue to increase.

Finally, when considering the overhead of global adaptation, we must also consider overheads for the prediction of the long-term resource usage as required by the optimizer. In our system, we perform global adaptation when an application joins or leaves the system, which is a relatively rare event. Therefore, the profiling required for predictions can be done on-line (while running the system in sub-optimal configurations); the time spent profiling is a negligible fraction of the overall time that an application runs. However, for more frequent global adaptation, on-line profiling at sub-optimal configurations can be too expensive. We cannot directly use past history because we only have the history for the application configuration that was chosen for a frame; the optimizer

Table 6.1 Average number of cycles for new system calls.

EnterSrt	BeginJob	FinishJob	WaitNextPeriod	ExitSrt
2554	1477	1054	845	2623

needs to make predictions for all the configurations. We could potentially use the same predictors as used in the per-application adaptation to predict the behavior of the next frame, and keep track of the outputs of these predictors over several frames. Determining whether this is feasible requires a study of how well these predictors perform for a span of several frames. Our results show that per-application adaptation is much simpler, and gives significant benefits over streams of several hundred frames.

System Calls made by the Application: Table 6.1 lists the cycles used by each of the five system calls made by the application while running the foreman sequence. These values have been obtained by averaging the elapsed cycles for each call for each frame of the sequence. These system calls add up to less than 0.1% of the cycles used in processing a typical frame, and so represent negligible overhead.

Soft real-time scheduling and DVFS: The SRT scheduler requires less than 500 cycles per application. The high resolution timer it uses requires between 1000 to 1500 cycles for set up. So the scheduler overhead is also small.

For DVFS, the Pentium-M processor decouples the voltage and frequency transition, thereby allowing voltage to be changed while executing instructions. The DVFS overhead is around 10 μs [15], making intraframe frequency transition feasible.

CHAPTER 7

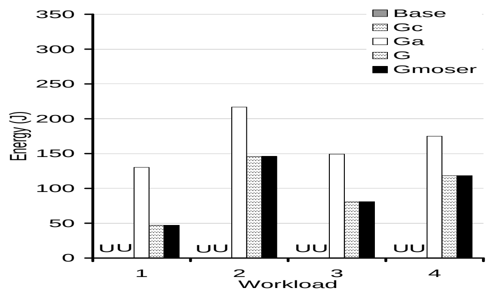
ENERGY SAVINGS

Sections 7.1 and 7.2 quantify the energy savings from global and per-application adaptation respectively. These sections assume CDVFS and budget sharing (only network budget sharing for global adaptation, and both CPU and network sharing for per-application adaptation). Sections 7.3 and 7.4 separately quantify the benefits of CDVFS and budget sharing, respectively. Since the primary benefit of budget sharing is in reducing missed deadlines, we discuss all deadline misses in Section 7.4.

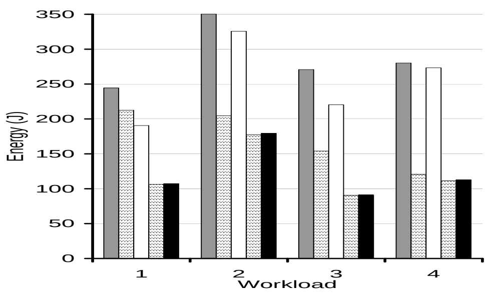
7.1 Benefits from Global Adaptation

Figure 7.1 presents the benefits of global adaptation. For each scenario (CPU/network constrained/unconstrained), for each workload, it gives the energy consumption of the base nonadaptive system and four global adaptation systems. In the CPU constrained cases, the base system is unable to meet the computation requirements of the base application configuration, indicated by "U." The first three global systems use the brute-force optimizer (Section 3.1) and so represent the best (but impractical) case for global adaptation. The three bars respectively represent a system with only CPU adaptation (with the base application configuration), a system with only application adaptation (with the base CPU configuration), and a system with both CPU and application adaptations, represented as Gc, Ga, and G, respectively. The last bar in the set uses the more practical optimizer by Moser et al. (Section 3.1) and includes both CPU and application adaptations, represented by GMoser.

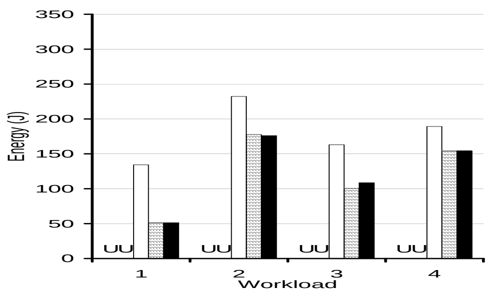
First, focusing on the results with the brute-force optimizer, we find that global adaptation provides significant energy savings over the nonadaptive system across all scenarios and workloads



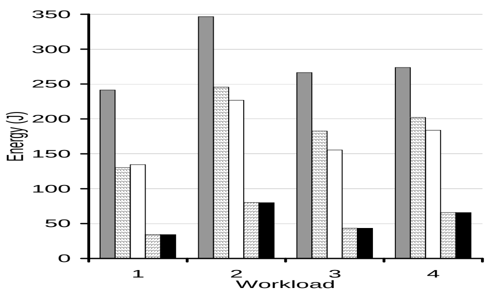
(a) CPU Constrained



(b) Network Constrained



(a) Both Constrained



(b) Unconstrained

Figure 7.1 Benefits of global adaptation.

studied. In the CPU constrained cases, the nonadaptive system is unable to even run the workload since it cannot meet the CPU constraint. In this case, G can change the application configuration to use less computation.

Overall, we see benefits from both CPU and application adaptation, with the best savings coming from the combination. On average, Gc saves 35% while Ga saves 25% over Base, for the cases where Base is able to run the workload. The combination of the two, G, saves an average of 70% energy over Base for these cases.

To see how global adaptation achieves its energy savings, Table 7.1 shows the application and CPU configuration chosen by G for all cases (the application configurations are roughly ordered by the amount of compression they perform – configuration 0 is the highest compression). We find that G chooses a wide variety of configurations depending on the video stream and the resource constraints.

Comparing the brute-force and the more practical optimizer, we find that the practical solver provides very similar energy benefits. Nevertheless, since our focus is on the benefits of per-

Table 7.1 Configurations chosen by global.

Constraint	CPU				Network			
Workload	1	2	3	4	1	2	3	4
CPU	768	1066	940	1058	946	943	812	807
App 1	15	15	15	15	3	2	2	2
App 2	14	15	14	14	2	1	2	1
Constraint	**Both**				**None**			
Workload	1	2	3	4	1	2	3	4
CPU	768	1131	1027	1165	600	711	627	706
App 1	15	10	13	3	15	15	15	15
App 2	14	11	12	12	14	15	14	14

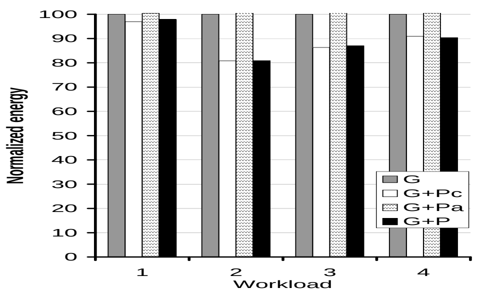
application adaptation, we henceforth use the brute-force optimizer to give global the best showing.

7.2 Benefits from Per-Application Adaptation

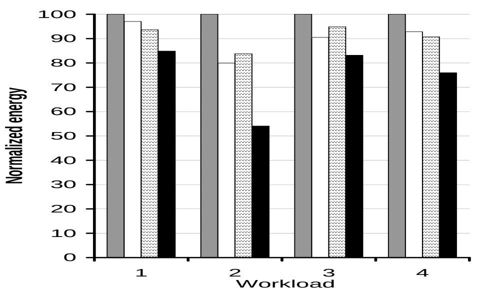
Figure 7.2 shows the energy consumed by various systems that combine per-application adaptation with global adaptation, normalized to the system G from Section 7.1 (i.e., only global adaptation, with adaptive CPU/application). For each scenario and workload, we show the energy of G, G enhanced with per-application adaptation for the CPU or G+Pc (i.e., the application configuration is fixed at that chosen by G); G enhanced with per-application adaptation for the application or G+Pa (i.e., the CPU configuration is fixed at that chosen by G); and G enhanced with per-application adaptation for both the CPU and the application or G+P.

Overall benefits of per-application adaptation: Overall, we find that G+P consumes less energy than G for all the scenarios and workloads. The magnitude and source of the benefits depends on the magnitude and nature of the resource constraints in the system, and the nature of the encoded streams. The benefits from G+P over G range from mostly negligible (in a few cases) to a significant 46% in the network-constrained case. The benefits are particularly significant when both streams have higher variability/complexity (workloads 2, 3, and 4). For these workloads and across all the constrained scenarios, the energy savings of G+P over G range from 10% to 46%, average 22%.

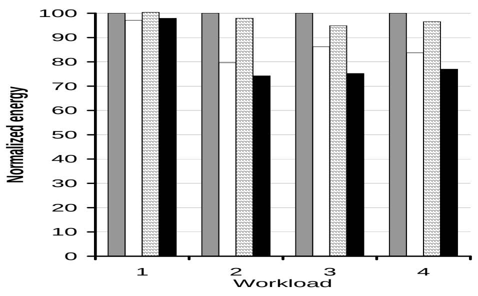
Benefits of per-application adaptation in the CPU: Per-application adaptation in the CPU provides significant benefits in all of the constrained scenarios where both streams in the workload have higher variability/complexity (workloads 2, 3, and 4). For these cases, the energy savings of G+Pc over G range from 7% to 20% (average of 14%). Even in the unconstrained case,



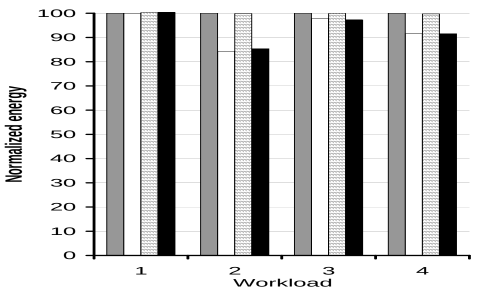
(a) CPU Constrained



(b) Network Constrained



(a) Both Constrained



(b) Unconstrained

Figure 7.2 Benefits of per-application adaptation.

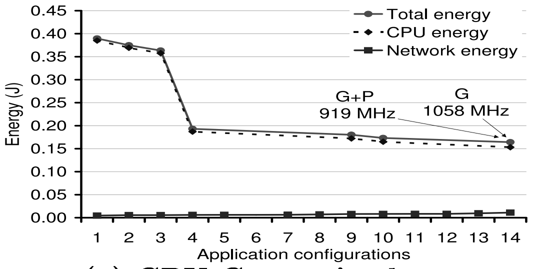
G+Pc provides a significant benefit for some workloads.

Benefits of per-application adaptation in the application: Per-application adaptation in the application provides discernible benefits only in the cases where there is a network constraint (scenarios 2 and 3).

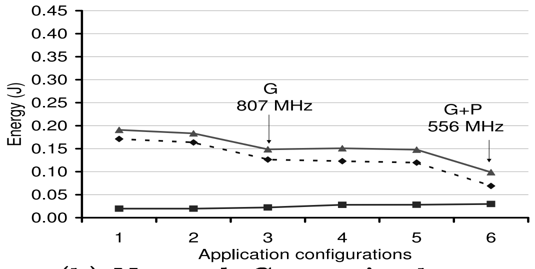
In scenario 2, which is primarily network constrained, G+Pa provides energy savings of 5% to 16% over G (average of 9%). Compared to G+Pc (i.e., a system that already has per-application CPU adaptation), the benefit of adding per-application application adaptation is quite significant – 8% to 32%, average 18% for G+P over G+Pc – in this scenario. It is noteworthy that the benefits of combining CPU and application adaptation are more than additive in some cases.

In scenario 3, which is both CPU and network constrained, the benefits of G+P over G+Pc are much lower, but not negligible – 7% to 13% – for workloads 2 to 4.

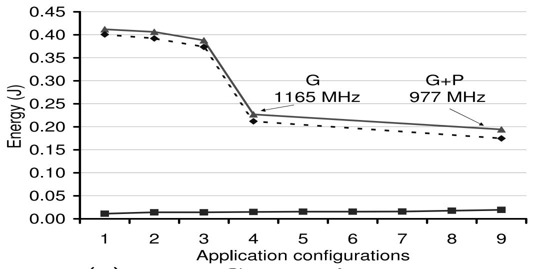
Analysis: Next, we analyze the reasons for the above results in each of the scenarios in more detail. Figure 7.3 shows the network, CPU, and total energy for each application configuration for a specific frame of workload 4 for each of the four scenarios (parts (a)–(d)). The application



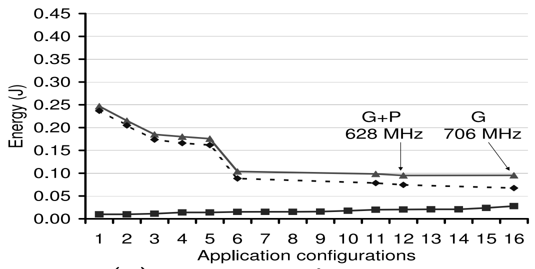
(a) CPU Constrained



(b) Network Constrained



(a) Both Constrained



(b) Unconstrained

Figure 7.3 Results analysis.

configurations are ordered in increasing order of bytes generated. Only those configurations where the CPU cycles for the configuration decrease with increasing bytes are shown – the remaining configurations are clearly suboptimal, since they neither reduce total energy nor meet any more constraints compared to the previous configuration. Configurations that do not meet the required constraints are also not shown. (For this reason, the same number on the x-axis may represent different actual configurations in the four graphs.) On each curve for total energy, we mark the application configuration chosen by the G and G+P systems, along with the frequency chosen.

Recall that network energy is simply the product of (bandwidth dependent) energy/byte and bytes generated. For CPU energy, we first need to determine the frequency at which the frame will complete the required cycles within the time allocated by the global adaptation. Network energy increases going from left to right due to increasing byte count, while CPU energy decreases; further, CPU energy initially falls faster than network energy rises due to the quadratic dependence on voltage. In general, therefore, we would expect that the total energy curve should initially fall along with CPU energy and then rise; further, this expected bowl-shaped curve would be a fairly shallow bowl because the effects of network and CPU energy change would cancel each other. In our graphs, for the most part, CPU energy is dominant, and so we find that the total energy curve primarily follows the CPU energy; i.e., we see the left drop and the shallow bottom of the bowl, but not the right rise. One slight exception is the unconstrained system, where the total energy curve has just started to rise towards the right, although by a negligible amount.

We can now analyze the four cases. It is easiest to start with the unconstrained case (part (d)). The G system chooses the rightmost application configuration based on its estimate of the 95th percentile cycle and byte count. The G+P system has a better estimate of these counts for the current frame, and does affect an application adaptation to a configuration to the left of G. However, because of the shallow nature of the bowl, the change in the application configuration by itself does not result in energy savings. CPU adaptation does save energy because G+P is better able to predict the cycle count and choose a lower frequency than G.

Next consider the network constrained case (part (b)). From an energy-minimization point of view, we would like to pick the rightmost configuration shown on this graph, and picked by G+P. However, G is not able to pick that configuration because its estimate of the byte count is too high (based on the 95th percentile) given the available bandwidth constraint. So G is forced to pick a less energy-efficient configuration that can meet the network constraint, enabling significant energy savings from G+P.

For the CPU-constrained case (part (a)), both G and G+P are able to pick the configuration with the least computation (rightmost), and so the most energy efficient. Thus, G+P does not see any benefit from application adaptation, compared to G. However, G+P does see benefit from CPU adaptation because of its ability to better predict the cycle count and use a lower frequency.

The case with both CPU and network constraints can be similarly analyzed.

Limitations of the Predictor: The energy savings from per-application adaptation are dependent on the prediction of the cycles and bytes for each configuration for the next frame. To isolate the impact of the current predictors, we performed experiments with an oracular predictor (Section 4.5). Our data showed that the current predictors in G+P are effective for energy savings – they lose on average only 4% (range 1% to 6%) of energy savings compared to an oracular predictor (detailed graphs omitted for space). The key drawback of the current predictors is in their ability to meet frame deadlines, which we discuss in Section 7.4.

Table 7.2 Deadline miss ratio with/without budget sharing.

	Budget Sharing				**No Budget Sharing**			
	App1		App2		App1		App2	
	G	G+P	G	G+P	G	G+P	G	G+P
CPU constrained								
1	0.3	0.7	0.3	0.0	0.0	0.7	0.3	0.0
2	0.0	1.3	0.0	3.8	0.0	**7.0**	0.0	3.3
3	1.3	1.3	0.0	1.0	0.3	4.0	0.0	0.3
4	0.0	0.3	0.0	0.0	0.3	3.7	0.0	0.0
Network constrained								
1	1.0	4.1	0.0	0.0	**8.3**	**9.0**	0.7	4.0
2	0.0	**6.8**	1.5	**8.5**	4.5	**22**	**5.4**	**24**
3	0.3	5.0	1.0	0.0	0.3	**24**	2.0	**5.7**
4	0.3	5.0	0.3	1.3	**10**	**20**	**20**	**27**
Both								
1	0.0	0.7	0.7	0.0	0.0	3.3	0.0	2.7
2	0.0	3.1	2.1	4.1	0.3	2.7	0.0	5.0
3	1.8	4.0	0.0	4.3	**6.0**	**30**	0.0	**29**
4	0.3	2.7	0.3	1.3	**6.0**	**30**	0.0	**29**
Neither								
1	0.0	0.3	0.3	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.8	0.0	1.5	3.5	3.3	4.8
3	0.0	0.0	0.0	0.0	0.7	2.3	4.3	0.3
4	0.0	0.0	0.0	0.0	1.3	**5.7**	4.3	0.7

7.3 Benefits from Emulating Continuous DVFS

All the results with CPU adaptation so far have included CDVFS as discussed in Section 2.1. Here we isolate the benefits of this adaptation. Overall, we find that CDVFS is quite beneficial for both the global and per-application adaptations in many cases. Across all scenarios and workloads, the average energy savings that G sees from CDVFS are 15% (maximum 33%), and that G+P sees from CDVFS are 13% (maximum 21%).

7.4 Benefits from Budget Sharing

The main benefit of budget sharing is in reducing the number of deadline misses; it has negligible ($< 1\%$) effect on energy. (Recall that we include frame drops in our deadline miss ratio as well, and include drops/misses due to CPU or network.) Table 7.2 shows the deadline misses for G and G+P for each workload and scenario studied, with and without budget sharing. The cases with miss ratios $> 5\%$ are highlighted in bold.

We first note that with budget sharing, we see acceptable deadline misses (within 5%) for all workloads and scenarios with only one exception. Workload 2 in the network-constrained scenario for G+P shows a miss ratio of 6.8% and 8.5% for its two applications. Without budget sharing, the deadline miss ratios are high (up to 30%) for several cases (covering all workloads) for both G and G+P. This shows that budget sharing is effective and critical for our system. However, budget sharing is not a foolproof technique (as illustrated by the above exception case), and improving the accuracy of the predictors for both the global and per-application adaptation case is an important avenue for future work.[1] It is also conceivable that some of the deadline misses could be avoided with buffering; however, the use of buffering has its limitations in that it introduces a delay which may not be acceptable for applications such as teleconferencing. The appropriate combination of buffering, improved predictors, and budget sharing remains a subject of our future work.

7.5 Summary

To summarize, our key findings are as follows:

Global adaptation by itself provides significant energy savings from both CPU and application adaptation.

Per-application adaptation provides significant energy savings over and above global adaptation. Overall, per-application CPU adaptation is effective in more scenarios, but per-application application adaptation shows high benefits when network bandwidth is at a premium.

Emulating continuous DVS shows significant energy benefits for both global and per-application adaptation.

Budget sharing is critical in avoiding deadline misses for both global and per-application adaptations. However, it is not foolproof, indicating the need for investigating better predictors.

[1]Note that simply increasing the percentile value of the selected frame for global prediction is not desirable since it will imply an even higher energy consumption for global.

CHAPTER 8

RELATED WORK

CPU/OS Energy Conservation: There is much work on CPU energy conservation through DVFS that saves energy by dynamically adjusting the CPU frequency and voltage based on application workload. In most prior work, the workload is either predicted heuristically based on the average CPU utilization [17, 18, 19] or derived from application worst-case CPU demands [20, 21, 22]. In contrast, GRACE-2 uses the runtime frame-based CPU usage information of multimedia applications to guide the DVFS adaptation, achieving the energy saving of DVFS while delivering soft real-time guarantees. Further, in contrast to most work that focuses on DVFS (e.g., [6]), GRACE-2 also considers adaptive applications, which makes it harder to predict the workload.

Our implementation of CDVFS emulation is based on the theorems presented in [7]. To the best of our knowledge, this is the first such implementation and evaluation of emulated CDVFS in a real system. Previous work such as [23] has shown the potential benefits of CDVFS at an algorithmic level.

Application Energy Conservation: Multimedia applications generally adapt output quality for CPU and energy usage. Cornel and Satyanarayanan [24] proposed three time scales of adaptation for wireless video applications. Flinn et al. [25] explored how to adapt applications for energy saving. Similarly, Mesarina and Turner [26] discussed how to reduce energy in MPEG decoding. Further, some researchers also propose OS or middleware support for application adaptation. For example, Odyssey [27] supports mobile application adaptation to trade off data fidelity and energy consumption. Puppeteer [28] and Agilos [29] are middlewares to help applications to adapt their QoS to variations of resource availability.

GRACE-2 differs from all of the above work, since it coordinates the adaptations of multiple applications and CPU frequency. Further, the application adaptations considered here also preserve application quality. More recently, Lara et al. [30], Efstratiou et al. [31], and Poellabauer et al. [32] proposed frameworks to coordinate adaptations of multiple applications. However, these three related works do not consider adaptation of system resources.

There is also a body of research that investigates the relationship between CPU energy for data processing (e.g., compression) and the energy consumption for transmitting the processed data. Barr and Asanovic [33] analyze the total system energy consumption for various lossless compression algorithms, including energy consumption from the network and CPU.

Cross-Layer Energy Adaptation: There has also been recent research in cross-layer adaptation for energy conservation. Ecosystem [34] attempts to balance system resources, including CPU and network, with the demands of multiple applications to allocate the necessary resources to each application in an energy-efficient manner. However, it adapts on a large scale, using all available information for each adaptation. The focus of our work is on hierarchical adaptation, showing the benefits of per-frame adaptation when applied in addition to global adaptation.

GRACE-2 is built on GRACE-1 [1] and our previous work on application adaptation [3]. GRACE-1 was our first prototype and introduced our notion of coordinated cross-layer adaptation. Its focus, therefore, was on studying the benefits of coordinated adaptation; our focus in GRACE-2 has been to study the benefits of *hierarchical* coordinated adaptation. GRACE-1 incorporated global adaptations in the CPU and application (which affected application quality). We additionally perform application and CPU adaptation at the frame granularity (per-application adaptation) and show it provides significant benefits over global adaptation alone. GRACE-1 also uses an internal scheduler adapation to reduce deadline misses. Our budget-sharing support has the same goals, but is more sophisticated as required in a scenario with per-application application adaptation. Finally, GRACE-1 did not consider any network constraints or network energy. GRACE-2 considers a network bandwidth constraint and considers network transmission energy.

Our work in [3] introduced the adaptive application studied here and studied its benefits in conjunction with an adaptive CPU in simulation and for only one application at a time at a per-application granularity. GRACE-2 takes this work much further since it coordinates adaptation of multiple applications and uses a combination of per-application adaptation and global coordination.

QoS- or Energy-Aware Resource Allocation: There is a large body of work dealing with QoS-aware resource allocation that is related as well. Q-RAM [35] models QoS management as a constraint optimization, which maximizes the overall system utility while guaranteeing the minimum resource to each application. Similarly, IRS [36] coordinates allocation and scheduling of multiple resources to admit as many applications as possible. These approaches, however, assume that the hardware layer is static, i.e., each system resource operates at a fixed mode. In contrast, GRACE-2 targets a mobile system with adaptive CPU and adaptive applications, and provides QoS support while saving energy. Rusu et al. [37] proposes two approximation algorithms for optimization that consider constraints of energy, deadline, and utility. Their algorithms share similarities with our global coordination algorithm, but they do not consider application adaptation.

CHAPTER 9

CONCLUSIONS

This thesis describes the GRACE-2 system, which implements a hierarchical cross-layer adaptation framework to reduce energy consumption in mobile devices. GRACE-2 implements an adaptive CPU, adaptive applications, and an adaptive soft real-time scheduler. To coordinate all of the adaptations, GRACE-2 uses a novel hierarchical approach where an expensive global adaptation occurs infrequently in response to large system changes and an inexpensive per-application adaptation occurs frequently (at every frame) in response to small changes. The two levels of adaptation are tightly coupled by requiring the per-application adaptation to respect the budgets allocated by the global adaptor. Besides a cross-layer adaptation, the GRACE-2 system also respects constraints and optimizes for energy across two layers – the CPU and the network.

This work is the first to comprehensively explore the benefits of per-application adaptation over global adaptation in a cross-layer adaptive system over a variety of scenarios (CPU-constrained, network-constrained, both CPU and network constrained, and a lightly loaded CPU and network). Our results show that per-application adaptation in GRACE-2 provides significant energy savings when added to global adaptation (up to 46%). While the magnitude and source of the benefits depends on the resource constraints in the system, we can expect a typical mobile device to operate under all of the conditions evaluated in this thesis. Further, given the low overhead of per-application adaptation and the relatively low added system implementation complexity over and above a system with global adaptation, the benefits achieved are clearly worthwhile to exploit.

There are several avenues of future work. We are currently working on incorporating an adaptive network layer into GRACE-2 that responds to variations in network bandwidth. We are also exploring joint energy optimizations in other components of the system. Finally, we are exploring integrating application adaptations that save energy further by changing the visual perception – this requires incorporating a notion of utility within the global optimizer and respecting that utility in the per-application adaptation.

REFERENCES

[1] W. Yuan et al., “Design and evaluation of cross-layer adaptation framework for mobile multimedia systems,” in *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, 2003, pp. 1-13.

[2] Sachs et al., “GRACE: A cross-layer adaptation framework for saving energy,” Sidebar in *IEEE Computer, Special Issue in Power Computing*, 2003, pp. 50-51.

[3] D. Sachs, D. L. Jones, and S. V. Adve, “Adaptive video coding to reduce energy on general-purpose processors,” in *Proc. of Intl. Conference on Image Processing*, 2003.

[4] M. Weiser et al., “Scheduling for reduced CPU energy,” in *Proc. of Symposium on Operating Systems Design and Implementation*, 1994, pp. 13-23.

[5] C. J. Hughes et al., “Variability in the execution of multimedia applications and implications for architecture,” in *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*, 2001, pp. 254-265.

[6] W. Yuan and K. Nahrstedt, “Energy-efficient soft real-time CPU scheduling for mobile multimedia systems,” in *Proc. of Symposium on Operating Systems Principle*, 2003, pp. 149-163.

[7] T. Ishihara and H. Yasuura, “Voltage scheduling problem for dynamically variable voltage processors,” in *Proc of Intl. Symp. on Low Power Electronics and Design*, 1998, pp. 197-202.

[8] Cisco, Cisco aironet 350 series client adapters datasheet, 2004, http://www.cisco.com/en/US/products/hw/wireless/ps4555/.

[9] M. Caccamo, G. Buttazzo, and L. Sha, “Capacity sharing for overrun control,” in *Proc. of Real-Time Systems Symposium*, 2000, pp. 295-304.

[10] M. Moser, D. P. Jokanovic, and N. Shiratori, “An algorithm for the multidimensional multiple-choice knapsack problem,” in *IEICE Transactions on Fundamentals of Electronics*, 1997, pp. 582-589.

[11] S. Krantz, S. Kress, and R. Kress, *Jensen's Inequality.* Boston, MA: Birkhauser, 1999.

[12] H. H. Chu and K. Nahrstedt, “CPU service classes for multimedia applications,” in *Proc. of IEEE Int. Conf. on Multimedia Computing and Systems*, 1999, pp. 296-301.

[13] W. Yuan, “GRACE-OS: An energy-efficient mobile multimedia operating system,” Ph.D dissertation, University of Illinois at Urbana-Champaign, 2004.

[14] C. J. Hughes, J. Srinivasan, and S. V. Adve, "Saving energy with architectural and frequency adaptations for multimedia applications," in *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*, 2001, pp. 250-261.

[15] Intel, Intel Pentium-M processor datasheet, 2003, http://www.intel.com/design/mobile/datashts/25261203.pdf.

[16] G. Anzinger, "High res posix timers," 2001, http://high-res-timers.sourceforge.net.

[17] D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld, "Policies for dynamic clock scheduling," in *Proc. of Symposium on Operating Systems Design and Implementation*, 2000, pp. 73-86.

[18] T. Pering, T.Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc of Intl. Symp. on Low Power Electronics and Design*, 1998, pp. 76-81.

[19] K. Flautner and T. Mudge, "Vertigo: Automatic performance-setting for linux," in *Proc. of Symposium on Operating Systems Design and Implementation*, 2002, pp. 105-116.

[20] H. Aydin, R. Melhem, D. Mosse, and P. Alvarez, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *Proc. of Real-Time Systems Symposium*, 2001, pp. 95-105.

[21] T. Pering, T.Burd, and R. Brodersen, "Voltage scheduling in the lpARM microprocessor system," in *Proc of Intl. Symp. on Low Power Electronics and Design*, 2000, pp. 96-101.

[22] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. of Symposium on Operating Systems Principle*, 2001, pp. 89-102.

[23] W.-C. Kwon and T. Kim, "Optimal voltage allocation techniques for dynamically variable voltage processors," in *Proceedings of the 40th Conference on Design Automation*, 2003, pp. 125-130.

[24] M. Corner, B. Noble, and K. Wasserman, "Fugue: time scales of adaptation in mobile video," in *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, 2001, pp. 75-87.

[25] J. Flinn and M. Satyanarayanan, "PowerScope: A tool for proling the energy usage of mobile applications," in *Proc. of 2nd IEEE Workshop on Mobile Computing Systems and Applications*, 1999, pp. 2-10.

[26] M. Mesarina and Y. Turner, "Reduced energy decoding of MPEG streams," in *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, 2002, pp. 202-213.

[27] B. Noble et al., "Agile application-aware adaptation for mobility," in *Proc. of Symposium on Operating Systems Principles*, 1997, pp. 276-287.

[28] J. Flinn et al., "Reducing the energy usage of office applications," in *Proc. of Middleware 2001*, 2001, pp. 252-272.

[29] B. Li and K. Nahrstedt, "A control-based middleware framework for quality of service adaptations," *IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms*, vol. 17, num. 9, pp. 1632-1650, 1999.

[30] E. de Lara, D. Wallach, and W. Zwaenepoel, “HATS: hierarchical adaptive transmission scheduling for multi-application adaptation,” in *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, 2002, pp. 100-114.

[31] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst, “A platform supporting coordinated adaptation in mobile systems,” in *Proc. of 4th IEEE Workshop on Mobile Computing Systems and Applications*, 2003, pp. 128-137.

[32] C. Poellabauer, H. Abbasi, and K. Schwan, “Cooperative run-time management of adaptive applications and distributed resources,” in *Proc. of 10th ACM Multimedia Conference*, 2002, pp. 402-411.

[33] K. Barr and K. Asanovic, “Energy aware lossless data compression,” in *Proc of International Conference on Mobile Systems, Applications, and Services*, 2003, pp. 231-244.

[34] H. Zeng et al., “Ecosystem: Managing energy as a first class operating system resource,” in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 123-132.

[35] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, “A scalable solution to the multi-resource QOS problem,” in *Proc. of Real-Time Systems Symposium*, 1999, pp. 315-326.

[36] K. Gopalan and T. Chiueh, “Multi-resource allocation and scheduling for periodic soft real-time applications,” in *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, 2002, pp. 34-45.

[37] C. Rusu, R. Melhem, and D. Mosse, “Maximizing the system value while satisfying time and energy constraints,” in *Proc. of Real-Time Systems Symposium*, 2002, pp. 246-257.