# A NEW FRAMEWORK FOR HIERARCHICAL CROSS-LAYER ADAPTATION


BY

DANIEL GROBE SACHS

B.S., University of Illinois at Urbana-Champaign, 1998
M.S., University of Illinois at Urbana-Champaign, 2000


DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

# ABSTRACT

Battery technology has not kept the pace with innovations in microprocessors and wireless networks, and as a result saving energy is increasingly important on modern wireless-equipped laptops. To save energy, one technique we turn to is reconfiguration, or *adaptation*, of system components to permit more efficient operation.

Our adaptive GRACE system provides for coordinated adaptation across all system layers and applications through the use of an adaptation hierarchy. Resources are broadly allocated to applications by a global adaptation layer, which considers all applications and system layers but is relatively expensive and can therefore run only infrequently. These allocations are refined and converted into adaptation decisions on a job-by-job basis using a cross-layer "per-application" adaptor that considers only the current application's demands and allocations. Finally, each system layer optimizes itself independently within the application's allocation to minimize its energy consumption.

To validate this approach, a simulation of the GRACE system was constructed around a custom-built adaptive video encoder that provides the system the ability to reduce its CPU utilization at the cost of increased network-bandwidth requirements. These simulations show that the GRACE architecture can save more than 50% of the total CPU and network energy by appropriately shifting demand from the CPU to the network.

The GRACE system also allows utility to be allocated between applications. Conventionally, this is done using suboptimal heuristics. However, Lagrangian techniques can also be applied to the allocation problem, supplying a method whereby optimal allocations (to within convex-hull approximations) can be made without requiring exact foreknowledge of upcoming workloads or searches of the cross-product of present and future applications. Through the use of Lagrangian optimization, unbounded improvements in total utility can be achieved compared to the constant-workload heuristic; simulations demonstrate a factor of two improvement in total utility in advantageous circumstances.

*To my grandparents, who all loved learning*

# ACKNOWLEDGMENTS

I am heavily indebted to the assistance of my advisor, Douglas L. Jones, as well as the coauthors on the manuscripts from which this dissertation has been created: Sarita V. Adve, Albert F. Harris, Christopher J. Hughes, Won Jeon, Robin Kravets, Klara Nahrstedt, Vibhore Vardhan, and Wanghong Yuan. I would also like to thank Judith Grobe Sachs for her assistance editing portions of this dissertation, and Kannan Ramchandran and Naresh R. Shanbhag for their help in defining and refining my research goals.

This work was supported in part by the National Science Foundation under Grant No. CCR-0205638.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Until recently, "multimedia" was a wired-only phenomenon. Even "wireless" solutions, such as television, had generally been hooked into external sources of power. Traditional wireless environments have also been receive-only, with transmit capability being restricted to a small segment of the voice communication market (cordless and cellular telephones) and hobbyists (through amateur radio).

However, the confluence of several phenomena—including third generation cellular networks, 802.11, and rapidly improving computer and display technology—have recently permitted the creation of true wireless multimedia devices. In addition to receiving media, modern laptops can capture, encode, and wirelessly transmit multimedia streams.

Battery technology has not kept up with the advances in computing and communications, and therefore energy has become a significant limitation in the operation of these types of portable multimedia devices. Also, as computers have become faster, computation has become a larger part of the energy budget of mobile devices. For this reason, we can no longer do energy-aware encoding without taking into account the energy used in processing as well.

This dissertation explores some of the problems encountered in creating a system that optimizes end-to-end energy consumption by jointly adapting behavior across the various layers of the system.

## 1.1 The GRACE Project

This work has been done as part of the broader Illinois GRACE (Global Resource Adaptation through CoopEration) project [1], which aims to develop adaptive systems in which all layers—hardware, operating system, network, and applications—coordinate to optimize multimedia quality of service and resource usage. We believe that the correct approach is to allow all system layers, from the application software to the CPU and networking hardware, to adapt in response to changing conditions. Furthermore, this adaptation should be fine-grained; that is, we should be able to take advantage of short-term variation in the behavior of the system and its environment. This requires that we recompute optimal configurations for the entire system frequently; ideally, we do this independently for every work unit (frame) of each application.

However, this philosophy introduces a challenge—how do we achieve this fine-grained cross-layer adaptation between all applications without the large overhead of looking across all applications and recomputing complete configurations every frame? This problem is further complicated by the fact that this allocation problem is equivalent to the NP-hard knapsack problem [2], preventing the exact solution of the allocation problem even on a sporadic basis.

We believe that the challenge of fine-grained cross-layer adaptation can be met using a hierarchical approach: resource allocations can be made at a global level, but exact decisions about how these resources are to be used can be deferred to lower levels that are more aware of instantaneous details. The results of the global adaptation are monitored, and if the resource requirements of an application vary significantly from the allocation, or if the application exceeds or falls short of the expected utility by a significant amount, we again trigger a global optimization.

## 1.2 Related Work

While significant work has been done on the general concept of energy-aware optimization of multimedia applications, the work described in this thesis differs in several important ways. Our architecture enables decoupled adaptation; we provide a minimal interface between various system layers that permits end-to-end optimization, but allows the layers to make their own decisions about when and what to adapt. This contrasts previous architectures, which tend to have closely coupled system layers that require detailed information about the inner workings of all components of the system, or operate the adaptation across broad time scales and are unable to take advantage of short-term variations in conditions.

### 1.2.1 Adaptive applications

Adaptive applications have been proposed in many contexts. For mobile computing, the adaptations have focused on changes in available bandwidth [3–5] and energy [6,7]. A significant amount of work has also been done offloading parts of the application's computation to wired (and therefore not energy-limited) servers, such as [8] and [9]. The work described in [9] is particularly interesting because it may enable offloading of work from the wireless device to a fixed transcoder without incurring a significant rate penalty.

Also relevant is work explicitly considering the energy consumed encoding the video, such as that found in [10] and [11]. This idea of trading off computation complexity for encoding efficiency forms an important part of the adaptation framework introduced in Chapter 3, and an encoder similar in nature to the complexity-adaptive decoder described in [10] and [11] is used for this work. However, even this prior work considers other parts of the system as a "black box" and cannot account for adaptation occurring in other parts of the system.

While most work in complexity-adaptive encoders, including our own, concentrates on motion-compensated DCT video encoding, it is important to mention that other approaches exist. For instance, the recent introduction of rate-scalable or "progressive" image and video encoders [12–15] allows adaptation through the generation of bitstreams that can be *truncated* without excessive loss of fidelity; the more data that are successfully transmitted, the better the reconstruction quality. These wavelet-based encoders also have the desirable "embedding" property, meaning that any low-rate encoded bitstream is a *prefix* of a more detailed, higher-rate bitstream; they therefore do not require choosing the eventual quality of the encoded stream before encoding begins.

Progressive encoding and decoding offers two advantages over traditional encoders: first, if a transmission error occurs, a coarse version of the transmission can be reconstructed using only data transmitted before the error, and second, if a computation overrun occurs, encoding and decoding can be terminated before completion, generating a lower-fidelity representation of the encoded stream.

### 1.2.2 Other single-layer adaptations

In addition to adaptive applications, large bodies of literature address adaptive or adaptable system layers. One of the most common single-layer adaptations is the the adaptation of CPU and architecture; see [16] for a summary of past work in this field. Adaptive and application-specific networks are also common [17–26]. Finally, as power management has been introduced to CPUs and operating systems, schedulers have become increasingly power-aware and in some cases explicitly optimize the system's energy use [27–29].

### 1.2.3 Joint network-application adaptation

There has been considerable exploration of joint adaptation between the application and network layers, primarily done by the signal processing community under the name of "joint source-channel coding." This work primarily focuses on jointly optimizing specific applications (e.g., video or image source coding) and specific network functions (e.g., error/channel coding or congestion control). This is typically done by feeding the network state back to the application, and allowing the application to drive the network's adaptation. In most cases the coding for both source (application) and channel (network) are intermingled and inseparable [30–41]. In some cases, the network directly drives the application [42]. In other cases [43–49], and my own master's thesis work [50, 51], the separation between layers is preserved through the use of embedded image and video coders and passing only the rate-distortion curve of the encoded sequence. And although some of this prior work explicitly considers processing energy [38, 44], most has concentrated on network bandwidth and its associated energy costs, largely ignoring computational complexity and processing energy and time.

Another approach to the joint source-channel adaptation problem is the use of *middleware*, or system-level monitors that estimate resource availability and coordinate the allocation of resources across applications and nodes. This type of approach is frequently applied to the bandwidth management problem, and a survey of this type of work along with several examples can be found in [52]. For example, Barghavan's TIMELY system [53] combines network and application adaptation with a revenue model used to ensure good quality of service and minimize quality-of-service variations. The "Bandwidth Management" system of Shah et al. [54] similarly aims to provide fair quality of service allocations across different nodes acting in an 802.11-style wireless network. Although system utility is implicitly or explicitly optimized by these systems—often on a distributed basis, expanding on the single-node optimization done by the GRACE system—the optimization strategies are dedicated to doing management of bandwidth and not energy. As a result, the optimization is somewhat at odds with energy minimization, which is often achieved at the cost of increases in bandwidth use.

Finally, the networking community has studied interactive networks, designed to provide applications the information they need to adapt themselves without exposing the details of modulation and protocols to the application. This type of approach allows the status of the network to drive the adaptation, without the need for the application or its author to understand the operational details of the network protocols. This can be accomplished by notifying the application when network state changes [55] and allowing the application to make its own reconfiguration decisions, or by allowing the application to describe the payoff from different quality of service levels the network can provide [42]. In either case, separation between layers is preserved; one layer describes its behavior to the other, which makes an adaptation decision based on the information provided.

### 1.2.4 Cross-layer adaptation frameworks

Because my work is fundamentally a coordination framework for cross-layer adaptation, it shares some similarity with many other such frameworks. These frameworks typically coordinate a subset of system layers, such as resource management and applications [56–58] or adaptive CPU and OS resource allocation [59–61].

For example, Q-RAM [56] and IRS [62], much like our global adaptor, coordinate resource allocation across applications to maximize quality within all resource constraints. Several researchers have also implemented a more complete global adaptation layer similar to our GRACE-1 [2] framework. Examples include ECOSystem [59, 61] (part of the Duke University Milly Watt project), which uses a model called "currentcy" to manage energy, with the goal of achieving a specified battery lifetime, and the coordination algorithms proposed by Rusu et al. [60], which consider constraints on energy, deadline, and utility in the presence of hardware voltage scaling. Also, Efstratiou's "coordinated adaptation platform" [63], Georgia Tech's "Q-Fabric" [58], and de Lara's "EACS" [57] frameworks, like our GRACE-1 framework, combine application adaptations with OS resource management.

## 1.3 GRACE Project Publications

The GRACE system described in this dissertation builds primarily on the work done by members of the GRACE group. My first GRACE publication [64] introduced the adaptive application and showed how cross-layer adaptation of the CPU [65] and application could be exploited to optimize the system's total energy consumption. Likewise, other group members developed the adaptive scheduler [29] and the network protocol and estimation components [26]. These components have been integrated and tested together with the adaptive application and algorithm from [64] in [66].

As this work is being performed as a collaborative effort with several other faculty and students, many sections of this dissertation are derived from joint publications with several other authors. Parts of this introduction chapter are based on a publication in the 2002 SHAMAN workshop [1] and collaborative work creating the initial grant proposal for the project. Also, a description of the system described in Chapters 3 and 4 was published as a technical report [67]. This work also appeared in abbreviated form as a sidebar in a special issue of IEEE Computer magazine [68].

# CHAPTER 2

# ADAPTIVE APPLICATION

A basic component of all the adaptive systems I introduce in this thesis is our adaptive encoder. This encoder is designed to enable power savings by providing the ability to trade off not only between resource consumption and the quality of the encoded stream, but also between different types of resources. The adaptive encoder presented here is an expanded version of the one used for my initial work in application/CPU cross-layer adaptation [64].

Specifically, our adaptive encoder builds on a standard H.263 encoder by providing the option to eliminate various parts of the encoding process.

## 2.1 Adaptive Encoder Core

The encoder used in this work is based on the TMN (Test Model Near-Term) 1.7 encoder [69], which encodes standards-compliant H.263 streams. We modify the encoder to trade off computational complexity against the number of bits output by providing mechanisms to vary the compression efficiency of the encoder.

Because the above TMN encoder uses a full search to find motion vectors, we replaced the motion search with a fast search. The search implementation is similar to the logarithmic motion-search technique [70], but simplifies its implementation (at the cost of some efficiency) by searching all four diagonals as well as up, down, left, and right in one step rather than searching only two of the diagonals in a separate step after the cardinal directions are checked.

After the implementation of this fast motion-search algorithm, the encoder was profiled to identify the largest consumers of CPU time. Even with the fast motion-search algorithm, the motion-search function remained the largest single consumer of CPU time. The next largest consumers of CPU time were the DCT and IDCT functions. As a result, the motion search and DCT functions were targeted for adaptation.

Also, because the adaptive features require that certain information be passed along to the decoder for correct decoding, the modified encoder and decoder are no longer H.263 compliant. As a result, conventional decoders cannot decode streams generated by the adaptive encoder. A compatible decoder has therefore been implemented (based on the corresponding TMN H.263 decoder) and is included in the encoder library.

Detailed descriptions of the functions in the adaptive video-encoder library have been included as Appendix A.

### 2.1.1 Adaptive capabilities

This encoder provides several adaptive capabilities. First, as in all video encoders, the quantizer step size can be adjusted, affecting the tradeoff between bit rate and the quality of the stream. However, unlike conventional video encoders, our adaptive encoder also provides the capability of adjusting tradeoffs between computational complexity and bit rate.

This is done in two ways. First, the encoder provides the capability to prematurely abandon motion search when a "good enough" match is found. Second, for blocks with a low sum-of-absolute-differences, the encoder can skip the DCT computation and quantize the pixels without the DCT coding. Last, the encoder can generate I-frames with some or all of the macroblocks sent completely uncoded.

All of these parameters can be adjusted on a frame-by-frame basis by an adaptation controller. The algorithms for selecting a configuration are part of the GRACE system, and will be described in future chapters.

#### 2.1.1.1 Motion search threshold

The logarithmic motion search we use reduces the complexity of the motion search significantly, but it still takes approximately 25% of the total CPU time. To cut this down further, we allow the motion-search function to "escape" if, at any point in its search, it finds that the sum of absolute differences for the current motion vector is less than or equal to an adjustable threshold. When this "escape" occurs, the selected motion vector will be used as-is and not further refined.

The motion-search threshold can be set to arbitrary numbers from zero up. A value of zero ensures that the full motion search is run. Small numbers will cause the system to skip the motion search if little change has occurred or if a good match is found; large values will disable the motion search entirely.

#### 2.1.1.2 DCT threshold

After the motion search is completed, each macroblock is split into 6 8x8 DCT blocks (four luma blocks and two chroma blocks). Normally, these blocks would then be transformed using a DCT, then quantized and coded. However, the DCT and IDCT together were the largest user of the CPU in the encoding process, adding up to 30% of the total CPU demand. To reduce their impact, we have added the ability to skip the DCT and quantize and send the untransformed coefficients instead.

Before the DCT is performed, the absolute value of all of the elements of the 8x8 DCT block is summed. If the sum exceeds the specified threshold, the DCT is performed; if not, the DCT is skipped and the untransformed input block is copied as the DCT output.

To successfully decode a stream in which not all blocks have been DCT transformed, we must deviate from the H.263 specification by adding an extra bit before each 8x8 block of coefficients is transmitted. This bit indicates whether or not a DCT was performed on that block. When the decoder receives the DCT block, it checks the bit and if the bit is not set, no inverse DCT is performed.

Any DCT blocks associated with a macroblock that encoded in INTRA mode or that are part of an I-frame are always DCT-transformed. However, the flag bit is still sent to simplify decoding.

#### 2.1.1.3 I-frames and uncoded macroblocks

Last, like all encoders, our encoder has the capability of generating I-frames that do not depend on any prior frames. However, our encoder extends on this capability by allowing some or all of the macroblocks in an I-frame to be transmitted entirely uncoded.[1] If a macroblock is sent uncoded, all processing (including the motion search, DCT, quantization, and variable length coding) is skipped, and the raw YUV data from the incoming frame is copied straight to

[1]In fact, the encoder permits uncoded macroblocks to be included in P frames as well. However, we have found this mode to be suboptimal.

the output, along with a marker (implemented as an extension to the macroblock mode table) that indicates that the macroblock was transmitted using the "uncoded" mode.

This feature is controlled by a setting that indicates the probability that a given block will be transmitted uncoded. Normally, this is set to zero (all blocks are encoded). If it is set to a nonzero value, a random number in the range $[0, 1)$ is generated for each macroblock that is to be encoded. The macroblock is encoded if and only if the random number is greater than or equal to the uncoded-block probability parameter. Each block is therefore selected as coded or uncoded independently, and with equal probability.

An extension to the H.263 macroblock mode tables was required to indicate the presence of an uncoded macroblock to the decoder. Also, for efficiency, the uncoded data is byte-aligned in the output stream.

### 2.1.2 Additional capabilities

In addition to the adaptive functionality, the encoder also has explicit functions to save and restore the encoder state. These are used for error recovery; rather than sending an I-frame if a frame is lost, the controlling application has the option of backing up the encoder state to a previous frame that arrived intact at the receiver. This is done through a pair of functions that save and restore the encoder state. It is the application's responsibility to select the previous frame to use, and to ensure that the decoder state matches the encoder state when the frame being decoded was encoded.

### 2.1.3 Omissions and unsupported features

Several features of the H.263 standard are not supported in the adaptive encoder, although they were supported by the original H.263 encoder. These include arithmetic encoding, use of fused PB frames, and automatic rate control.[2] Half-pel motion compensation has been disabled in this version of the encoder, although it can be re-enabled with a compile-time flag.

Like the original TMN H.263 decoder, our decoder has limited ability to recover from stream errors and should not be presented with corrupted frames. No error concealment is performed. It is assumed that any corrupted frame will be dropped by the controlling application.

[2] Arithmetic encoding and PB frames may still work if the adaptive properties of the encoder are disabled, but this is untested.

**Table 2.1** Encoder configurations

| # | Frame Type | DCT Thresh | Motion Thresh | # | Frame Type | Uncoded MBs |
|---|---|---|---|---|---|---|
| 0 | P | 0 | 0 | 8 | I | none |
| 1 | P | 0 | 750 | 9 | I | 1/7 |
| 2 | P | 500 | 750 | 10 | I | 2/7 |
| 3 | P | 500 | 1250 | 11 | I | 3/7 |
| 4 | P | 500 | 2000 | 12 | I | 4/7 |
| 5 | P | 500 | 3000 | 13 | I | 5/7 |
| 6 | P | 1000 | 5000 | 14 | I | 6/7 |
| 7 | P | 20000 | 20000 | 15 | I | all |

## 2.2 Configuration Space

The motion-search threshold, DCT threshold, and block-drop probability are all continuous parameters. However, our adaptation algorithm is designed to work with a set of configurations that is discrete and well-ordered. Therefore, it is necessary to choose a set of discrete operating points that the adaptive algorithms can select.

We have chosen a total of 16 configurations. These configurations were selected for well-ordered and well-spaced cycle counts. However, due to variations between different streams, there was one test case in which the cycle count was nonmonotonic.

The configurations are numbered in order of increasing output size and decreasing computational complexity. The first configuration, numbered zero, sets both the DCT and motion-search threshold to zero. This ensures that all possible compression is done. The next configuration increases the motion-search threshold, resulting in a significant decrease in the average CPU cycle consumption with an accompanying small increase in bandwidth. The next 6 configurations, up to configuration number 7, increase both the DCT and motion-search threshold.

Configuration 8 requests a fully encoded I-frame. Configurations 9 through 15 replace a random subset of macroblocks with completely uncoded copies of the original frame. Configuration 9 sends, on average, 1/7th of the macroblocks uncoded, configuration 10 2/7ths, and so forth to 15, which sends the entire frame uncoded. Each macroblock is chosen to be sent coded or uncoded independently with the chosen probability.

Table 2.1 gives a list of the motion-search threshold, DCT threshold, frame type, and the percentage of uncoded macroblocks used for each of the 16 selected configurations.

### 2.2.1 Configuration selection

The application includes a file ("configs.h") that contains the configurations that have been selected for use. It declares a function `set_config` taking an integer parameter that indicates which configuration (0-15) to use, and calling the appropriate set of encoder functions to set the encoder configuration. It also declares a function `config_is_i()`, which returns a nonzero value if the encoder has been set to encode I frames. This is used to properly set dependency information.

The file also declares several "constants." `NUM_OPTIONS` is set to the total number of configurations that have been declared. `I_ABOVE_OPTION` is set to the minimum configuration number representing an I-frame; all higher numbers represent I-frames as well. `QUANT_ONLY` is set to the lowest configuration number which does not send uncoded macroblocks. `MAX_CONFIG` is set to the highest valid configuration number. In the current implementation, these "constants" are actually implemented as variables, which change when the quantizer is reset. This allows the encoder to lock out inappropriate configurations.

## 2.3 Encoder Performance

This selection of configurations gives us a wide variety of different operating points that have varying tradeoff between CPU and network utilization while keeping the stream quality roughly constant. In this section, we will show these tradeoffs explicitly.

These results are based on several standard MPEG-4 test sequences at the CIF (352 x 288) resolution and using a relatively high quality ($Q = 6$). We evaluated each of the test sequences at all of the 16 byte/cycle tradeoff configurations listed, and at two different values for the quantization step size.

We specifically consider the encoding of three standard MPEG test sequences of 300 frames each, which represent the variety of different types of sequences we will encode. The first, "Akiyo," is a talking-head sequence. The second, "Mobile," is a sequence with panning and several moving objects. The third, "Foreman," has a talking head at the beginning and then pans over to a moving view of a construction site.

We also evaluate a combined sequence of 5500 frames, consisting of a large number of different test sequences concatenated together. It is intended that this represent the average

**Table 2.2** Comparison of original H.263 encoder and adaptive encoder. Encoder operating in its maximum compression mode (Q = 6, CIF).

| Stream | Encoder | Bytes | Time |
|---|---|---|---|
| Combined | TMN H.263 | 21098480 | 1138.42 s |
| | adaptive | 28105113 | 283.31 s |
| Akiyo | TMN H.263 | 176914 | 41.02 s |
| | adaptive | 231280 | 13.00 s |
| Mobile | TMN H.263 | 4181474 | 57.24 s |
| | adaptive | 5464199 | 18.14 s |
| Foreman | TMN H.263 | 1023868 | 57.21 s |
| | adaptive | 1680845 | 15.65 s |

performance of the encoder across a large corpus of varying types of video streams. The composite sequence was used to create the encoder-performance tables used by the adaptation algorithm described in future chapters.

### 2.3.1 Compared to the unmodified TMN H.263 encoder

Due to the simplified motion search, elimination of half-pel motion compensation, and the extra bits required to add the DCT and uncoded-macroblock flags, the adaptive encoder does not compress as effectively as the unmodified H.263 encoder even if full compression is enabled. The modified encoder, in its maximum-compression mode, uses about 33% more bits for the same quality. However, CPU load even when full compression is used is approximately one-fourth the CPU load of the original TMN H.263 encoder.

Because our work is aimed at laptops, which tend to have relatively high CPU power demands and relatively low network power demands, we do not support the original full search or half-pel motion compensation as options in our adaptive encoder. Enabling the full motion search would also result in frame rates of approximately 5 fps (frames per second) on our test platform. However, in the future when faster microprocessors are available or on platforms with higher network costs, these functions could be enabled.

Table 2.2 compares the performance of the modified encoder against the original H.263 encoder with advanced options disabled.

### 2.3.2 Comparison of adaptive modes

With the encoder quantizer set to 6 to achieve a high-quality video stream, we get a range of approximately an order of magnitude in the CPU load between the sending the stream completely uncoded and executing the full compression task. The range in the number of bytes generated varies across approximately two orders of magnitude.

#### 2.3.2.1 Byte and cycle comparison

Figure 2.1 shows the the number of cycles and number of bytes required to encode an average frame of each of the four test sequences for all 16 application configurations. We see that the bytes and cycle counts are nearly monotonic with configuration numbers; the only exception is a slight rise in the number of CPU cycles going from Configuration 7 to Configuration 8 on the “Akiyo” curve with Q = 6. Furthermore, we see that the number of bytes and cycles both vary by approximately a factor of two before we start sending uncoded macroblocks; the remaining configurations have the expected linear change in both cycles and bytes as the number of uncoded macroblocks increases.

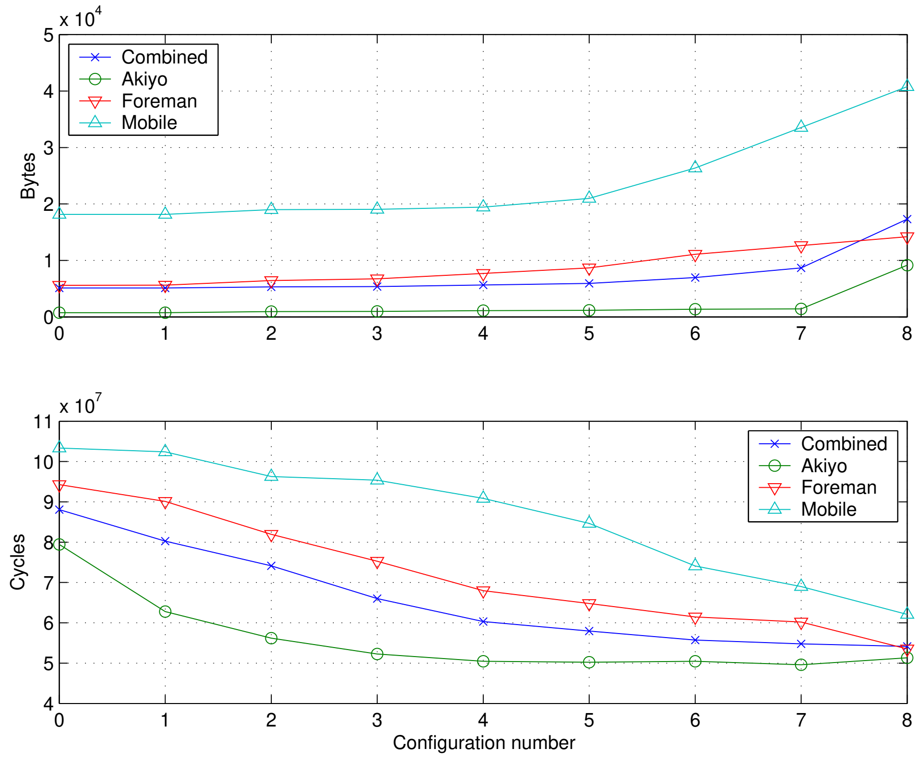
#### 2.3.2.2 PSNR comparison

To verify that our adaptations preserve the good quality of the video stream, Figure 2.2 shows the PSNR for the decoded stream as the encoder configuration is varied. From this graph, we see that for a quantizer step size of 6, configurations 0-7 (representing P-frames) all stay within a range of approximately 1 dB from the the “full compression” mode.

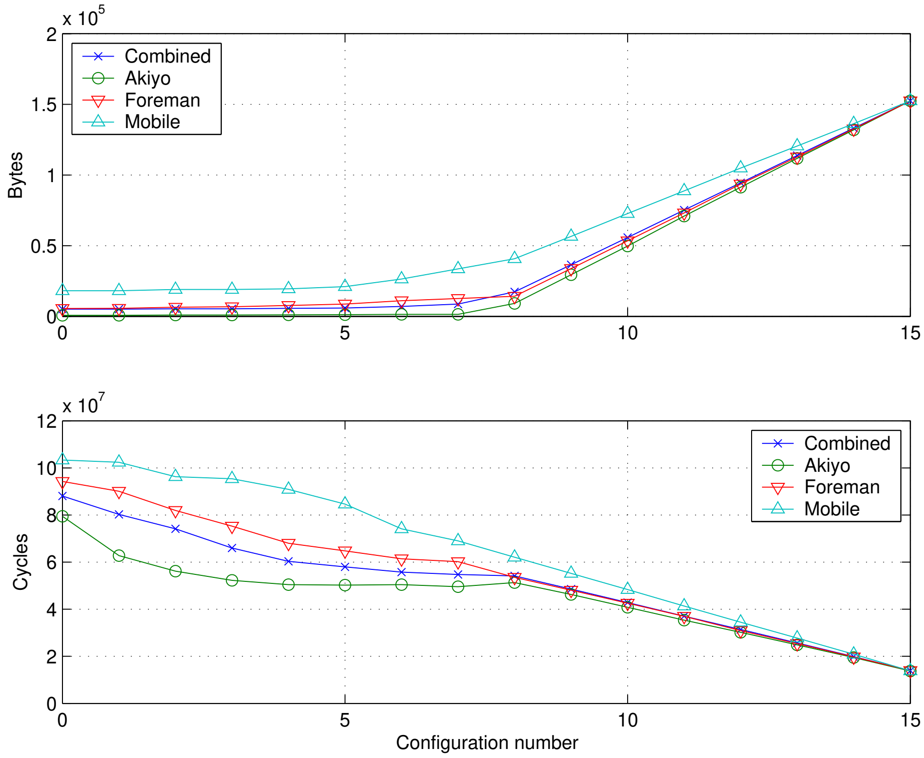
### 2.3.3 Perceptual quality comparison

Perceptually, configuration 6 and 7 result in slight quality loss at Q = 6; although detail is preserved better than in configuration zero, some color artifacting occurs.

For configurations 8 and above (I-frames and uncoded macroblocks), unsurprisingly we see the PSNR improve dramatically. For configuration 15, which is entirely uncoded and therefore has perfect fidelity, it is undefined. Examples of encoded images for configuration 0, configuration 7, and configuration 15—the baseline encoder, the encoder with DCT and motion search disabled, and the raw frame data, respectively—are shown in Figure 2.3.




(a) Configurations 0 through 8




(b) Configurations 0 through 15 (including uncoded macroblocks)

**Figure 2.1** Bytes and cycles for high-quality streams (Q = 6).

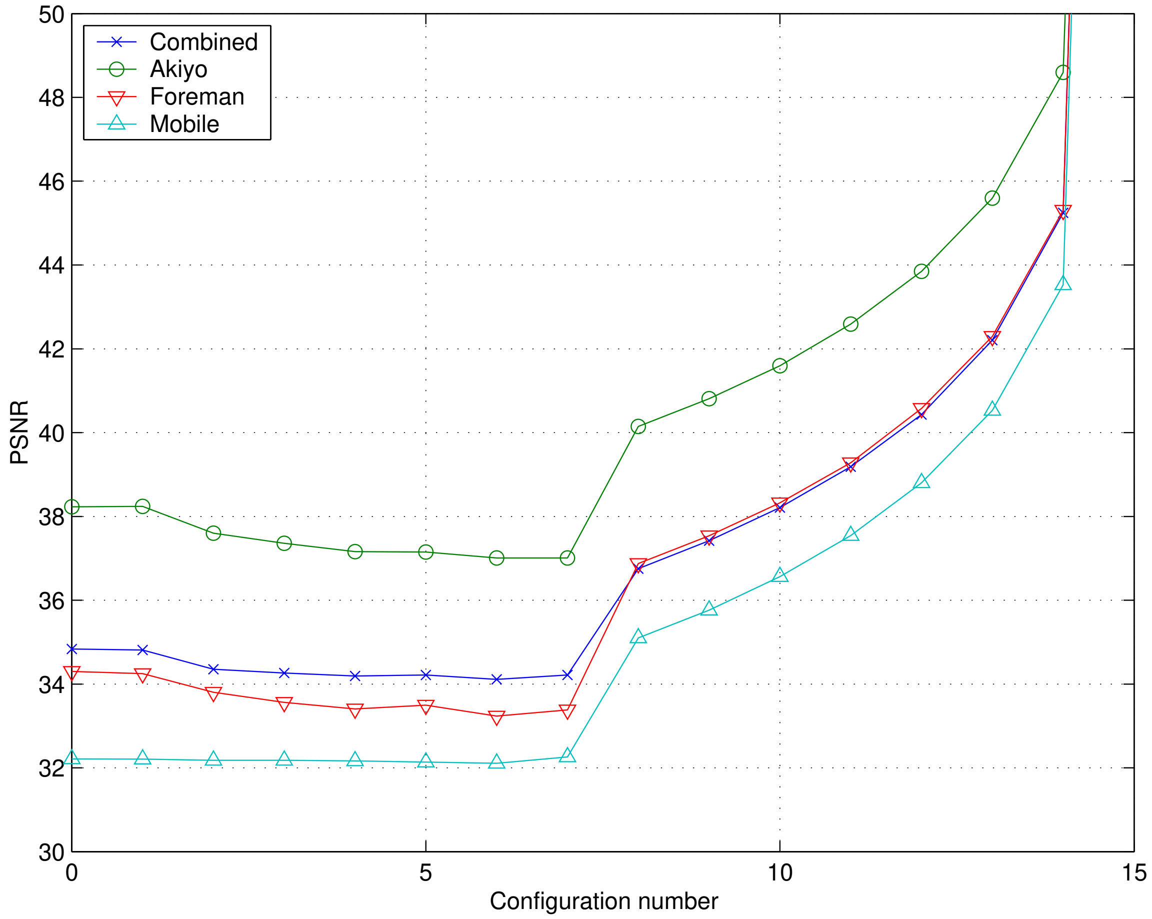



**Figure 2.2** PSNR for high-quality streams (Q = 6).
All configurations are active.


**Figure 2.3** Sample high-quality frame.
Left to right: configuration 0, configuration 7, configuration 15 (uncoded).

### 2.3.4 Reduced quality operation

We also evaluated the encoder at quantizer step size Q = 12, which provides a reduced quality stream at a significantly lower bit rate. Comparison of the bit rate and cycle counts for various configurations are shown in Figure 2.4.

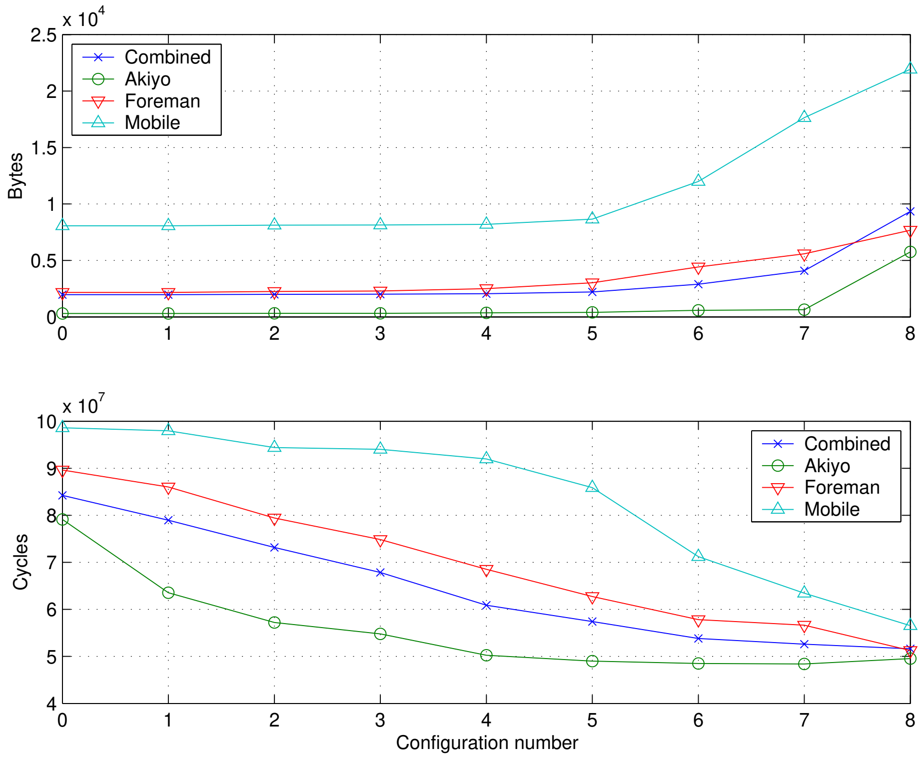
However, as the quantizer step size is increased, the effects on the images of eliminating the DCT becomes more visible. This is due to the fact that there are fewer terms large enough to be quantized to nonzero values, which increases the error disproportionally. Also, the error is less concentrated at high spatial frequencies and therefore can be more easily noticed. As a result, using the configurations that eliminate many DCTs with Q = 12 results in images that are noticeably degraded. In addition to the 2 dB drop in image PSNR for configurations 6 and 7 shown in Figure 2.5, there is also significant blockiness and color shifting visible in configurations 6 and 7. The blockiness in these configurations has a distracting harsh quality, as shown in Figure 2.6.

Due to the distracting quality loss when configurations 6 and 7 are used with Q = 12, we restrict the encoder to using modes 0-5 and 8 when operating in its reduced quality mode. By avoiding configurations 6 and 7 and configurations that substitute uncoded macroblocks for coded macroblocks, we avoid the distracting artifacts and ensure quality remains comparable across all of the available configurations. The uncoded-macroblock configurations are also disabled, as the restricted quantizer will only be used in network-constrained situations.

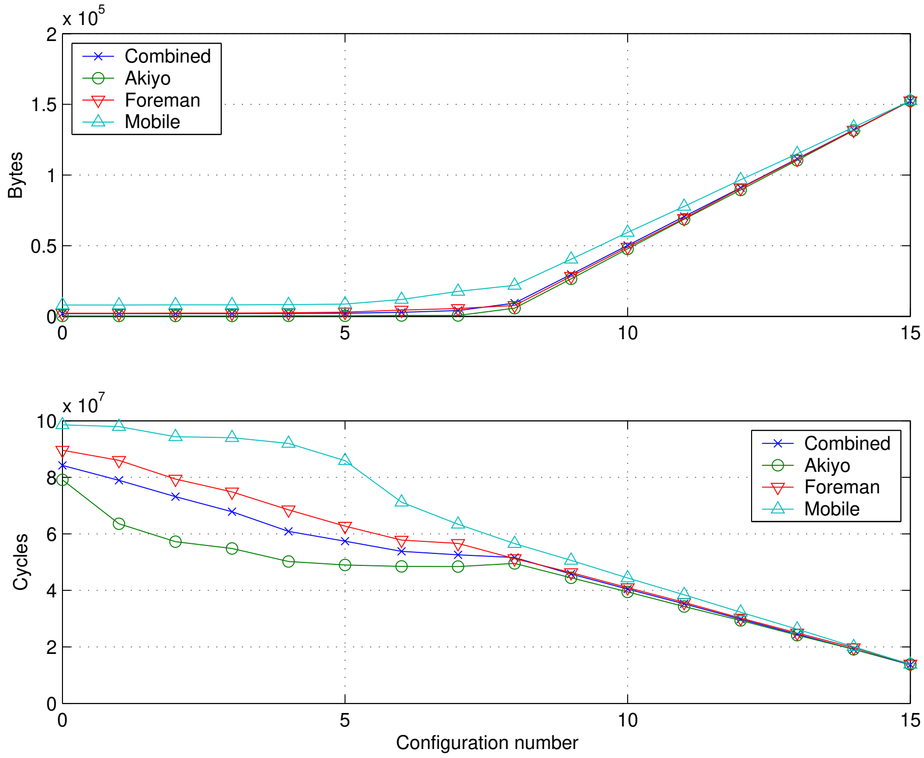
## 2.4 Conclusion

Although the encoder library presented in this chapter is neither standards-compliant nor state-of-the-art, it provides the ability to trade off between CPU and network utilization, and gives a large range of operating points ranging from full compression to no compression at all. In future chapters, we will use this capability to enable energy savings by moving resource utilization between CPU and networking.

The same techniques presented in this chapter can be applied to modern video coders, such as MPEG-4 and H.264, although it may not be possible to make all of the associated modifications while retaining compatibility with these standards. Applying these techniques to a more modern video coder could provide a more efficient video encoding where the same




(a) Configurations 0-8




(b) Configurations 0-15 (including uncoded macroblocks)

**Figure 2.4** Bytes and cycles for low-quality streams (Q = 12).

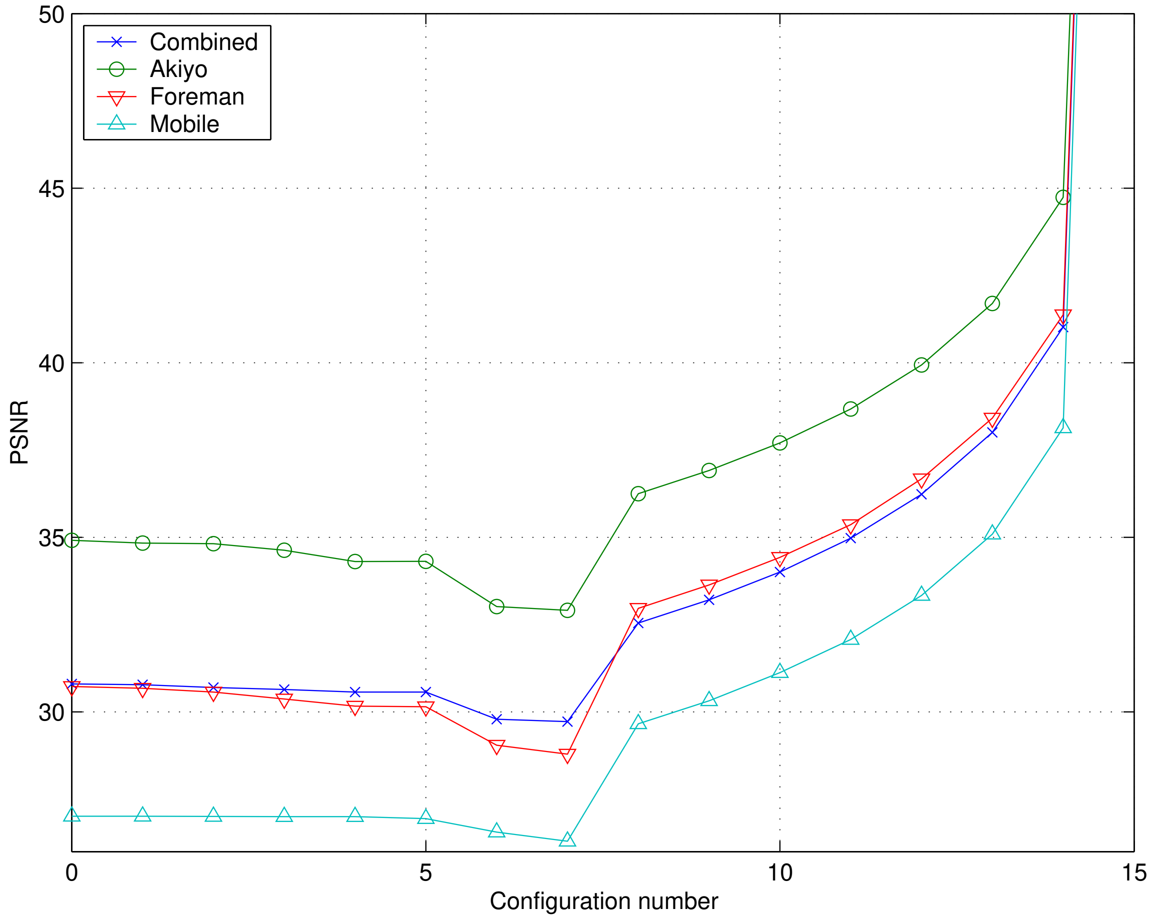



**Figure 2.5** PSNR for low-quality streams (Q = 12).
Configurations 6 and 7 have noticeably reduced quality and are therefore disabled.


**Figure 2.6** Sample low-quality frame.
Left to right: configuration 0, configuration 5, configuration 7 (disabled).

number of cycles achieves better compression, and the maximum achievable compression ratio is higher. Furthermore, by porting a modern video encoder to use the same interface presented here and in Appendix A, it could be dropped into the broader GRACE framework with no other changes to the adaptive encoder application.

# CHAPTER 3

# THE GRACE SYSTEM FRAMEWORK

## 3.1 Introduction

In the previous chapters, we have introduced the concepts of coordinated adaptation between system layers, and the hierarchy of adaptations — from global adaptations that affect the entire system to per-layer internal adaptations that affect only a single layer and application.

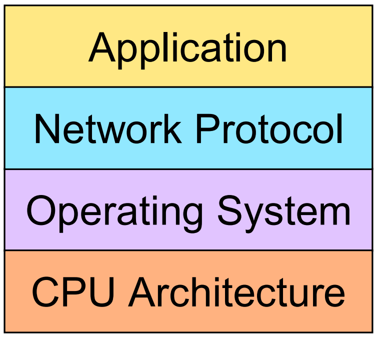
The UIUC Global Resource Adaptation through CoopEration (GRACE) project aims to unify adaptation across system layers and applications through the use of coordinated, cooperative adaptation. In this chapter, I present our vision of the GRACE system.

### 3.1.1 Overview

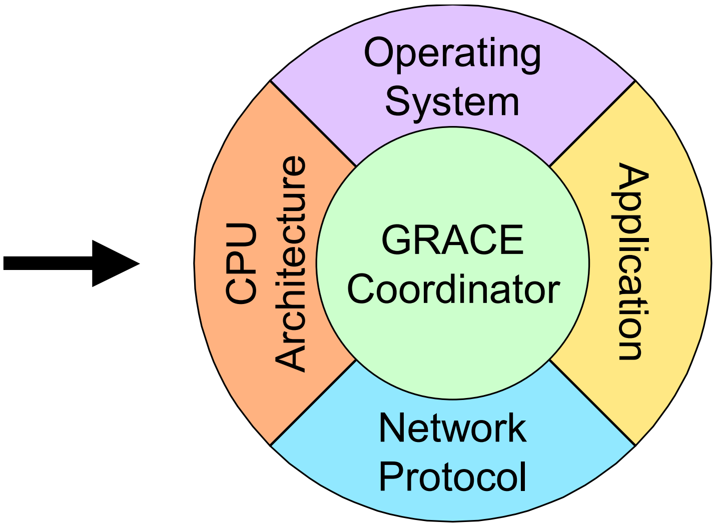
The GRACE vision (Figure 3.1) differs from most adaptation systems in that it coordinates the adaptation of all system layers, rather than allowing each system layer to adapt independently and possibly at cross-purposes.

The GRACE system incorporates several unique features to limit complexity while permitting fully cross-layer adaptations. The most important of these is the use of an adaptation hierarchy, which permits us to combine the benefits of large scope and frequent adaptations without incurring large computational overheads.

Ideally, a cross-layer adaptation system would constantly reallocate resources across applications to achieve the maximum possible benefit from the adaptation. However, as the full cross-layer adaptation involves solving an NP-hard optimization problem, the computational load associated with frequent reallocation is prohibitive. For this reason, we split up the adaptation into a hierarchy, in which relatively simple adaptations are done on a continuous basis and the more complicated allocation problems are solved only when necessary.




(a) Current systems




(b) GRACE system

**Figure 3.1** GRACE vision.

Second, we have designed a set of interfaces that minimize the information transferred between system layers while preserving the ability of the system to optimally allocate resources and configurations to each application and system layer. These interfaces abstract the data used by various system layers to contain only information meaningful to that layer. For instance, the application estimates its demands in terms of transmitted bytes and cycles, and passes this information on to the CPU and network layers, which translate the information into an appropriate selection of operating modes and return to the application answers about feasibility and energy requirements. These interfaces permit the various system layers to be unaware of the internal details of adaptation in other layers, rendering it unnecessary for each layer to contain a detailed model of the workings of the other layers.

The adaptivity provided by the GRACE system can be used two ways. First, the performance of applications running on the system can be traded against energy consumption to find the best balance of instantaneous utility and runtime. Second, the behavior of various system components can be tweaked to trade off between utilization of different system components, such as the CPU and the network interface. Although changing the performance of the applications running on the system is an intrusive change (that could potentially bother the user if done too frequently), trading off between different configurations that offer the user the same quality of service can be done at any time.

### 3.1.2 Terminology

We use "utility" to refer to abstract measures of user satisfaction. In practice, for video this is likely to mean a metric like PSNR (pseudo signal-to-noise ratio), possibly scaled by a factor representing the importance of a particular multimedia application to the user.

"Resources" are used to refer to a measure of the ability of the system to serve applications. An application consumes resources when it sends or receives data over the network, or performs computation; also, a system layer can consume resources on behalf of applications. For example, if the network layer does error-correction processing on application data, this consumes processor time that must be charged against the resource allocation for the application.

## 3.2 System Energy Model

Our system consists of two major adaptive components: a CPU that can change its operating frequency and voltage in response to changes in processor demand, and a network that constantly adjusts transmit power and data rate to minimize power consumption given current network conditions.

### 3.2.1 CPU

Current microprocessors are capable of dynamically adjusting the voltage requirements, $V$, and the operating frequency, $f$. Dynamic voltage and frequency scaling (DVS) takes advantage of these capabilities to alter $V$ and $f$ in response to application demands [71].

The power required to drive the processor is determined by three components, the voltage $V$, the frequency $f$, and the effective capacitance of the microprocessor $C_{\mathit{eff}}$. The value of microprocessor frequency and voltage scaling comes from the fact that as $f$ is increased, the required voltage $V$ also increases at the same time—ideally, in a linear relationship. As a result, reducing the operating frequency of the microprocessor and then reducing the voltage to an appropriate level for that frequency provides a superlinear (ideally, quadratic) reduction in the amount of energy consumed per cycle. Architectural adaptations can affect $C_{\mathit{eff}}$, but are not considered here.

The total energy consumed by the CPU ($E_{CPU}$) to complete a job is therefore determined by the power consumed by the CPU, the frequency, and the number of cycles, $I_c$, the job takes. Prior work [71] has shown that the number of cycles $I_c$ is does not vary significantly as the microprocessor speed is scaled for multimedia applications that we consider. The equation we use to compute total CPU energy consumption for a job is therefore as follows:

$$E_{CPU} \propto C_{eff} \times V^2 \times I_c \tag{3.1}$$

### 3.2.2 Network

The power used to drive a wireless network card is composed of the power required to drive the transmitter, $P_t$, the power to keep the card in transmit mode, $P_{xmit}$, and the power required to operate the rest of the circuitry of the card, $P_{base}$. There is also an energy cost associated with protocol processing for each packet, $E_{proc}$. Therefore, for a given packet of size $D$ at a transmission rate of $R_b$, the total per-packet energy consumption, $E_{net}$, is defined as

$$E_{net} \equiv \left[\frac{D}{R_b} \times (P_t + P_{xmit} + P_{base})\right] + E_{proc}. \tag{3.2}$$

where $E_{proc}$ is roughly constant relative to packet size but depends on the cost of the CPU, and $P_{base}$ and $P_{xmit}$ are fixed by the interface device. In the context of this energy model, energy conservation techniques aim to affect one or more of the other components. For example, for any given channel characteristics, transmit power control minimizes $P_t$, while bandpass modulation adaptation affects $R_b$. Additionally, reliability mechanisms such as retransmission and forward error correction (FEC) increase the amount of data sent, essentially increasing $D$. Finally, idle-time power management techniques allow $P_{base}$ to be conserved during idle periods in communication.

### 3.2.3 Application energy model

Our application energy model consists of two parts: the energy used by the CPU, and the energy used by the network. Obviously, a real system will have other components draining energy from the battery as well. However, the power consumption of these components - the camera, display, chipset, RAM, and other associated parts - are more or less fixed for this type of application. (We expect that for the bulk of the video conferencing session, we will be able to turn the hard disk off.) We therefore lump the energy consumption of all these system components into a fixed "external energy" cost that is not involved in the adaptation process.

## 3.3 The GRACE Framework

The GRACE framework is based on the concept of *hierarchical* adaptation, in which we do expensive adaptations occasionally, and limited-scope but inexpensive adaptations constantly. The combination of the different layers of adaptation allow us to achieve most of the benefits of continuous, global adaptations without incurring the overhead of running full cross-layer adaptations on a frame-by-frame basis.

The hierarchical adaptation of the GRACE system consists of three levels: "global" adaptations, which allocate invariant resources to the competing applications of the system; "per-application" adaptations, which are low-overhead fine-grained cross-layer adaptations that can be done on a job-by-job basis; and "per-layer" local adaptations, in which a single system layer chooses the best configuration for itself in ways that are not directly visible to other layers or applications.

Although we presently do not consider distributed systems, the hierarchical framework we describe can be extended to handle them. This would involve another, higher layer of adaptation ("cross-node" or "network-global") that would distribute tasks across nodes, calling upon the existing global allocation framework to find the best configurations of tasks within a node and using similar optimization algorithms to find the set of tasks running on each node that optimizes overall user satisfaction.

### 3.3.1 Long-term/global adaptation

"Global" (per-node) adaptation works across applications and layers, and as shown in Figure 3.2, works across applications and across layers. It is responsible for optimally allocating resources to each of the applications running in the system, balancing quality of service for each application against the power consumption and hence the runtime of the system. Because this optimization must, in general, consider all configurations of all system layers and applications running in the system, it operates only when necessary due to a systemwide change in applications or resources, and therefore at time scales of many seconds or more.

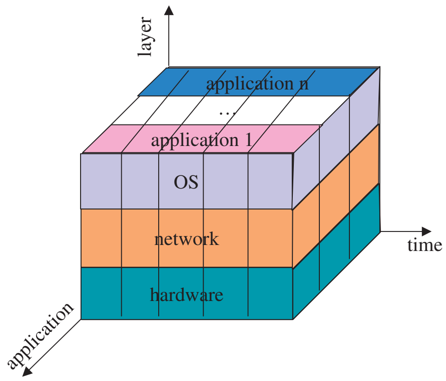



**Figure 3.2** Global adaptation.

Data on "average" jobs is used to choose the set of application configurations which form an optimal tradeoff between utility and energy for the "average" frame. Once such a set of application configurations is found, these configurations are converted to an allocation, splitting the total available variable resources across applications proportionally to how these "average" frames use the resource.

A key to our global resource allocation is that the resources are allocated before changes due to reconfiguration are introduced. To do this, we use "CPU time" and "Network time" as our invariant resources. Instead of counting cycles or instructions, both of which can vary as we reconfigure the hardware, we allocate a certain amount of wall-clock CPU time per frame (job) to each task (application). No matter how the hardware is reconfigured, the application is not permitted to exceed its wall-clock time allocation. Likewise, as long as the network round-trip time is insignificant compared to the time taken for the network to handle a single application frame (job),[1] we can similarly allocate "time on the network" or "raw bandwidth" to each task. This allocation will be used to transmit both its application data and any required error-control coding.

The global optimization process is managed by the global coordinator, which communicates with other system layers as shown in Figure 3.3 to determine the resource demands of different applications and the current state of the system and network. The global coordination process starts out with the global optimizer querying all of the application predictors for lists of available configurations. The application predictors, along with the CPU and network estimators also used in the adaptation process, combine precomputed, static profiles with runtime data to

[1]Without this assumption, it may be desirable to interleave packets of different applications.

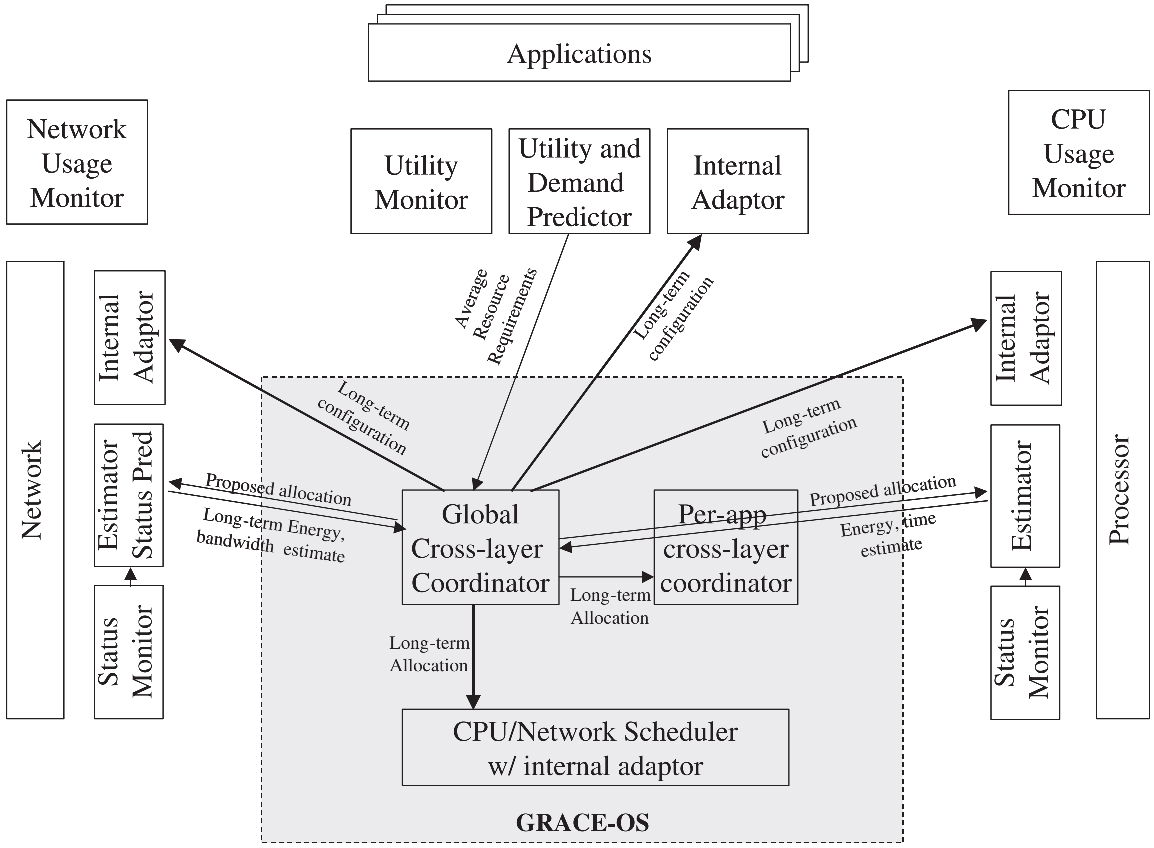



**Figure 3.3** Global adaptation communications.

estimate the current resource demands of the application. The application predictors then report back a set of available configuration profiles, each indicating the expected utility, CPU use, and network requirement for the associated application configuration.

The global optimizer then uses this data to repeatedly propose a set of operating points to the hardware and network predictors. These configurations are then checked for feasibility, utility, and energy consumption, and the best configuration for each application is selected.

Once a feasible set of configurations and utilities with acceptably low total energy consumption is found, the global coordinator assigns the application its configuration and resource allocation. The assigned configuration is stated in terms of the utility and period of the application; this allows the lower adaptation levels to select an appropriate application configuration within a range that the user finds indistinguishable. Because the resource allocations cannot depend on the state of the system, CPU allocations are expressed as the amount of CPU time available to the application per frame or period, and the network allocation is expressed as the proportion of "raw bandwidth," or time on the network, allocated to the application. This "raw" bandwidth allocation includes the time spent on retransmissions, and forward error correction, and does not vary even if the data rate is changed.

After the final utility and resource allocations are determined, the global coordinator reports to the per-application coordinator the allocations it selected. The final resource allocation is also reported to the application, CPU, and network adaptors and their corresponding schedulers. Reporting the resource allocations to the internal adaptors during the global adaptation process allows them to prepare for the demands of the configurations selected by the global coordinator. This is especially useful to the application, which may need to prepare user-interface elements (such as window size) for the selected utility.

### 3.3.2 Short-term, per-application adaptation

"Per-application" adaptation works across layers on a job-by-job basis, as shown in Figure 3.4.

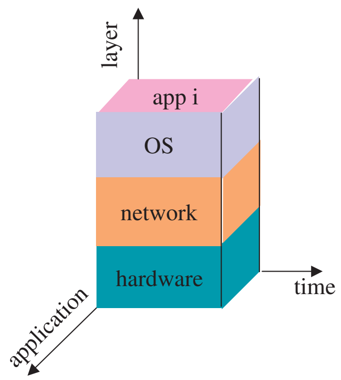



**Figure 3.4** Per-application adaptation.

Before the per-application adaptation can run, resources must be allocated to each application by the global adaptor. Along with CPU and network allocations, the global adaptor also allocates a desired quality of service to each application. This allocation dictates the quality level that the per-application adaptor must use if possible. But the per-application adaptor is free to choose any application, CPU, and network configuration that achieves the requested quality of service. This freedom allows the per-application adaptor to select the set of configurations for all layers which minimizes the energy consumed by a single job, while respecting the allocations of CPU time and network bandwidth.

We choose to restrict the per-application adaptor to energy minimization due to the short-term nature of its adaptations. Allowing the per-application adaptor to change the delivered utility (i.e., the stream's quality of service) could potentially introduce rapid, annoying fluctuations in the quality of the multimedia stream.

If external conditions do not permit the per-application adaptor to choose a configuration that meets the requested quality, it is expected to provide the maximum quality possible while respecting the original allocation. To simplify recovery from situations in which the resource availability is too low or varies dramatically from one frame to the next, the per-application adaptor is permitted to allow jobs to temporarily overrun their CPU or network allocation. However, if such an overrun occurs, the per-application adaptor is required to reduce the allocation for the next frame or job associated with the application proportionally, effectively "borrowing" the allocation from the next frame. To prevent indefinite borrowing, if as a result of this "borrowing" the allocation for a particular frame or job falls to zero, the per-application adaptor forces the next frame to be skipped. This is a policing measure designed to enable the system to catch up from past resource overconsumption.

As shown in Figure 3.5, the per-application adaptor selects a suitable configuration by first collecting detailed, short-term predictions of the behavior of the job from the application predictors. It then queries other system layers for energy estimates, and chooses the application configuration which minimizes the amount of energy that will be consumed. It is also responsible for notifying the global coordinator when changes in the operating environment make it impossible to achieve the expected utility within the existing resource allocation, or if the amount of energy consumed or resource utilization varies greatly from the amount assumed by the global coordination.

Unlike the allocation of variable resources across applications done by the "global" coordinator, we do not consider the effect that the task currently being scheduled has on other tasks. Compared to the global allocation problem, which (ideally) requires considering every single possible combination of configurations across all applications and system layers, this optimization is simple and can be done frequently without incurring excessive overhead.

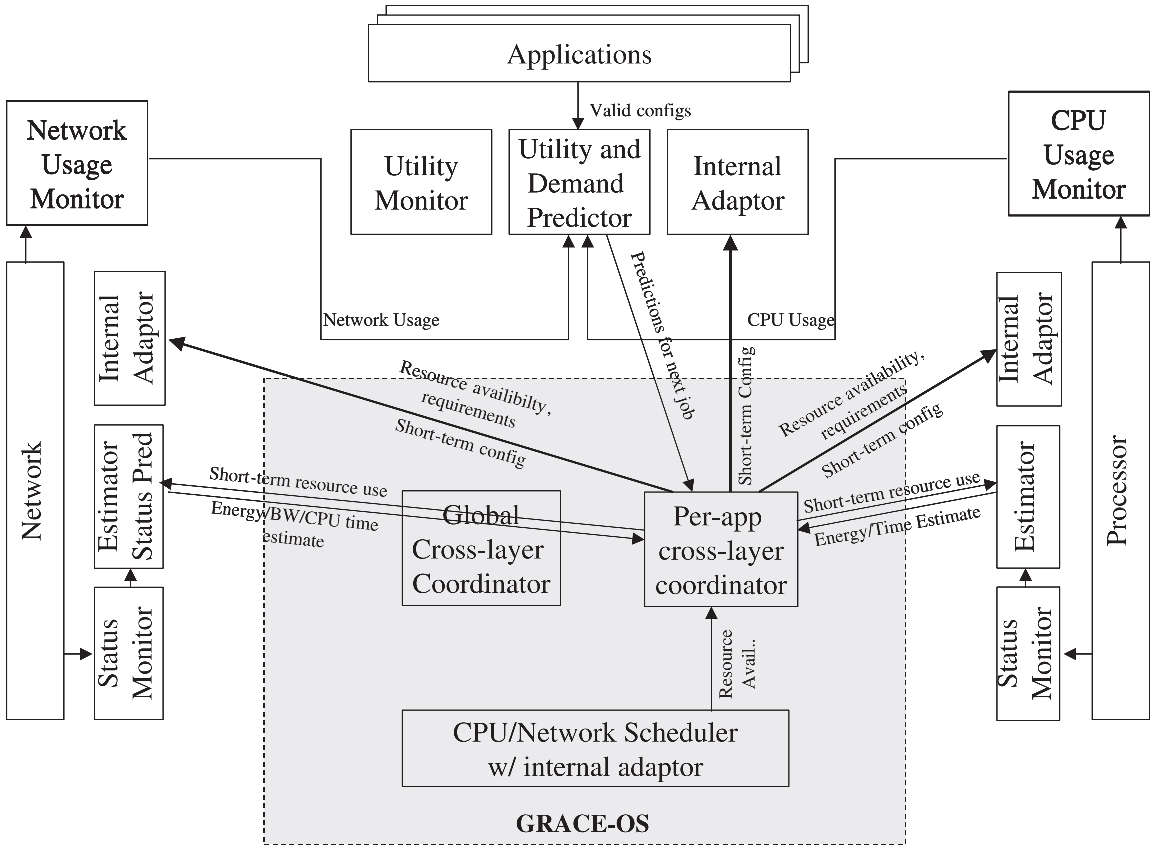



**Figure 3.5** Per-application adaptation communications.

### 3.3.3 Internal or "per-layer" adaptation

Finally, each layer is capable of performing "per-layer" internal adaptation, as shown in Figure 3.6. The internal adaptations are adaptations, possibly across applications but within resources that are allocated to the layer, which minimize energy consumption but are not visible to other layers and do not affect user-visible aspects of performance. The internal adaptations must restrict themselves to finding configurations for the layer that minimize the layer's aggregate energy consumption.

Examples of such "per-layer" adaptations include allowing the scheduler to reclaim unused time from jobs once that job is finished, the hardware adaptation mechanism choosing the most efficient hardware configuration given the time available to a task and its instruction count, and the network choosing between various combinations of ARQ, forward error correction, physical-layer bit rate, and energy per bit.

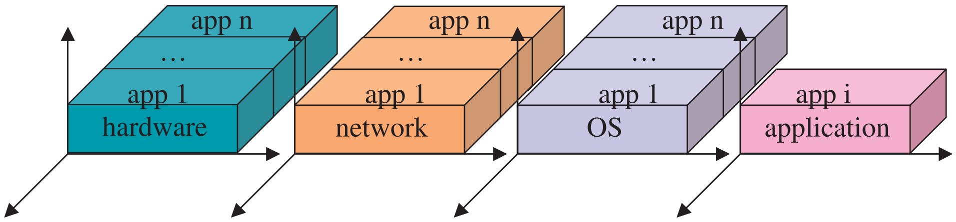



**Figure 3.6** Internal adaptation.

For all layers, the internal adaptation also includes implementing the configurations determined in the per-application adaptation process. This means looking at the actual application configuration chosen by the per-application coordinator. The application must reconfigure itself to use that configuration. The network and CPU layers must look at the associated expected resource consumption for that configuration, and reconfigure themselves to efficiently deal with that workload within the resources (CPU time, network bandwidth) made available to the application by the scheduler.

### 3.3.4 Job completion and feedback

After the job finishes its execution, the actual resource usage (including the number of instructions executed, the total amount of network bandwidth used, and the total energy consumption) is reported back by the resource usage monitors to the per-application coordinator. The per-application coordinator evaluates this information, and determines if it shows that conditions have changed significantly since the global adaptation was performed. If so, feedback is sent to the global cross-layer coordinator, which may choose to reevaluate the global resource allocation.

# CHAPTER 4

# GRACE INTERFACES AND ALGORITHMS

This chapter presents a description of the algorithms and interfaces used in the simulation of the GRACE system built as part of the testing and development process. The interfaces are built as an abstraction layer, which admits multiple realizations including a simulator (described in Chapter 5) as well as an implementation on actual hardware [72].

As an example of the use of these interface, I also present an adaptive application running on these abstract interfaces. This application, an adaptive video encoder, is based on the adaptive video encoder core presented in Chapter 2. It is presented along with a matching nonadaptive video decoder, which registers with the GRACE system using some additional interfaces designed to simplify the integration of existing real-time, nonadaptive applications into the GRACE framework.

## 4.1 Application Interfaces

The GRACE system includes a set of interface libraries that sit between the system resources and the application. These provide a high-level interface to application registration, scheduling, and network capabilities provided by the GRACE OS. In this section, we provide an overview of these interfaces.

The GRACE application interfaces are divided into three components: Global allocation and coordination, network management, and CPU and scheduling.

### 4.1.1 Global coordination and configuration management

The global coordination interface incorporates the following functions:

- Initialize and update the list of configurations available for the application

- Request that global allocation be performed
- Check if allocation has been updated and retrieve the application's current allocation
- Update global allocation tables based on actual performance of the application

Upon startup, it is the application's responsibility to load its configuration list into the global coordinator and request a global allocation. Then the application must wait for the coordinator to finish reallocation and issue an allocation.

The application must check for an updated global allocation at each frame. If a new allocation is issued, the allocation must be stored and sent along to the CPU and network interfaces.

### 4.1.2 CPU and scheduling

The CPU interface incorporates the following functions:

- Register as a real-time application
- Start a new real-time job
- Reconfigure the CPU based on predictions for available cycles and time
- End a real-time job
- Predict energy required for a given cycle count
- Retrieve information on cycles, time, overruns, and CPU energy consumed
- Report a new or changed global allocation to the CPU adaptor

The CPU adaptation procedure works in a somewhat counterintuitive way, because the predictions for the amount of CPU time and the number of cycles expected is not necessarily available when the job starts. Before the operating configuration of the CPU is set, the CPU is running using the application's real-time allocation but at an unknown CPU frequency; the CPU frequency will in fact be the last frequency used by the previous job.

The real-time job associated with each frame starts by running the per-application adaptor. The adaptor then runs the application predictor and selects an operating configuration for the application and CPU, and the reconfiguration request is sent to both the operating system's

CPU frequency controller and the application. After the operating configuration has been set, the application is permitted to begin its processing of the frame.

When the application completes its processing, an "end job" call is made. At this point the operating system computes estimates for the actual energy consumption of the CPU executing the task, and sends these values back to the per-application adaptor and from there to the application.

### 4.1.3 Network interface

The GRACE network interface currently only supports transmit functionality. This is because the GRACE framework does not currently have support for allowing a receiver to control a remote transmitter, which would be required for adaptation to occur on the receiver. Also, the present GRACE transmitter interface assumes that the processing required to support the network protocol is trivial compared to the processing required by the application and can be ignored. Therefore, $E_{proc}$ is assumed to be zero, and no explicit allocation is made for the network protocol; it is assumed that the processing demand and corresponding energy requirements are rolled up into the CPU demand for the application.

The current GRACE network interface incorporates the following functions:

- Initialize the network layer
- Estimate available bandwidth for the current job
- Estimate energy required to transmit a given number of bytes
- Transmit a frame
- Retrieve status of previously transmitted frames
- Send global configuration information to the network layer

These calls work at a relatively high level, and understand several application constructs such as frames, frame numbers, dependencies, and deadlines. These can be used by the network layer to intelligently manage the network buffer, for instance by dropping frames that will not be decoded because they are past their deadline or they refer to frames the decoder already indicated did not arrive. (The current implementation only does the latter.)

## 4.2 Global Coordination

In our system, the global optimization algorithm is separated from the framework, so that it can be easily replaced for testing different algorithms.

The implementation of the GRACE architecture presented in this chapter is incomplete in one important respect: the global allocator considers only energy and not utility. All applications are considered to be identical in utility and importance. In Chapter 6, the global optimization problem will be revisited, extending the optimization to cover utility as well. Also, in this implementation, applications are responsible for policing their own behavior and ensuring they do not use more CPU time or network bandwidth than they have been allocated. Future implementations of the GRACE architecture will move policing responsibility into the operating system.

In this implementation of the GRACE interfaces, global coordination is handled in a separate coordinator process. This coordinator reports allocations to the global-coordinator interface in each application, which then relays the information to the associated CPU and network adaptors.

### 4.2.1 Coordination protocols

GRACE applications register with the coordinator through by sending a list of configurations to the coordinator, which then replies with an appropriate configuration and allocations of computational and network resources.

Communication between the applications and the coordinator is managed by the GRACE library linked to each GRACE-aware application, which communicates to the coordinator process using using a message queue. The present implementation does not implement any security model that prevents malicious applications from posing as other applications.

The configuration request message contains the following information:

- A flag indicating whether global reallocation should be performed.
- The number of configuration records being sent in this message.
- Zero or more configuration records.

Applications are added to the GRACE system when the first request for an allocation with that process ID is received, and detached from the GRACE system when a configuration request containing zero configuration records is sent. The coordinator also checks periodically (presently at a one-second interval) that each registered process ID is still running on the system. If a process is found to be no longer running during this check, a detach request is automatically generated, and the resources associated with that application are reallocated to the other real-time applications running on the system.

The configuration record is a structure describing each configuration to the global coordinator. It provides the following information for each configuration:

- Configuration number
- Average (mean) cycles consumed
- Average (mean) bytes consumed
- 90th percentile cycles consumed
- 90th percentile bytes consumed
- Period or interval between job release (microseconds)
- Maximum permissible job lateness (microseconds)[1]
- Per-job utility for successful completion
- External energy associated with the job

The per-job utility and external energy values are not used by the global optimization algorithm described in the next section, which assumes all applications are of equal importance. This limitation will be addressed by the optimization algorithm presented in Chapter 6.

[1]Normally, when an EDF scheduler is used with a periodic task, jobs are required to complete before the next job for that application begins. However, for our soft real-time system, frames are permitted to "borrow" time from subsequent frames if they overrun. We permit this by setting the maximum allowable lateness to a nonzero value. Any job that exceeds its allowed lateness will be terminated.

### 4.2.2 Global optimization algorithm

The optimization algorithm we use converts the optimization problem into a set of knapsack optimization problems, and then uses well-known algorithms to solve the knapsack problems. The inputs to the global optimizer are the list of applications and the associated configurations, byte counts, and cycle counts, along with an estimate of the available network bandwidth in total bytes per second.

The outermost loop for the optimizer fixes the CPU speed, evaluating the feasibility and total energy consumption at each of the native operating frequencies offered by the CPU. Although this is suboptimal, the power lost by using this heuristic can be bounded using Jensen's inequality to be no more than the difference in power between adjacent CPU speed steps. And in practice, the per-application adaptation allows us to make achieve these intermediate speeds and make up most of the energy we would otherwise lose.

The inputs available to the inner optimizer are therefore the CPU speed (fixed by the outer loop), the network bandwidth, and the list of applications and associated configurations. At this point there is sufficient information available to convert the list of application configurations into figures for energy, percent CPU utilization, and percent network utilization for each configuration.

This is a classic knapsack problem setup in two dimensions. Utility is defined as an offset minus energy; the capacity of the CPU and network dimension is 1 minus an unallocated reserve for best-effort applications. The knapsack problem is solved using either a brute-force optimizer that evaluates all possible combinations of configurations, or an optimizer based on work presented by Moser et al. [73]. Currently, we use the brute-force optimizer for optimizing one or two applications, and the Moser optimizer if allocations for more than two applications are being made.

When the optimization is completed, the set of application configurations with the highest utility (lowest energy) is saved, along with their associated allocations (percentage of network, percentage of CPU). Any unused CPU time or network bandwidth is distributed across applications proportional to their allocation.

It is possible that no valid combination of configurations will be found at a given CPU speed. If this occurs, the CPU speed will be marked as “infeasible“ and the next higher CPU speed will be evaluated.

If the optimizer fails to find a valid combination of application configurations at any possible CPU speed, allocation is performed via a “backup” method that, rather than minimizing energy, attempts satisfy the CPU constraint while minimizing the network overrun. This backup optimizer sets the CPU to its maximum frequency, and then solves a one-dimensional knapsack problem with the CPU utilization as the constraint and the negated network utilization used as the “value.” In other words, the set of application configurations meeting the CPU utilization constraint that minimize total network utilization is selected. Allocations of CPU time and network bandwidth are then made proportionally based on the 90th percentile demands of each application. This ensures fairness and is likely to generate a workable allocation, even if a 90th percentile allocation of network bandwidth cannot be made.

The reason for this process is that the predictions are typically much more accurate for CPU utilization than for network bandwidth. As a result, the difference between the expected network utilization and the 90th percentile allocation level is large. The backup optimization allows us to make a reasonable allocation, even if no allocation allows us to satisfy the 90th percentile demands. If the backup allocation fails—in other words, if the CPU is overloaded no matter what configuration is made—all applications are assigned configuration 0 and all resources are divided evenly.

Although the global optimizer chooses an application configuration, the per-application adaptor is only required to respect the allocations of CPU time and network bandwidth, not the exact configuration choice. As a result, the actual application configuration and CPU frequency used to encode any particular frame may vary from the values chosen by the global optimizer.

## 4.3 Per-Application Coordinator

Linked into fully adaptive GRACE applications is a system component we call the per-application coordinator. This component is responsible for determining the immediate resource availability, and working with the application predictors to choose and implement an appropriate application configuration for each job.

Although the per-application adaptor is part of the GRACE system, it runs in the application's process space and communicates with the scheduler via system calls. The CPU and network adaptation and estimators are also implemented partly in the per-application adaptor code linked into the GRACE application.

### 4.3.1 Step 1: Predict resource demands and availability

The first step of the per-application coordination process is to collect the resource consumption for the job about to be scheduled.

To do this, the application adaptor first calls the application predictor, which identifies one or more valid configurations, dependent on the current state of the application. These configurations are returned tagged with estimates for the byte and cycle counts.

Also, due to the reclamation of unused CPU time and variation in network bandwidth, the actual resource allocation for a frame may vary from the originally expected value. For this reason, the per-application adaptor also queries the CPU and network scheduler to determine an updated estimate for both the availability of CPU time and of network bandwidth for the next job.

### 4.3.2 Step 2: Determine energy costs for application configurations

After the potential configurations for the job are identified and their resource requirements estimated, the per-application coordinator queries the CPU and network estimators to determine the feasibility and energy consumption of each application configuration returned by the application predictor. When the per-application adaptation is performed, the per-application coordinator queries the hardware predictor about the actual energy required to complete the estimated workload *within the time allocated by the scheduler.*

Likewise, the network estimator is queried to determine the amount of energy that will be required to perform the network activity associated with a proposed configuration.

### 4.3.3 Step 3: Determine most efficient application configuration

Finally, the network and CPU energies for each potential configuration of the application are added, and the configuration that minimizes the total energy consumption for this particular job is chosen.

The per-application optimizer simply looks at the list of application configurations for which energy has been calculated, and returns the application configuration that uses the least energy while still meeting the present resource constraints. Once a suitable application configuration is selected, the CPU adaptor is sent the estimated cycle count so the processing speed can be set appropriately.

Depending on the application predictor that is selected, the per-application adaptor may be presented with only one application configuration. In this case, the application configuration selected by the predictor is used, even if predictions indicate it will fail.

## 4.4 Adaptive Encoder Application

To validate our system design and evaluate its potential for saving energy, applications must be designed to work within the framework. This section presents one such application, an adaptive video encoder based on the adaptive encoder core presented in Chapter 2.

### 4.4.1 Application structure

The application is split up into several parts. These parts are:

- Encoder core
- Application sequencer
- Application predictors

It is linked against the GRACE libraries, which include the CPU and network predictors, the per-application adaptor, and the interfaces to the global coordinator, scheduler, and network layer.

Because the GRACE system provides much of its functionality in libraries that are linked to the application, the developer has the freedom to change or replace these functions with ones better suited to a particular application. As a side effect, though, the actual GRACE system components cannot be considered "trusted" by the system, and their work must be checked against the actual allocations. Monitoring functions must therefore be handled in the GRACE system core. However, the current implementation of the GRACE system assumes "friendly" applications and does not attempt to enforce resource allocations.

### 4.4.2 Encoder core

Although we have designed several applications that register with the GRACE kernel and use the GRACE infrastructure for resource allocation and monitoring, we have at this point only written one fully adaptive GRACE application. This is a video encoder application, built around an adaptive video encoder core derived from the free (GPL) TMN H.263 [69] encoder.

Our modified encoder was first presented in [64], and is described in detail in Chapter 2. This adaptive encoder allows a trade-off between the bit rate and processing time required for encoding, offering the ability to find optimal tradeoffs between quality, bit rate, and computational complexity. These parameters are controlled on a frame-by-frame basis by the GRACE software described in this chapter.

### 4.4.3 Communications protocol

The encoder application talks to the decoder via a pair of UDP sockets. The first socket is used to transmit the actual image data. The protocol splits up each frame into a header packet and one or more data packets.

The header packet contains the frame number, what frame (if any) the frame is predicted from, the total length of the frame, and the playout time for the frame.

The data packets each contain the frame number, the offset of the particular data chunk from the beginning of the frame, and a chunk of frame data. The frame data chunk is at least 32 bytes (to ensure that the data packets can be distinguished from header packets), and less than or equal to a maximum packet length, currently set at 1024 bytes.

The frame can only be reassembled and successfully decoded if both the header and all of the associated data packets arrive at the receiver. No provision is currently made for decoding a frame if a header packet arrives out of order. Reordering of the data packets does not cause decoding to fail as long as they are all received before the subsequent header packet.

The second socket is used to transmit feedback from the decoder back to the encoder. The decoder sends reports for each frame that it sees (i.e., any frame for which either the header or one or more data packets arrive). It does not send any report for frames for which no header or data packets are received; therefore, the protocol implementation generates an implied NAK whenever there is a missing frame number in the acknowledgment sequence.

### 4.4.4 Error recovery

The feedback link with the decoder is used to prevent error propagation between frames when the decoder encounters a transmission error. Before the encoder encodes any frame, it checks the acknowledgment stream to determine what frames are safe to predict from. To do this, it uses the following rules:

- If no negative acknowledgments have been received since the last frame was encoded, predict from the previous encoded frame.
- If at least one negative acknowledgment has been received since the last frame was encoded, predict from the previous frame that received a positive acknowledgment. If no frames have been positively acknowledged, force an I-frame.
- If more than 20 consecutive negative acknowledgments have occurred, force the next frame to be encoded as an I-frame.

These rules ensure that the decoder is always presented with frames that it can decode, as long as the acknowledgments have returned from the decoder. The last rule ensures that the encoder and decoder can resynchronize after unusual circumstances cause the encoder to have an incorrect previously acknowledged frame. Circumstances that can cause this condition include the decoder restarting and excessive numbers of lost positive acknowledgments on the feedback channel.

### 4.4.5 Application sequencer

The sequencer is the main loop of the application. It is responsible for calling the various application components, including the core, per-application adaptor, and image capture. It also mediates communication between the encoder core and the network and CPU adaptors.

The main purpose of the "application sequencer" is to simplify the construction of a GRACE application, by splitting the application up into a well-defined set of tasks and calling these tasks at appropriate times. In this way, a lot of the complexity of writing adaptive applications can be abstracted out to simply providing calls to encode a frame and switch configurations, and information about the various configurations available. The sequencer and GRACE libraries provide the per-application adaptor, interfaces between the system components, and the application predictors.

### 4.4.6 Predictors

Resource use estimates are provided by application-specific predictors. Our video encoding application integrates several different prediction algorithms, but all serve the same purpose: Estimate the resources required to encode and transmit a frame using a particular configurations, enabling the selection of the optimal application configuration for each frame.

In order for the adaptor to choose the best available application configuration, the application must include the ability to predict the performance of the various configurations available.

We implement this by including a separate prediction module with the application. This module is responsible for identifying a set of feasible configurations, and supplying estimates for the number of cycles the configuration will use and the number of bytes of encoded data it will generate.

## 4.5 Application Prediction Algorithms

Key to saving energy with the GRACE architecture is being able to make good decisions about what application configurations best match current conditions. Therefore, substantial research effort was devoted toward developing and evaluating different prediction algorithms used by the application and the GRACE system to predict the performance of the encoder.

### 4.5.1 One-step oracle predictor

The first of these prediction algorithms, "oracle," is not a prediction algorithm at all. Instead, it encodes each frame many times, once for each of the different application configurations, and saves the number of bytes and cycles used to encode the frame in each configuration. These values are then sent as "predictions" for byte and cycle counts to the adaptor. The oracle, therefore, represents what we could do with perfect information about the next frame.

We term this a "one-step" oracle because although the oracle provides exact information about the performance of the next frame, it does not necessarily find the globally optimal solution. This is because what we do this frame can affect the prediction used to encode the next frame. This means that occasionally another algorithm, such as the use of a fixed configuration, may actually result in a lower total energy consumption than the configurations chosen by the oracle predictor.

### 4.5.2 Fixed-configuration predictor

The second prediction algorithm, "fixed," only makes predictions for a single configuration at a time. (The configuration is specified either by a command-line option or by the globally selected application configuration.) It returns as a prediction the number of bytes and cycles used by the previous frame. If there is no valid previous frame, it instead forces the CPU to run at its maximum possible speed, and estimates that zero bytes will be transmitted.

This predictor is used for comparison as the baseline "no-adaptation" case. However, even when this predictor is active, CPU frequency adaptation still occurs.

### 4.5.3 One-step linear predictor

The second prediction algorithm we developed (first introduced in [64]) uses predetermined linear predictors to estimate the number of bytes and cycles that will be required for the next frame to be encoded, based on the configuration used for the previous frame and the number of bytes and cycles it took.

These predictors take the following form:

$$B(t)_{est} = B(t-1) \times m_B(old\ conf, new\ conf) + b_B(old\ conf, new\ conf) \tag{4.1}$$

$$C(t)_{est} = C(t-1) \times m_C(old\ conf, new\ conf) + b_C(old\ conf, new\ conf) \tag{4.2}$$

Because there are 16 different possibilities for $conf_{t-1}$ and $conf_t$, there are a total of 256 predictors for bytes and cycles that must be calibrated. This is done offline, by encoding a reference sequence (a composite of several MPEG test sequences) repeatedly, randomly switching configurations between frames. The linear predictors are fit in a least-squares sense against this calibration data.

Unfortunately, while these predictors worked fairly well between reasonably similar application configurations, they had a large mean-squared prediction error when used to predict more dissimilar configurations. This resulted in situations where the prediction error was so large that very inefficient configurations were chosen.

We therefore explored other predictors in an effort to find a design that was less prone to choosing very-wrong configurations. Although we could show energy savings with these predictors in the context of simple network models [64], they did not perform well when bandwidth constraints were added.

We first explored using more complicated predictors, such as polynomial functions and more than one step of history, but found no set of predictors reliable enough to use in constrained situations.

### 4.5.4 Reactive approach: "adaptive" predictors

The last prediction algorithm developed (and the most effective) we call "adaptive" in our comparison. Unlike the previously introduced predictors, the "adaptive" predictor does not estimate byte and cycle counts for all possible application configurations. Instead, the predictor directly chooses a single best configuration and reports estimates for byte and cycle counts for that predictor only to the per-application adaptor.

The algorithm used by the "adaptive" predictor to choose the best possible application configuration can be thought of as a three-step processes. These steps are:

1. Find the application configuration that, in the absence of constraints, minimizes energy consumed encoding one frame.

2. Find the application configuration that is numerically closest to the best configuration, which is estimated to meet resource availability constraints.

3. Update the resource-constraint tables to reflect current conditions.

#### 4.5.4.1 Finding the energy-optimal configuration

To find the energy-optimal configuration, the "adaptive" algorithm determines the lowest-energy configuration based on precomputed tables listing the average number of bytes and cycles generated by the encoder for different configurations on "typical" input. Combined with current predictions for network energy per byte and CPU energy per cycle, this information is used to determine the most efficient configuration for the application.

However, this most-efficient configuration may not meet current constraints on available bandwidth or CPU time. Therefore, we choose the configuration closest to the "energy-optimal" configuration that meets both of these constraints. We do this using a reactive approach, where recent resource use for particular configurations is fed back into the prediction algorithm and used to choose configurations that are not likely to exceed available resources.

#### 4.5.4.2 Prediction tables

The reactive predictor works by maintaining several sets of tables, one set of tables per set of "equivalent" configurations. The required tables are:

- Expected bytes required to encode a frame, for each configuration
- Expected cycles required to encode a frame, for each configuration
- Most recent byte count required for each configuration
- Most recent cycle count for each configuration
- Mappings from the CPU constraint (in cycles) to required configuration
- Mappings from the network constraint (in bytes) to the required configuration

The first two tables describe the application configurations that are available. There is one table entry per application configuration; if an application offers 16 configurations for a particular utility value, tables have 16 entries each.

The last two tables are used for quickly reacting to changes in both the video stream and present conditions. They are indexed by breaking the bandwidth constraints into "buckets," with each bucket mapping to an application configuration. The CPU constraint is divided into 30 buckets. The first 29 of these buckets is each 5 million cycles wide; this covers CPU constraints up to 135 000 000 cycles. Any value exceeding the limit that can be represented by the table is mapped to the highest table entry, which is effectively considered to have infinity as its upper bound.

The network constraint is divided into 21 buckets, based on the maximum number of bytes that will be generated by the encoder $b_{max}$. (This number is extracted from the global prediction tables.) The first 15 of these buckets are each $\frac{b_{max}}{30}$ bytes wide, covering up to $\frac{b_{max}}{2}$ bytes of available bandwidth. The next 5 buckets are $\frac{b_{max}}{10}$ bytes wide. The last bucket covers everything above the maximum number of bytes $b_{max}$.

The "buckets" each contain the configuration number associated with the constraint. They also contain a counter for the number of bucket underruns, which is used to update the tables.

#### 4.5.4.3 Table initialization

The average byte and cycle tables are initialized by setting their values to those contained in global configuration database. The previous byte and cycle counts are also initialized from the global configuration table. The last byte count for each configuration is initialized to the average byte count, and the last cycle count for each configuration is initialized to the 90th-percentile cycle count.

The mapping tables are then initialized using the 90th-percentile data from the global-prediction tables. For each possible application configuration associated with the current quality, the 90th-percentile bytes count is mapped into its associated bandwidth bucket. Then this and all higher-byte-count buckets are assigned to the application configuration. Likewise, the 90th-percentile cycle count is mapped to its associated cycle-count bucket, and it and all lower cycle-count buckets are mapped to this configuration.

Because this process is repeated for each application configuration in order of increasing byte count and decreasing cycle count, this assigns each to appropriate values based on the 90th-percentile estimates.

This initialization is only done when the global allocation changes the working utility; even if the global configuration feedback results in changes to the global tables, these changes are not fed back into the mapping tables. The table update process takes care of updating the configuration tables in response to the current conditions.

#### 4.5.4.4 Configuration selection algorithm

The configuration is selected with a two-step process. The first step is to identify the configuration which, in the absence of all constraints on CPU time and network bandwidth, would consume the least energy.

To do this, the predictor estimates the energy required to encode and transmit an average frame under the current conditions for each application configuration. It uses the number of bytes and cycles required from an average frame, pulling these numbers from the associated tables (and indirectly from the global-adaptor interface). The number of cycles expected from an average frame is mapped into a CPU speed required to run the configuration within the available time, or the maximum speed if the CPU cannot run fast enough, and an estimate of the total energy required for the encoding is computed based on the estimated speed. Likewise, the number of bytes expected for an average frame is mapped into an energy value. These values are added, and the configuration with the lowest total is called the "energy-optimal configuration."

However, this "energy-optimal configuration" is not necessarily achievable; it may require too much CPU time or too much network bandwidth to use under the current conditions. Therefore, the mapping tables are consulted to find a range of valid configurations. The number of available cycles (calculated from the time allocation minus any overrun from the previous frame, times the maximum CPU speed available) and bytes (calculated from the network allocation times the current bandwidth, minus the previous overrun) are computed. The byte limit is looked up in the byte mapping table, which gives a minimum compression level required (i.e., maximum configuration number). Likewise, the cycle limit is looked up in the cycle mapping table, which gives a maximum compression level (i.e., minimum configuration number).

If the "energy-optimal" configuration falls within these limits, that configuration is set and used to encode and transmit the frame. If it does not, however, the configuration numerically closest to the "energy-optimal" configuration but within the feasible range is selected. If no configuration is feasible, the current version of the code selects configuration 0, i.e., to do all possible compression.

#### 4.5.4.5 Bucket update algorithm

The expected bytes and expected cycles table are read from the global configuration database. Therefore, if the profile data contained in this database is changed, the expected-bytes and expected-cycles prediction tables will be updated as well. However, as we have not yet developed effective algorithms for updating this data, no such updates occur in the present GRACE system implementation.

The previous cycle count and previous byte count tables are updated based on the actual cycle and byte count for the last frame encoded. However, to prevent potentially unreliable data from corrupting the byte count table, the increase is limited to 1.1 times the previous value. Also, the saved value is only updated if the frame dependency is on either the previous if the frame's dependency is not on either the previous frame or nothing at all.

Both the CPU-constraint mapping table and the network-constraint mapping table are updated after every frame based on the performance of the encoder. The update algorithm only activates if one of the tables constrained the selected value for the application configuration, and it only affects the table that constrained the value.

For the network tables, the update algorithm compares the actual number of bytes generated by the encoder against the limits of the bucket corresponding to the original constraint.

First, the actual configuration used is compared against the value stored in the applicable bucket. If more compression was used than the table requires (i.e., the configuration number is less than the value in the table), no update is performed unless the actual byte count of the configuration used exceeds the upper bound of the bucket. In this case, the bucket is immediately reset to one less than the configuration used to encode the previous frame.

If the network table is controlling the configuration (i.e., if we are bandwidth-constrained), the network table is updated. If the number of bytes that the encoder generated was less than the lower border of the bucket, a counter is incremented. If, after the increment, the value of the counter is three, the value stored in the bucket is incremented, decreasing the compression (increasing the byte count) and decreasing the amount of CPU time required.

If the actual byte count is within the range of the bucket, the counter is cleared, and if the byte count is larger than the bucket, the counter is cleared and the configuration associated with the bucket is decremented immediately.

The configuration number stored in the table is then clipped to the range of available configurations.

Next, if the table value was increased, the new value is then checked against the previous byte-count table to make sure that it is appropriate for the bandwidth constraint. This is done by multiplying the "high" table limit by 1.5, and then comparing that to the previous byte count table. If the previous byte count is greater than the "high" table bound times 1.5, the configuration is assumed to use far too much bandwidth to be feasible and the previous value is restored in the network-constraint table. This accounts for the possibility of a large difference in bandwidth required for two adjacent configuration numbers.

Finally, higher-number buckets are all checked for monotonicity. If the value in any higher-numbered bucket is numerically less than the value of the current bucket, it is replaced by the value in the current bucket.

The CPU cycle table is handled slightly differently. Instead of waiting until three underruns have occurred before changing the active configuration, it changes the constraint to configuration mapping immediately. This is because the processing occurs before the transmission of the frame, and if the amount of processing time is increased into the next frame's allocation, it is likely that the time can be made up by the network layer; the network layer is generally not being pushed in the CPU-bound case. There is also no automatic lockout of configurations based on the CPU constraint value.

Finally, higher-number buckets are all checked for monotonicity. If the value in any higher-numbered bucket is numerically greater than the value of the current bucket, it is replaced by the value in the current bucket.

## 4.6 GRACE Support for Nonadaptive Applications

As part of the GRACE system, we built a simplified API to allow multimedia applications that are not adaptive to interface with the rest of the GRACE system. This library has three calls:

`grace_register` Request an allocation, given demand estimates (average and 90th percentile cycles and bytes, and an associated period). Note that because the application is non-adaptive, we do not need to collect information on utility.

`grace_new_frame` Finish a job and wait for the next period.

`grace_deregister` Terminate real-time operation and return the application's CPU and network allocation to system.

This API allows the decoder, which cannot adapt and does not need the full per-application adaptor structure, to be smaller and maintain the structure of a conventional nonadaptive application while still working within the GRACE framework.

The fixed application interface includes a small version of the per-application adaptor and predictor. Presently, this sets CPU frequency assuming that the workload for each frame will be the same as the workload for the frame before it. Energy estimates are not used or supported, as these are unnecessary for applications that do not support adaptation. This implementation also does not support detection of or response to varying frame encoding methods (i.e, I/B/P frames), nor does it support applications whose workload can vary on a frame-by-frame basis. Such applications could use the full GRACE interface, or an expanded version of the simplified interface presented in this section.

## 4.7 GRACE Decoder

To decode the data streams encoded by the adaptive encoder and complement our adaptive encoder for GRACE system testing, we have also built a nonadaptive but GRACE-aware decoder. This decoder supports the basic GRACE functionality but does not include the components that support adaptive applications.

The decoder application is a simple, nonadaptive application that accepts a data stream from our adaptive encoder, decodes it, and displays it using an X11 window.

Although the decoder application supports decoding streams from the adaptive encoder, it does not include any mechanism to predict CPU load if the encoder configuration of the incoming stream varies. However, experiments have revealed that the range of variation is much smaller than the variation encountered as the encoder configuration is changed. Since for 802.11-based GRACE laptops the dominant consumer of energy is the CPU, if the GRACE system runs a two-way videoconference with a non-GRACE infrastructure node, the most power-efficient systemwide configuration is for the infrastructure to compress the sequence to its minimum possible size. This allows the mobile node to use as much of the total network bandwidth as possible to reduce its encoding complexity and hence CPU load.

The decoder maintains an archive of previous decoded frames to which the encoder can refer when it encodes a frame. This archive is kept in a circular queue (currently set to length 10). When a frame is received, the dependency information is read out of the header and the queue is searched for the appropriate history frame. The decoder's state is then reset so that the incoming frame is decoded with reference to the correct history frame.

If an incoming frame is incomplete – that is, the decoder does not receive both the header and all associated data blocks – or it refers to a frame that has not been successfully decoded, the frame is dropped. The decoder does not attempt to decode incomplete frames or frames for which it does not have the correct previous frame. This causes playback to be jerky when many packets are lost, but prevents distracting artifacts.

Due to the fact that the current GRACE environment does not support application clocking or blocking reads, the decoder is implemented using nonblocking operations only. When the decoder is dispatched by the GRACE kernel to decode a new frame, it reads out all of the packets waiting in its input buffer. If a complete frame can be assembled, it is decoded. Any remaining packets, up to one complete frame's worth, is held in a buffer to be decoded and displayed during the next frame interval.

Currently, the decoder does not respect the playout time stamped by the encoder application. This feature will be added in future revisions of the decoder.

# CHAPTER 5

# GRACE SIMULATION ENVIRONMENT

To evaluate the performance of the GRACE system in a repeatable environment, we have implemented a simulation of the GRACE kernel, scheduler, and network. This simulation platform has exactly the same interfaces to the CPU, network, and global optimizer core as the native implementation of the GRACE system.

Most of the simulation system is implemented in a simulator core that communicates with each of the applications and the global optimizer as shown in Figure 5.1. The simulator core contains an EDF scheduler for both the CPU and network, the CPU and network predictors and energy models, and the interface to the per-application adaptors in each application. It is responsible for collecting the global predictions from the application, monitoring the performance of applications and calling the global optimizer at appropriate times, and telling each application its allocation for each frame. It then schedules each frame's CPU and network utilization, and informs the application about whether or not the frame met its deadline.

It is difficult to meaningfully model other components, such as the display and the hard disk, since their energy consumption depends primarily on user settings (for display brightness and disk timeout) and the behavior of nonadaptive system components that the user is running in parallel with the media tasks. Therefore, for the purposes of both energy optimization and energy measurement we consider only the network and CPU energy consumption.

## 5.1 Architecture

The simulation system is built around a simulation of the GRACE framework written in Python. The simulation of the framework incorporates the scheduler for both CPU and network,

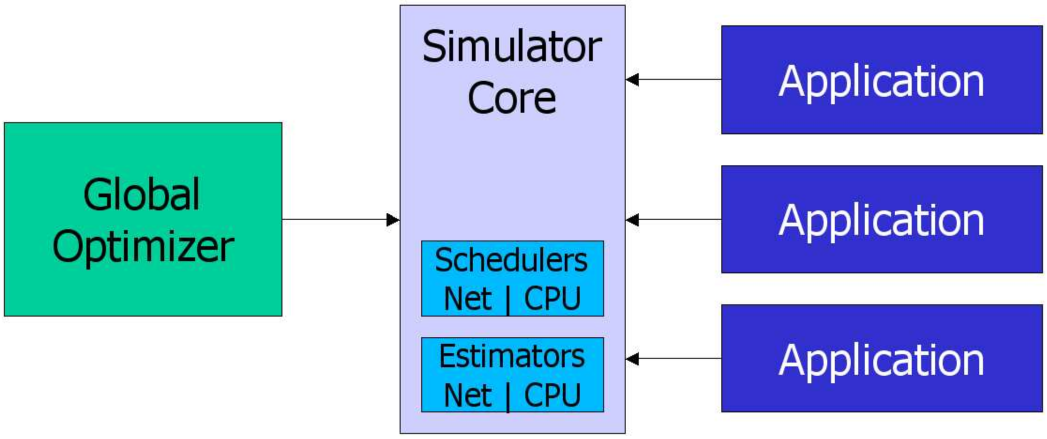



**Figure 5.1** GRACE simulator structure.

prediction and estimation for the energy consumption of the system, and an interface to the global optimization module.

### 5.1.1 CPU and network scheduling

The CPU and network scheduling is based on a pure EDF scheduler with no notion of budget or overrun handling. It is a preemptive system with 1-ms time slices. At any time, the ready process with the earliest deadline is scheduled on both the CPU and the network.

Our implementation does not take the conventional approach of deadline being equal to the start time plus the period. Instead, the deadline can be set independently of the period; typically we set the deadline to be the start time plus three times the period. This is because after the CPU completes its work, the job is rescheduled on the network but retains its current deadline. The three-times-period deadline allows time to encode and transmit a frame as well as a margin for overruns; therefore, despite the use of an EDF scheduler, a single application overrunning its allocation will not necessarily cause other applications to miss their deadline.

If the three-times-period deadline is missed, the frame is reported to have been lost. No effort is made to distinguish between failure to complete processing of the frame (a CPU overrun) and a network overrun. In both cases, the frame is reported as lost, and the application ensures that subsequent frames are not encoded with reference to the lost frame.[1]

[1]One simulation inaccuracy is that if a CPU overrun occurs, the scheduling of that job is terminated, but the application has in fact already run that frame to completion. It would be more accurate to force the incompletely encoded frame to complete its processing whether or not a CPU overrun occurs.

Although we do not explicitly evaluate the performance of a bidirectional communication setup in the simulations reported in subsequent sections, our scheduler does support both receiver and transmitter applications. Support for receiver applications is handled through the use of a corresponding transmitter, which attaches to the simulator but reports only network utilization and not CPU use. This transmitter then uses a back channel to communicate with the receiving application.

This means that receiving is considered the same as transmitting to the simulation environment. As a result, the network energy consumed for transmit and receive activities will be the same. Although this seems like it would be a poor model, it is actually a reasonable model for 802.11-style networks because dominant energy cost for an 802.11b network transceiver is the power consumed by the 802.11 MAC processor and other shared radio components. The actual radiated power is a small part of the total power consumption of the 802.11 device.

Another side effect of this model is that EDF scheduling is applied to the receiver as well as the transmitter, which would be difficult to implement in practice.

### 5.1.2 Monitoring and allocation enforcement

Because the EDF scheduler does not directly enforce the resource allocations, the simulator works with the application to ensure resources are consumed fairly.

As each new frame is dispatched, the simulator compares the amount of CPU and network time used for the previous frame against the actual allocation from the previous frame. If the frame used less network or CPU time than its allocation, it is provided a new allocation equal to the allocation provided by the global simulator. If, however, the previous job overran one of its allocations, the amount of overrun time will be subtracted from the allocation to the next job. This forces the application to "catch up" from the previous overrun by requiring it to use less of the resource that it overran in the previous job.

The application is responsible for checking the updated allocation every time a new job starts. If the allocation is zero, the frame must be skipped; this is used to keep the application from getting too far behind. (Currently the allocation will be zero if and only if the overrun from the previous frame is greater than or equal to the global allocation of the corresponding resource.) Otherwise, the application should attempt to choose an application configuration and CPU speed that enable the frame to be completed within the allocation, or that come closest to meeting the allocation if this is not possible. The details of this selection are left to the application and its associated predictors.

### 5.1.3 Global coordination

Unlike in the testbed, the applications request and receive the allocation from the simulation core instead of the global coordinator directly. In the simulation, the global coordinator requests are accepted by the simulator core, and passed along to the global coordinator when reallocation is required.

However, because the global coordinator needs to be portable onto the testbed, it is not part of the Python simulator core. Instead, it runs as a separate C process that communicates with the simulator. This allows the core of the global coordinator—the code that actually does the optimization—to be the same for both the simulator and the testbed.

The global allocation process used is the one described in Chapter 4.

## 5.2 CPU Model

Our simulation platform runs on a dual-processor Athlon MP 2000+ machine with 1 GB of RAM. The native speed of the processor is 1667 MHz.

### 5.2.1 Simulated CPU hardware

Because the desktop Athlon microprocessor does not support voltage and frequency scaling, we model a closely related Athlon Mobile processor, the Athlon XP-M 1700+. This processor uses the same core as the Athlon MP 2000+ in the simulation platform, but adds voltage and frequency scaling features.

We assume that the number of cycles that would be used on an actual Athlon XP-M processor is the same as the number of cycles used by the same code running on the simulation host. This assumption is justified by the prior work of Hughes et al. [71], which showed that the number of cycles used to encode multimedia streams is not significantly affected by the CPU clock multiplier.

### 5.2.2 Frequency and voltage scaling

The available processing frequencies and associated voltages and active power for the Athlon XP-M 1700+ microprocessor are shown in Table 5.1 [74].

**Table 5.1** CPU power vs. frequency for Athlon XP-M 1700+

| Freq | Power | Voltage | Rel. energy/cycle |
|---:|---:|---:|---:|
| 1466 MHz | 25.0 W | 1.25 V | 100% |
| 1266 MHz | 19.9 W | 1.20 V | 92% |
| 1133 MHz | 16.5 W | 1.15 V | 85% |
| 1000 MHz | 13.2 W | 1.10 V | 77% |
| 533 MHz | 6.4 W | 1.05 V | 70% |

### 5.2.3 Energy model

Energy required by the CPU is estimated by using the peak power demands of the processor model, along with the voltage scaling table and the rated maximum power dissipation of 25 W. We can plug the voltages into the standard voltage and frequency scaling formula (3.1) to get estimates of the CPU power for operation at lower frequencies. The CPU is assumed to draw the calculated power during operation, and no power when asleep.

Although the processor hardware is limited to five distinct operating frequencies, each application is allowed to choose any speed in the range of 533 MHz to 1467 MHz for each job. For our simulator, we model this by interpolating the power requirements linearly between operating points. This is because a real implementation of such a system could emulate an unsupported processor speed by dynamically switching between supported operating points as the job progresses.

## 5.3 Network Model

The GRACE group's research into network protocols for 802.11-style networks came to the surprising conclusion that to minimize the total energy consumption of the network layer, the best approach is to choose the maximum possible data rate that does not result in excessive packet losses [26]. This is due to the low radiated power limit for the unlicensed bands used by 802.11; with a maximum radiated power of 100 mW, the energy consumption of "overhead" components of the wireless interface, such as the media access-control and modulation hardware, is relatively large. As a result, the amount of energy that can be saved by reducing the radiated power is minimal, and if the data rate must be reduced in order to reduce the transmit power, the reduced energy consumption of the power amplifier does not counteract the extra energy required to keep the rest of the wireless interface awake. Therefore, a simulation model for an 802.11-style network needs to consider only a (potentially varying) maximum bandwidth and the energy consumed per byte transmitted. Taking advantage of this observation, our simulator reads the available network bandwidth in terms of application bytes per second, and the associated energy requirement in joules per byte of application data transmitted, from a bandwidth trace file.

To simplify implementation, we make two important assumptions. First, we assume that as long as the bandwidth constraint is not violated, transmission will be error free, and second that the network protocol is relatively lightweight and does not require explicit accounting for the CPU time associated with handling the transmitted data. An alternate interpretation of these assumptions is that the bandwidth costs associated with retransmissions and forward error correction are incorporated into the bandwidth trace, and that any required protocol overhead is incorporated into the application's CPU demand.

The network trace incorporates both instant bandwidth availability and energy requirements and long-term estimates. The instant bandwidth and energy values are used for both the actual simulation of the transmission and for the short-term estimation required by the per-application adaptor, and the long-term estimates are used for global allocation. The long-term estimates are formed by passing the actual network bandwidth and energy figures through a low-pass filter, and are intended to indicate long-term expectations for bandwidth availability.

Our simulation traces use varying data rates, intended to represent different potential operating environments. Although the data rate varies, we compute the energy-per-byte value by assuming that the wireless card has two states, active and shut down, and perfect power control. Whenever the network is not busy transmitting application data, the network card is shut down. The assumed wireless network interface draws 750 mW when it is active and transmitting data, and no power when shut down. The energy per byte value included in the network trace is therefore simply the power demanded by the active network divided by the number of application bytes transmitted per second.

## 5.4 Application Adaptation in an Unconstrained System

Our first set of results show the best-case effectiveness of application adaptation. In this section, we are not concerned about our ability to implement our adaptation algorithms in a practical system. Instead, we aim to show that the concept of application adaptation enables energy savings, and to sketch out how much energy savings we can achieve by exploiting our adaptive application.

### 5.4.1 Workload and simulation description

We do this by first evaluating the simplest possible implementation of application adaptation: the energy consumption of a single application running in an unconstrained environment. We run the encoder application encoding the "Foreman" test sequence (300 frames) at a CIF resolution and at 10 frames per second using our simulation environment. The application is adapted only globally; one configuration is chosen at its start and that configuration is used throughout the entire encoder run. All other forms of adaptation (the per-application adaptor, and CPU adaptation) are disabled.

Because we want to identify combinations of CPU and network that allow the application adaptation to provide energy savings, we look at a range of different CPU powers, while the network power is held at a constant value. This allows us to cover not only the regime where CPU power dominates the system power, but also regimes where the CPU and network both draw significant amounts of power and the regime where network power dominates.

Our energy model is therefore similar to that described in Section 5.1. The network power is fixed at 750 mW for an achievable bandwidth rate of 1 600 000 bytes per second. This bandwidth is sufficient to transmit a CIF-sized video stream at 10 fps without any encoding. This corresponds to an energy requirement of approximately $4.7 \times 10^{-7}$ joules per byte.

The CPU is also similar to the one described in Section 5.1. Instead of fixing the CPU peak power at 25 W, we vary the peak CPU power across the range of 0.01 W to 50 W. We do this to determine how much energy can be saved using our application adaptation across all three CPU/network power regimes.

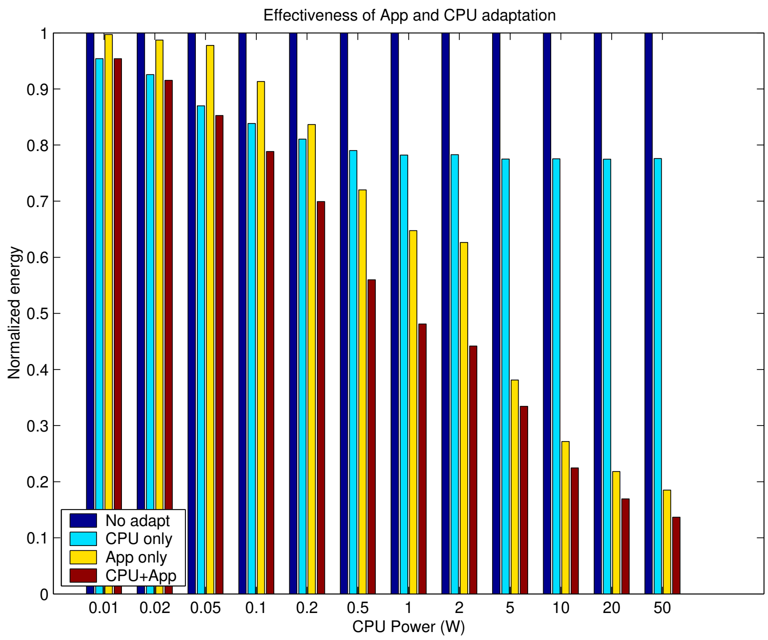



**Figure 5.2** Energy savings from global-only adaptation for varying CPU powers. Shown normalized, with 1.0 representing the energy use of the nonadaptive system.

### 5.4.2 Power savings from global adaptation

Figure 5.2 shows how the addition of globally optimal application adaptation and CPU adaptation affects energy consumption of our encoder. The vertical axis of this figure is the total (CPU plus network) energy energy consumed. The horizontal axis is the maximum (peak) power of the CPU, or the power that the CPU consumes when it is operating at its highest frequency. The network power remains fixed at 750 mW, so varying the peak power of the CPU sweeps from regimes where network power dominates to ones where the CPU power dominates as we go from left to right. The bars show the energy taken for various system configurations, normalized against the energy taken by the nonadaptive system. The nonadaptive system is defined as having a normalized energy use of 1.0.

We first consider the "CPU-only" case, adding CPU frequency and voltage adaptation (also known as DVS, or dynamic voltage scaling) to our unconstrained, single-application system. We implement DVS by scaling down both the processing frequency and the CPU power using the power scaling data in Table 5.1. The CPU frequency is set to 1000 MHz, which is the minimum operating frequency that permits the encoder to complete processing each frame within the 100-ms frame time. This results in a reduction of the CPU power by 23%, the difference between the CPU power at its maximum frequency and the 1000 MHz frequency required by the encoder. As a result, the energy savings we can achieve from CPU adaptation alone is limited to 23% even in cases where the CPU power is much greater than that of the network.

Next, we consider the "App only" case, which adds global application adaptation to the non-adaptive system. Global application adaptation chooses a single application configuration which minimizes the total network and CPU energy consumption for the entire sequence. Because there are no constraints on what configurations can be chosen, this results in the application selecting to send an uncoded stream when the CPU is the dominant consumer of energy. As a result, CPU utilization is minimized and a total energy savings of over 75% can be achieved when the CPU is the dominant consumer of energy.

For the "CPU+App" case, we examine the simultaneous adaptation of both the application and the CPU by jointly optimizing both these configurations. In other words, we choose the combination of application configuration and CPU frequency that results in the lowest total energy consumption encoding the entire "Foreman" test sequence while avoiding CPU overruns.

Both CPU and application adaptation are most effective in regimes where the CPU consumes a large fraction of the total system power. Because CPU adaptation reduces the power consumed by the CPU without any increase in the network's power consumption, unlike application adaptation it can provide some energy savings even if the amount of power consumed by the CPU is small. However, the effectiveness of CPU adaptation is limited by the energy per cycle required by the CPU at the lowest possible frequency: 70% of the energy per cycle required at full speed for our microprocessor model. Therefore, the total energy savings from CPU frequency and voltage scaling is limited to 30%, no matter how the CPU and network are utilized.

The use of application adaptation, on the other hand, can reduce the CPU workload tremendously, providing huge energy benefits when the CPU dominates the total energy consumption of the system. Furthermore, the benefits are synergistic. If the CPU workload is reduced sufficiently, we can reduce the CPU frequency to 533 MHz. This results in an additional 7% energy savings from CPU adaptation, resulting in a total energy savings of 83% for a "realistic" CPU power of 20 W.

These results clearly justify the value of performing adaptation of the application in contexts where the network is not severely bandwidth constrained. If the CPU draws a peak power of 20 W, we would be reducing the total system power by nearly 10 W. Even accounting for the energy costs associated with the rest of the system, this is a large energy savings. Assuming the idle power of approximately 10 W for the complete system, this means that reducing our application's power consumption from approximately 13 W to approximately 3 W decreases the total power consumption of the system by over 40%, greatly increasing battery lifetime.

### 5.4.3 Frame-by-frame adaptation

We next consider the addition of frame-by-frame adaptation to the combined CPU and application adaptation system shown in Figure 5.2. Since we are primarily interested in theoretical benefits of per-frame adaptation, we use exact information about resource demands and energy requirements to select a configuration for each frame. Specifically, before choosing a configuration for a job, this "oracle" algorithm determines the byte and cycle count for each possible application configuration, and then uses the byte and cycle count to compute an energy demand for each configuration. This information is then used to to select a configuration for each frame. As there are no network bandwidth or CPU utilization constraints on which configuration is chosen, the configuration that minimizes energy is always selected. The CPU speed is also selected so that processing of the frame is completed within its alloted time (i.e., 0.1 s). We allow the CPU frequency to be set to an intermediate point between two supported CPU speeds. This is handled by interpolation; the actual energy consumption is assumed to be equal to an appropriate linear combination of time spent operating at the next lower and higher supported operating point.

The use of an oracle to determine the application configuration allows us to specifically evaluate the effectiveness of per-frame adaptation in the context of a single unconstrained application, without considering the effects of suboptimal predictions or allocation. However, it is important to remember that this "single-step" oracle cannot account for dependencies between the particular encoding of frames, and therefore frame-by-frame selection of configurations that minimize energy for the current frame is not guaranteed to be globally optimal.

Figure 5.3 shows a comparison of the energy consumed by systems employing per-frame adaptation in different system components. Because at this point we are interested in the

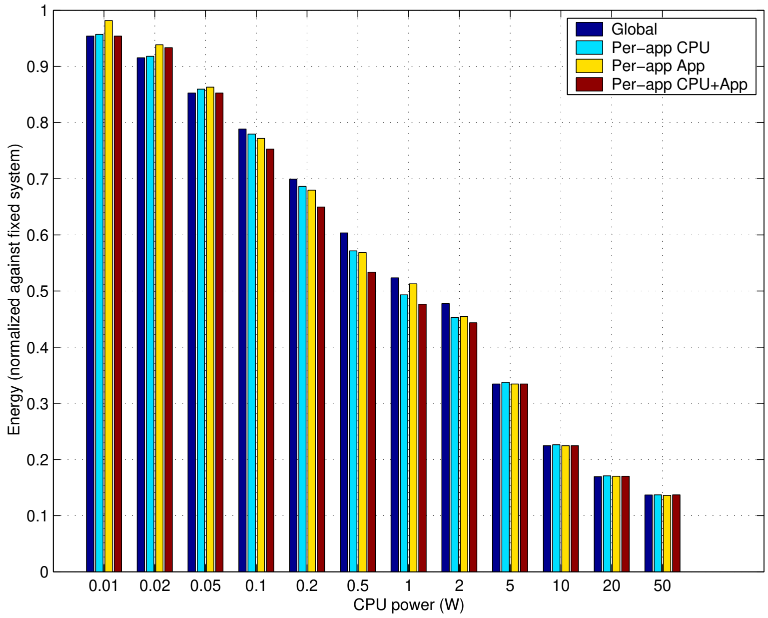



**Figure 5.3** Energy savings from per-frame adaptation for varying CPU powers. Shown normalized, with 1.0 representing the energy use of the nonadaptive system.

effects of per-frame adaptation, we start with enabling global adaptation for both the CPU and application as a baseline. We then add per-frame adaptation of the CPU ("Per-frame CPU"), of the application ("Per-frame App"), and of both the application and CPU ("Per frame CPU+App"). We maintain our normalized energy scale, so a normalized energy of 1.0 represents the energy consumption of the nonadaptive system.

Because the selection of the application configuration that minimizes total energy consumption is driven primarily by the relative power demands of the CPU and the network and not by changes in the application's behavior, in most cases we expect to see very little energy savings from doing frame-by-frame adaptation. And this is in fact the case, except in the intermediate CPU power range of 0.1 W to 2 W where the optimal application configuration is affected by changes in the source stream. Even in this range energy savings was small, peaking at about 7% more than the energy savings that can be achieved using global adaptation only. This increase in energy savings was split roughly evenly between that attributable to per-frame application adaptation, and that attributable to per-frame CPU adaptation.

Compared to the large energy savings afforded by global application adaptation, the additional energy savings associated with per-frame application adaptation is small and appears only when both the network and CPU contribute significantly to the total energy consumption. This is because neither the CPU nor the network dominate the total energy consumption, the variation in the stream does not result in a large enough change in the network or CPU utilization to drive a switch from "encode as much as possible" to "don't encode at all"; at most it induces slight variations in the optimal compression level.

Furthermore, the situation in which both the CPU and network consume significant amounts of power is more typical of PDA or cellular applications than modern laptops with 802.11-style wireless networking that is our primary focus. This means that without constraints on network bandwidth, per-frame adaptations of the application will be of minimal value on a higher-power laptop computer. Even on a PDA or cellular device, the amount of potential energy savings we abandon by not doing per-frame adaptation of the application is small.

This is an important observation, because it suggests a simple approach to the selection of an appropriate application configuration. Instead of considering the effect of application configuration on energy consumption, we can select the most energy-efficient configuration based on global statistics. We then use the per-frame application adaptation only to respond to constraints on the system. The performance of this approach will be examined in detail in Section 5.6.

## 5.5 Application Adaptation in Constrained Systems

Because per-frame application had minimal impact when the network bandwidth was unconstrained, we next examine the case where the network is subject to a bandwidth constraint. The network bandwidth constraint places a limit on the amount of complexity that can be shifted from the CPU to the network; some configurations that do little or no compression may exceed the network bandwidth constraint. This will reduce the energy savings that can be achieved when we would otherwise have shifted as much work as possible to the network.

The constrained case is important because, although 802.11b networks have a relatively high theoretical bandwidth, the actual bandwidth availability can vary dramatically depending on network utilization, distance from the base station, and interference. Furthermore, even in the best of conditions there is not enough bandwidth to transmit a 10 fps CIF video stream entirely uncoded on a standard 802.11b network.

For the purposes of this discussion, we introduce a bandwidth constraint of 100 Kbytes/s. This is the lowest fixed bandwidth that is sufficient to transmit the stream. The network power is unchanged at 750 mW. Note that because the network's power consumption has remained unchanged but the bit rate has been reduced, the amount of energy consumed per byte transmitted has increased significantly. We continue to use exact information (i.e., the "oracle" adaptor) to select an optimal application configuration that minimizes energy while also meeting the 10 Kbytes per frame (100 Kbytes/s at 10 fps) network constraint.

Because of the bandwidth constraint, we need to account for the possibility that frames will never make it to the receiver. To allow the system to recover from overruns, frames are dropped by the scheduler only if they have not been completely transmitted within three frame times of the start of their encoding. For a frame rate of 10 fps, the total time permitted from the start of encoding to the completion of transmission is 300 ms.

With the 100 Kbyte/s bandwidth constraint, even if the processor is run at its highest speed and all possible compression is done, there are still three frame drops over the 300-frame "Foreman" sequence. To permit some adaptation while maintaining a good application quality, we permit up to six dropped frames total when choosing the best fixed application configuration.

The "Foreman" sequence contains both a talking-head section and a more complicated pan-and-zoom section. For this reason, the 100 Kbyte/s bandwidth is only tightly constraining for roughly half of the total sequence. We therefore expect to see some benefit from the introduction of frame-by-frame adaptation.

Figure 5.4 shows the actual energy savings associated with global-only and global plus per-application adaptation, using the "oracle" adaptor. We see that for the higher CPU powers, when it is desirable to shift work from the CPU to the network, being able to dynamically choose a configuration that lowers workload while still respecting the bandwidth constraints allows significant energy savings. For the 20 W case, we see that adapting both the CPU and network on a per-frame basis allows an energy savings of 45% to be realized—this is nearly 20% more than the energy savings associated with global adaptation alone.

## 5.6 Realizable Energy Savings for a Single Application

In the previous sections, we have shown that given perfect information about the future behavior of the stream, we can potentially see significant energy savings from using our ap-

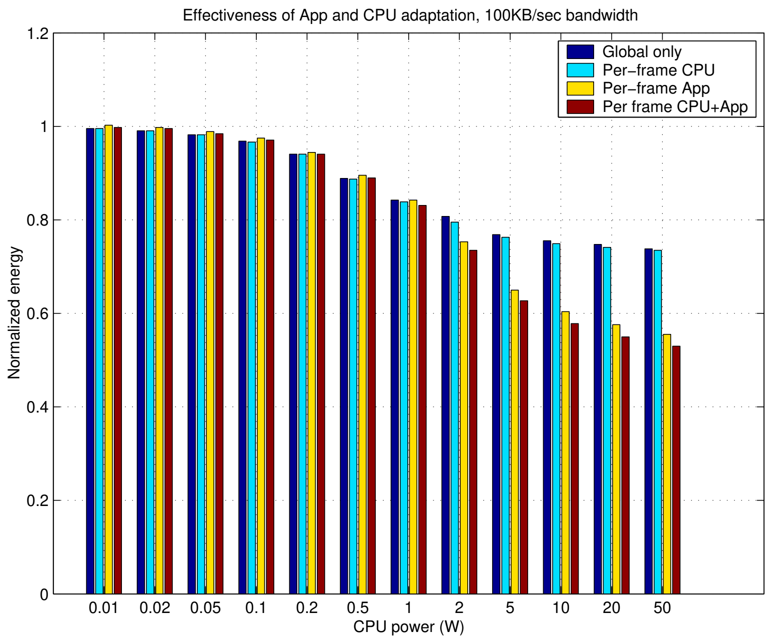
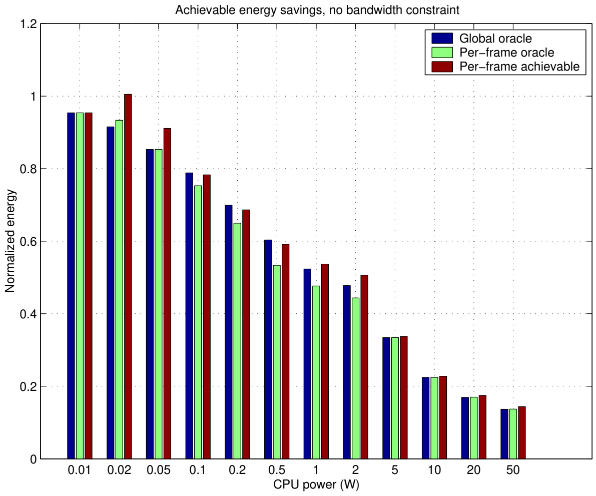



**Figure 5.4** Energy savings from per-frame adaptation with network-bandwidth constraints. Shown normalized, with 1.0 representing the energy use of the nonadaptive system.

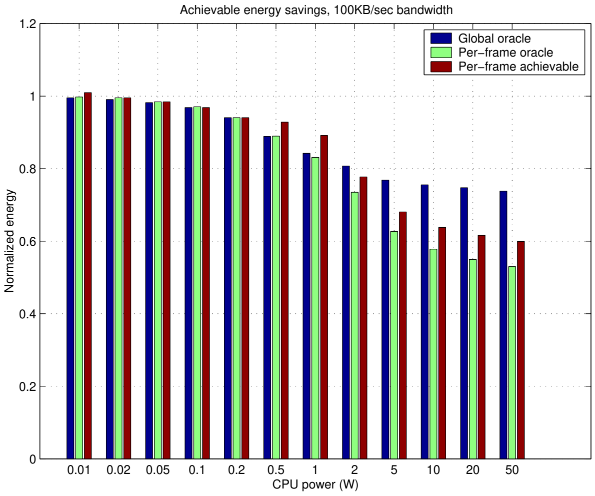
plication adaptation techniques. However, we have not shown that these savings can actually be realized without the use of oracles. In this section, therefore, we examine how much of the energy savings we can actually realize using the adaptation algorithm introduced in Chapter 2. The results of this comparison are shown in Figure 5.5. The unconstrained case, in which the network bandwidth is large enough to transmit the uncompressed stream, is shown in Figure 5.5 (a), and the case where the network bandwidth is constrained is shown in (b).

Because the adaptation algorithm is designed around the assumption that changes in application configuration are driven by the ratio of CPU and network power consumption and network bandwidth constraints only, it cannot realize energy savings from per-frame adaptation when the network bandwidth is not constrained.

In the constrained case, however, the per-frame adaptation is driven primarily by the constraint and we expect that our algorithm would achieve most of the realizable energy savings; we see that this is in fact the case. Our adaptation algorithm allows us to realize energy savings within 3% to 8% of the energy savings afforded by the per-frame oracle and its associated perfect information on the resource demands and energy consumption of upcoming frames. The




(a) Unconstrained network (1600Kbyte/s)




(b) Constrained network (100 Kbyte/s)

**Figure 5.5** Energy savings of oracle vs. actual algorithm. Shown normalized, with 1.0 representing the energy use of the nonadaptive system.

ability of this per-frame adaptation algorithm to achieve most of the potential energy savings justifies our simplified adaptation model.

## 5.7 Energy Savings from the GRACE System

In the previous sections, we have shown the value of application adaptation for saving energy in both constrained and unconstrained scenarios. But we have done so only in the context of a single application which is assigned all of the computing and network bandwidth provided by the system. Furthermore, the results in the previous section rely extensively on "oracles" that provide exact information on the resource consumption of jobs before they are actually run.

In this section, we simulate the performance of the multiapplication, multilevel GRACE system, as it would be implemented on a laptop computer. Because of the increased complexity of the model, it is difficult to exhaustively test all the different power regimes and workloads that we could possibly encounter in this environment. For this reason we fix the CPU power at 25 W and the network power at 750 mW active, levels that are typical of modern laptop computers. Again, both the network and CPU are assumed to draw no power when idle.

Because global allocation is requires solving a NP-hard problem, which are difficult even if heuristics are used, optimization overhead makes it impractical to solve the global allocation problem on a frame-by-frame basis [29]. As a result, these experiments consider only three systems: a fixed system that does no application adaptation at all, a system that includes global adaptation to allocate resources and application configurations as applications enter and leave the system, and a GRACE system that incorporates global resource allocation and frame-by-frame determination of the most efficient application configuration.

### 5.7.1 Workloads

To evaluate the performance of the GRACE system under varying system loads and application distributions, we consider three different workloads consisting of one, up to two, and up to three applications running simultaneously. These workloads start with the single-application workload we used in the previous section, and add additional encoders until the system is loaded to its capacity. The goal of using these workloads is to start with a moderate load, and examine the system's performance and energy consumption as the workload is increased to the point where the system begins to fail due to overcommitment of the network and CPU resources.

Although we currently only have a single adaptive encoder, we can introduce variation between the different applications running on the system by running the encoder at different resolutions and frame rates. For this reason, we run the first application using CIF resolution (352 x 288) and 10 fps, and the second and third applications at QCIF (176 x 144) and 15 fps.

The effect of these workloads is that, depending on the exact configuration of the adaptation algorithms and network, the system is lightly to moderately loaded running a single CIF encoder. It is under moderate to heavy loads with one CIF and one QCIF encoder operating, and heavily loaded or overloaded with all three applications running.

#### 5.7.1.1 One application

Our one-application workload consists of a single application running on the adaptive system. This workload is the same as the one we consider in the previous section: the application encodes the CIF-sized "Foreman" sequence at 10 fps. However, for this experiment we repeat the entire sequence four times, for a total of 1200 frames.

This single application does not load the system heavily. It never overruns the CPU, and only requires CPU frequencies of up to approximately 1000 MHz even if full compression is performed. Also, it requires only 100 Kbytes/s of network bandwidth to successfully transmit all but a small number of the frames of the sequence.

#### 5.7.1.2 Up to two applications

Our second workload consists of the same 10-fps CIF encode of the "Foreman" sequence, but this time with a 15-fps QCIF encode of the "Carphone" sequence running in parallel. The QCIF encoder starts after and ends before the CIF encode, resulting in a period of time at the beginning and end when only one application is running.

This workload can result in heavy loads on the CPU, requiring CPU frequencies of up to the full 1466 MHz to complete full compression of the two streams within the available CPU time. However, if full compression is performed, the network remains relatively lightly loaded, requiring only 150 Kbytes/s of network bandwidth to achieve drop rates in the range of 0.5% to 1%.

#### 5.7.1.3 Up to three applications

This workload combines up to two 15-fps QCIF sequences with a single 10-fps CIF sequence. For this workload, the CIF "Foreman" sequence is repeated three times and encoded at 10-fps (as in the two-application case). Alongside that, an encode of three repetitions of the QCIF "News" sequence at 15 fps starts 10 s into the 10-fps "Foreman" sequence. Also, QCIF versions of the "Akiyo," "Mobile," and "Tempete" sequences are encoded, 150 s into the workload. Because this workload is relatively complicated and involves several application entries and exits, a timeline has been included as Figure 5.6.

Unlike the one- and two-application workloads, performing all possible compression is capable of overloading the CPU even if the CPU frequency is set at its maximum of 1466 MHz. As a result, it also requires significant amounts of network bandwidth to keep the frame loss rate at an acceptable level. This is partially because the sequence cannot be encoded to its smallest possible size due to the CPU limitations. As a result, at tested network bandwidths below 400 Kbytes/s, a significant number of frames were dropped due to the combined effects of CPU and network overloading.

### 5.7.2 Effect of adaptation under fixed network constraints

To evaluate the performance of the GRACE system, we have run the simulation for each of these workloads under a variety of different network conditions. For each of these workloads and network conditions, we evaluate the energy consumption and drop rate for a "Fixed" system

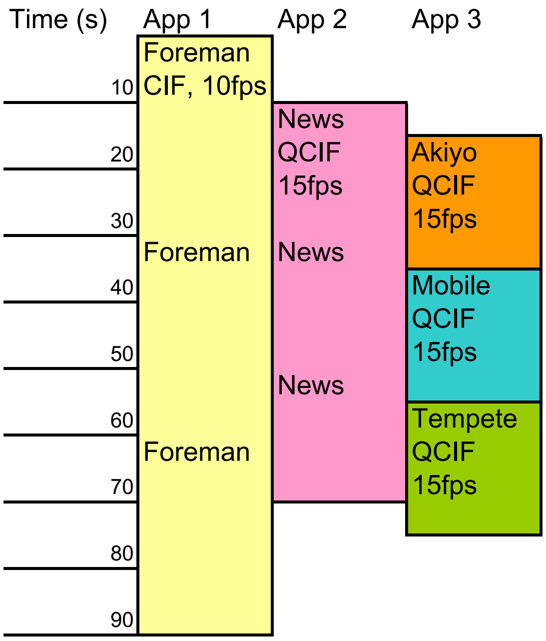
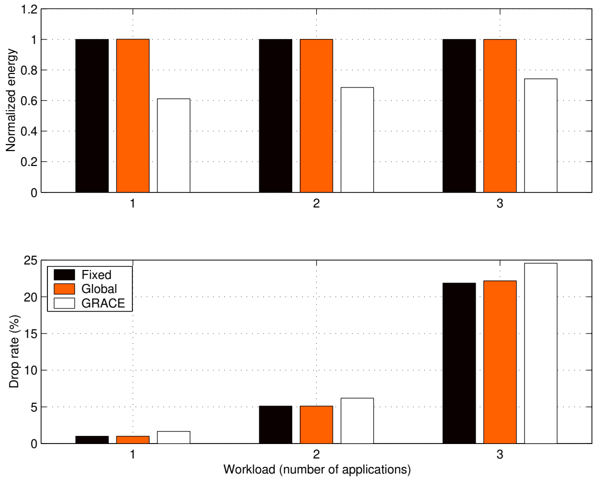



**Figure 5.6** Three-application workload.

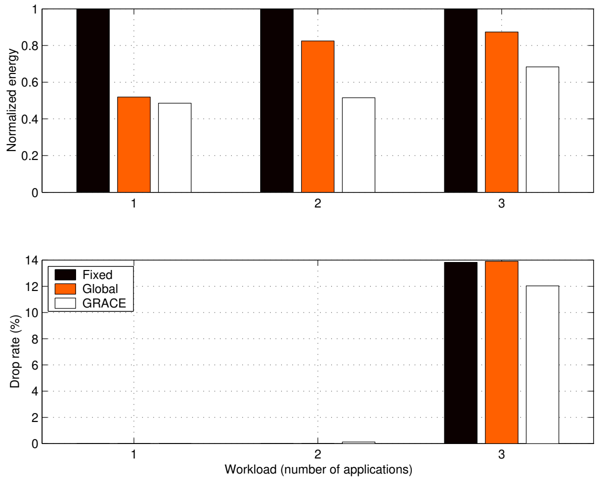
that does not adapt, an adaptive system that does global optimization on application entry and exit ("Global"), and our full "GRACE" system which does global optimization on application entry and exit as well as application adaptation on a per-job basis. Once again, the vertical axes on the energy graphs represent normalized energy, where 1.0 is the energy consumption of the fixed system. On the the drop-rate graphs, the vertical axis shows the drop rate: the percentage of the of total frames (from all applications) that were not successfully transmitted. The horizontal axis is the maximum number of simultaneous applications, representing the one-, two-, and three-application workloads.

We start by considering a heavily bandwidth-limited network capable of transmitting 100 Kbytes/s, the results for which are shown in Figure 5.7 (a). This bandwidth is barely sufficient for reliable transmission of the data for one application, and insufficient for the two- and three-application workload. As a result, we observe drop rates of 1%, 5%, and 22%[2] respectively for the fixed systems. Because of the bandwidth limitation, the global allocator makes the decision to do the full encode and compression in all cases, so the performance of the global-allocation system is the same. However with the addition of the per-application adaptor, we can realize

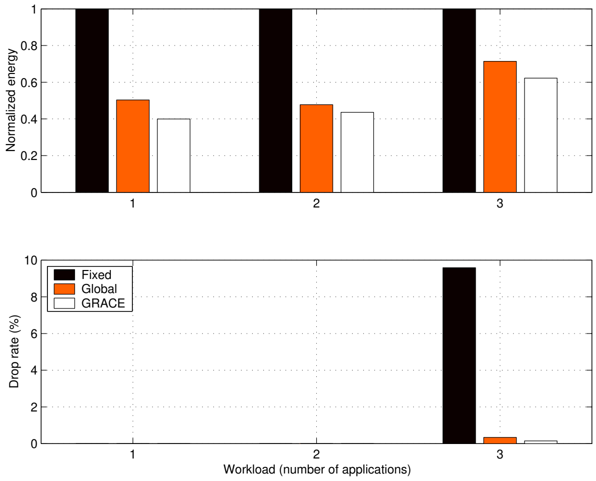
[2]Drop rates of greater than about 5% represents unacceptable stream quality, and as a result comparisons of the associated energy demands are not meaningful.



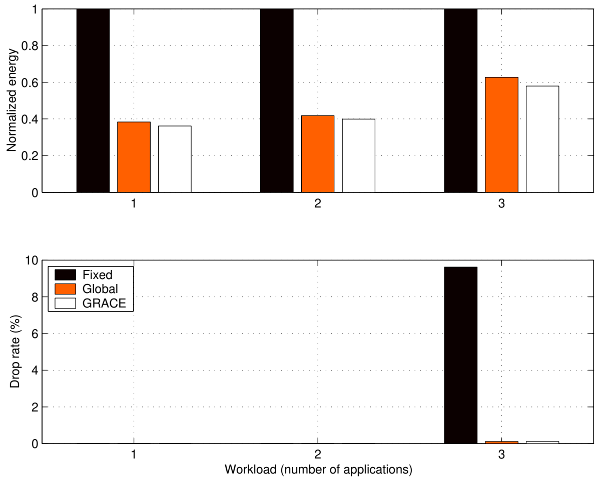

(a) 100 Kbytes/s




(b) 200 Kbytes/s




(c) 400 Kbytes/s




(d) 600 Kbytes/s

**Figure 5.7** Energy savings from GRACE system for fixed-bandwidth networks.

energy savings of up to 39%, although due to the very tight network constraint this is realized at the expense of a slight increase in the number of dropped frames.

As we increase the network bandwidth, we can save more energy through the use of application adaptation, and we drop fewer frames due to bandwidth constraints. However, as we increase the bandwidth, the amount of the total energy benefit that can be achieved with global adaptation alone also increases. At 200 Kbytes/s of network bandwidth (Figure 5.7 (b)), per-application adaptation only saves 3% more energy than global alone for one application. However, for two and three applications, we still see significant energy savings from per-frame adaptation.

At 400 Kbytes/s (Figure 5.7 (c)), we see another benefit of application adaptation. The three-application workload overburdens the CPU when full-complexity encoding is done, but given sufficient bandwidth it is possible to reduce the compression level and prevent frames from dropping due to the CPU overload. We also see a small but noticeable energy savings from the addition of per-application adaptation for all three workloads.

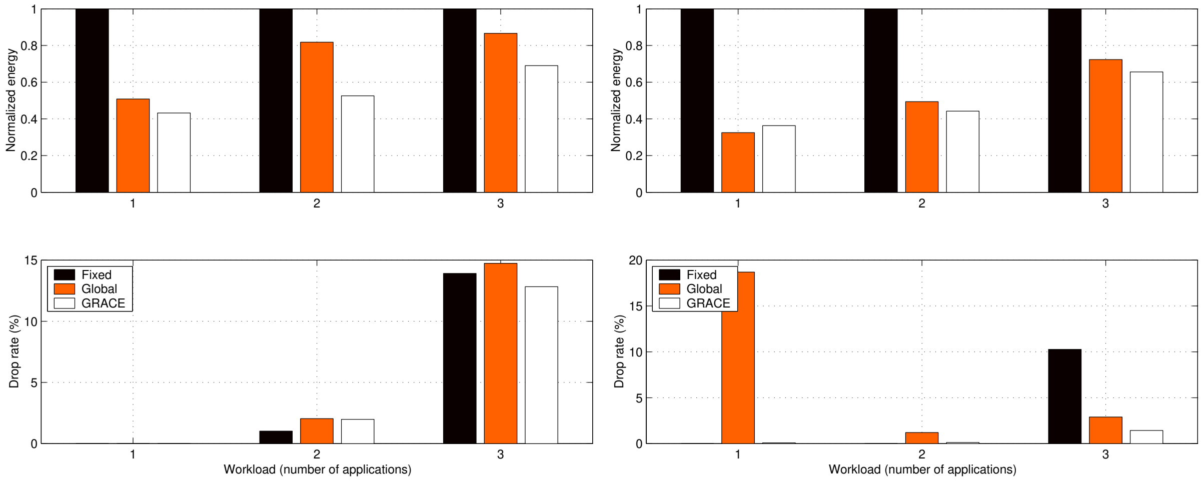
By 600 Kbytes/s—which is approximately the maximum bandwidth that can be achieved on an 802.11b-style wireless network—we see that the bandwidth is high enough that there is little need to use per-frame adaptation (Figure 5.7 (d)). Because the bandwidth constraint is loose, the global allocation can eliminate most of the computational workload even without the ability to make short-term decisions about which application configuration to use.

### 5.7.3 Effect of adaptation under varying network constraints

We next look at the performance of the various adaptive systems under varying network conditions. To do this, we consider networks that vary in bandwidth by more than a factor of three. The variation occurs continuously, varying linearly in a zig-zag pattern and making a complete cycle every 10 s. With the network-bandwidth variation, the effectiveness of the global adaptation is reduced. This results in either a reduction in achievable energy savings, or an increase in the drop rate, depending on the exact relationship between the global bandwidth estimate and the available configurations.

Both of these effects can be seen in our experiments. If we look at bandwidths that vary from 200 Kbytes/s to 600 Kbytes/s (Figure 5.8 (a)), we see that the total energy savings from the GRACE system is about 58% for one application, about 47% for two, and about 30% for three. Moreover, the bulk of the energy savings case for the 2- and 3-application workloads comes from the per-frame adaptation; global allocation alone cannot realize the energy savings we can achieve by taking advantage of the bandwidth available when the channel is good. However, we also see that the bandwidth is insufficient for three applications at times, resulting in a higher drop rate than is desirable.

If we increase the bandwidth to 400-1200 Kbytes/s (Figure 5.8 (b)), the global allocation chooses an application configuration that overruns the available bandwidth when the bandwidth is at the low end of the range, resulting in a similar energy savings for both the global and per-application adaptation but resulting in a high frame drop count for the one-application work-




(a) 200-600 Kbytes/s (b) 400-1200 Kbytes/sec

**Figure 5.8** Energy savings from GRACE system for varying-bandwidth networks.

load. The addition of per-job adaptation to the framework solves this problems. Energy savings here are higher due to the extra bandwidth, ranging from over 60% for the one-application case to approximately 35% for the 3-application case.

## 5.8 Conclusion

In this section, we have shown that the use of application adaptation as part of a multi-level hierarchical adaptation system allows us to achieve significant energy savings. We have also shown that in situations where the CPU is overloaded but the network is not, moving the workload from the CPU to the network can allow an increase in quality of service while at the same time lowering total energy consumption.

### 5.8.1 Analysis: Is per-application adaptation justified?

The main difference between the GRACE system and conventional cross-layer adaptations is the use of the frequent "per-application" adaptation. Most other adaptation systems adapt globally to allocate resources between applications, and permit the individual layers to make short-term decisions about their behavior, but do not have specific support for frequent but limited cross-layer adaptation. We therefore specifically consider the question of whether this additional complexity is justified in the context of our adaptive application.

First, we see that the benefits of application adaptation can be split into two parts: the benefits that can be achieved by correctly matching the use of the network and the CPU against the energy demanded by these system components, and the benefits that can be achieved by matching the application configuration against the network's available bandwidth.

The benefits of correctly matching the use of an *uncongested* network and CPU against their corresponding energy requirements can be realized to a large extent using global adaptation alone. Only in a small range of relative CPU and network weights does per-frame adaptation give benefits over and above global-only adaptation of the application, and even in the best case the additional energy savings that can be achieved from per-frame adaptation of the application in an unconstrained environment is approximately 8% as shown on the left side of Figure 5.5.

However, per-frame adaptation can have significant benefits when it is used to balance bandwidth use against bandwidth availability if the lowest-energy application configuration cannot always be used due to bandwidth constraints. In this case, we can achieve nearly 20% more energy savings than we could with global adaptation alone, as shown on the right side of Figure 5.5. Even when the system is expanded to the full GRACE implementation with multiple applications, we observe that the per-frame adaptation remains most effective when the network is tightly constrained or varying, and when the network constraint is relatively loose global adaptation alone realizes most of the potential energy savings.

Therefore, we can conclude that compared to doing adaptation at a global level only, the additional complexity of the GRACE system is of substantial benefit when the network is constrained—i.e., in the case where the state of the network prevents us from using the application configuration that would give the optimal tradeoff between CPU and network utilization. However, when the network does not constrain this selection, the use of per-frame adaptation to track the state of the network and application offers little additional energy savings over global adaptation alone.

### 5.8.2 Comparison to other GRACE results

Other GRACE work [75] also evaluates the performance of a similar adaptive application and adaptation algorithms in the context of a laptop transmitting data on an 802.11b wireless network. In fact, the core of the adaptive video encoder described in [75] is the video-encoder library described in Chapter 2. Because of the large-scale similarity of the application and environment, this work comes to the same general conclusions as we have here: application adaptation results in significant energy savings when the CPU is relatively more expensive than the network, and it is possible to realize much of this energy gain through the combination of profiling and on-line updates.

However, several details of the actual simulation environment are quite different. Most importantly, the work in [75] does not consider scheduling constraints or the system's frame rate when choosing system or application configurations. Instead of fixing frame rate and varying CPU speed and application configuration to minimize energy consumption consistent with the desired frame rate, the frame rate is allowed to change dynamically to minimize the total energy consumption. As a result, the CPU is allowed to run at its minimum speed at all times, and is assumed to take 3 W in operation. The network is modeled using a fixed rate and an operating power of approximately 1 W, and operates at one of the fixed 802.11b PHY rates of 1, 2, 5.5, or 11 Mbit/s.

Between the lack of a frame-rate constraint and the reduced power consumption of the CPU in this model, the amount of power used by the CPU relative to the network is significantly lower than in the model used here. As a result, the system in [75] yields less energy savings than we show here. Because of these different simulation conditions, the specific results cannot be directly compared to the results in this section.

# CHAPTER 6

# PROBABILISTIC GLOBAL OPTIMIZATION

## 6.1 Introduction

In the previous chapters, we have avoided the question of how to allocate quality by assuming that all applications support only one level of quality, which they must achieve at whatever cost in energy. However, multimedia applications generally support varying quality levels; parameters such as quantization step size, resolution, and frame rate can all be adjusted to effect a varying tradeoff between the quality of the sequence and the resources required to process it.

Ideally, we would be able to allocate the various resources—CPU time, network bandwidth, energy—in a way that best reflects the user's expectations. Allocating CPU time and network bandwidth is relatively straightforward; these resources are not conservable, so it makes sense to use as much of the resource as is needed to maximize the user's utility. However, energy is conservable, and therefore the greedy approach we use for CPU time and network bandwidth is only optimal in very restricted circumstances [2]. This problem has been addressed through calendaring [76], a process which takes a list of tasks and times, and finds a resource schedule that optimizes user's utility. However, this requires that the user know in advance what will be running and when, information that may be unavailable or unduly restricting.

However, there is another approach. The theory of Lagrange optimization [77] provides a framework in which we can approach the allocation problem *stochastically*, allowing us to find an optimal allocation even if we have only a probability distribution of the workloads that are to be run; in other words, we must know only *what* is going to run, and not necessarily *when*. Furthermore, the Lagrangian approach allows us to solve the energy allocation problem without evaluating the cross product of all applications entering and leaving the system.

## 6.2 Utility and Application Models

However, before we can talk about optimizing the system's utility and hence the user's satisfaction with the system performance, we must explore the relationship between the utility and quality metrics that are meaningful to the application.

The details of this relationship are largely beyond the scope of this thesis; a proper exploration of the relationship between the configurations of the encoder, stream quality in a quantifiable sense, and utility in a sense that is meaningful to the user would require careful evaluation with human subjects. With that said, we have to have some model of the relationship between configurations, quality, and utility to do any sort of utility optimization. Because accurate mappings from quality to utility can only be done through human trials, we will develop a flexible model that will support a large class of mappings between quality and utility. This will allow our model to incorporate the results of future human trials, whatever they might be.

### 6.2.1 Utility Model

First, we must establish a model for utility. The model that we use is that utility is additive across applications and additive across time. In other words, if we configure a particular application with a particular configuration, we credit the system with its corresponding utility for as long as that configuration is active. We therefore interpret the utility of the various applications as an "instant utility" that integrates over time for as long as the application is running.

This model ignores both synergies between applications and the possibility that the utility of an application is only realized upon its completion. However, the additive model for utility permits the use of fast algorithms to find optimal long-term global configurations. Without the assumption of additivity across time and applications, we cannot use the fast algorithms we will introduce. Instead, a search across the cross product of all possible applications running at all times would be required to find the optimum application configurations.

### 6.2.2 Application utility

We must next model the application. Each application can run in one or more different configurations. Each of these configurations is corresponds to a particular utility, along with a period and the associated per-period network and CPU requirement.

Different application configurations are not required to have unique periods and utilities. This is because the system is designed to support a hierarchical adaptation in which the lower-level adaptor is provided several alternatives from which to pick. If two application configurations share the same period and utility, they are considered interchangeable by the per-application adaptor, which is permitted to choose from any application configuration that matches the utility and period assigned by the global coordinator. This allows the per-application adaptor to make per-frame decisions about which configuration minimizes energy consumption within the currently available resources.

We also assume that as the utility of application configurations increases, the resource utilization will also increase; while it is possible that different configurations in a utility class will have varying resource demands, no configuration of lesser utility will use more resources of all types than any configuration of greater utility. We also assume that the resource utilization curves are approximately convex, and as we increase the resource usage, the benefit we get from using the additional resource is reduced.

In general, the utility of the application configurations can be assigned in one of two different ways. The first is to apply some function $f(\cdot)$[1] to some measure, such as PSNR, of the stream quality. In this case, the utility will necessarily be probabilistic at allocation time, and therefore we must allocate based on expected utility instead of achieved utility. The second is to simply assign a fixed utility to each application configuration; application configurations giving a higher quality (i.e., better fidelity to the original stream) are assigned higher utility values than configurations that achieve a lower quality. In our experiments, we choose the second approach, assigning increasing utilities to application configurations as the frame rate and number of quantizer steps is increased. Because the precise relationship between quality and utility can only be obtained through extensive psychophysical studies beyond the scope of this work, we have designed the framework to flexibly accept the results of such studies and

[1] The symbol $\cdot$ refers to any applicable function parameters.

in the interim use "placeholder" values for the purpose of evaluating the effectiveness of the framework.

### 6.2.3 Utilization Model

Because our system uses an EDF scheduler internally, we allocate CPU and network resources by attempting to prevent the utilization of each from going above unity. Therefore, when making our global allocation, we use normalized utilization metrics: the resource demand of each application configuration is listed as a number from zero to one, representing the fraction of the total CPU and network bandwidth it uses. The resource demands stated by the application include all associated system overhead, including CPU time used to support the network traffic the application generates and the network traffic used to improve reliability through retransmissions and forward error correction.

In reality, external conditions (such as the effective bit rate of the network and the stream being encoded) and the CPU frequency all vary. As a result, utilizations may also vary, and all of the utilizations we allocate for are actually *expected* utilizations. To account for this, we must actually constrain the *expected* utilization of the CPU and network to $E[\text{utilization}] \leq 1$. In general, this problem would be difficult to solve, as the utilization and availability of the various system components are not independent. Therefore, to keep the problem tractable we make the generally false assumption that the number of bits to encode and the bit rate of the network are independent, allowing us to multiply their expectations.

The expected cycle count and the expected byte count for each configuration are stored in a profile table. The CPU utilization is determined from the cycle count and a CPU frequency assigned to the application; this CPU frequency is considered to be part of the application configuration. For the network, the utilization is determined using an expected or average bit rate for the network layer, along with the expected application bit rate stored in the profile table. In some cases, the network protocol may consume significant CPU resources in addition to network bandwidth. In this case, the CPU utilization of the network protocol must be included when the application indicates its CPU requirements. This allows the optimization to properly account for all demands on the CPU.

## 6.3 Global Allocation Problem

The top-level allocation problem—the problem we are attempting to derive a feasible, useful solution for—is to achieve the best possible user experience by balancing the instant utility of the system against the amount of time the system is expected to run.

We can define this top-level problem as a maximization of the integral of the utility for all running applications until the system battery dies. Applications can come and go, but we assume that the system is free to reject any application at the admission-control stage to save energy, even if resources are otherwise available to run the application. This formulation was introduced by Yuan et al. in [76].

### 6.3.1 Problem definition

Inputs to this optimization problem are an application list and the state of the system. The application lists are derived from the global allocation data associated with each application, and the list of available operating frequencies for the CPU. We represent each application with a unique ID *app*; $freq_{app}$ represents the CPU operating frequency, and $conf_{app}$ is the configuration ID for the application. Each application starts and ends at a particular time, and is permitted to have its configuration changed by the global allocator at any time. When an application is not running, it uses no resources and has no utility. The state of the system is represented by the desired runtime *runtime* and the energy stored in the battery $E_{bat}$. The goal is to maximize the total utility of the system (that is, the integral over the utility for all applications over the desired runtime) given a fixed starting energy.

This problem can be formalized as

$$\max_{confs, freq} \int_t \sum_{apps} U(t, app, conf_{app})\ \mathrm{d}t \tag{6.1}$$

$$\text{s.t.}$$

$$\int_t \sum_{apps} P(t, app, conf_{app}, freq_{app})\mathrm{d}t \leq E_{bat}$$

$$\forall t \sum_{apps} C(t, app, conf_{app}, freq_{app}) \leq 1$$

$$\forall t \sum_{apps} N(t, app, conf_{app}) \leq 1$$

where

- $P(t,\ app,\ conf,\ freq)$: Average power in watts
- $C(t,\ app,\ conf,\ freq)$: Normalized CPU utilization (0 to 1)
- $N(t,\ app,\ conf)$: Normalized network utilization (0 to 1)
- $U(t,\ app,\ conf)$: Average utility (is integrated over time)
- $E_{bat}$: Energy stored in the system battery
- *runtime*: System run time

Because the application is actually time-sliced and scheduled on a per-job basis by an EDF schedule, we actually have to measure utility and power on an average basis rather than an instant basis. To determine the average power, we divide the energy consumed processing one job by the job rate. We define the instant utility to be constant across an entire job (in fact, we assume it is constant unless the application configuration changes, although the model does not require this), and therefore average utility and instant utility will be the same number. It is also important to remember that the utilizations are actually probabilistic, so we actually allocate based on their expected values.

It proved to be easier to state the battery constraint in terms of average power and runtime instead of energy. This can be trivially done by rewriting the energy constraint as

$$\int_t \sum_{apps} \frac{P(t, app, conf_{app}, freq_{app})\mathrm{d}t}{runtime} \leq P_{avg} \tag{6.2}$$

while preserving the associated CPU and network constraints.

### 6.3.2 Dual problem

The optimization problem we have presented optimizes utility given a constraint on energy. However, it may also be relevant to solve the dual problem, in which utility is constrained and energy is optimized. The problem setup is essentially similar, but instead of fixing the energy stored in the battery $E_{bat}$ as an input, we fix the desired utility $U_{req}$.

Making this change allows us to transform the optimization to the dual problem, in which we exchange the constraint on energy with the optimization target utility. However, because *lower* energy is desirable, we must use negative energy $-E(\cdot)$ in place of the utility $U(\cdot)$ in the optimization problem. Likewise, because we want to find a utility *not lower than* the desired utility but our original problem finds an energy *not higher than* the desired energy, we must replace the energy term $E(\cdot)$ in the original problem with negative utility $-U(\cdot)$. We must similarly negate the constraints. Once this transformation is made, the resulting problem takes the same form as the original problem and can therefore be solved using the procedures we develop.

## 6.4 Heuristic Solutions to the Allocation Problem

As is pointed out in [2], solving the top level problem directly requires precise knowledge about what applications will be running at any particular point in time in the future. Typically, this information will not be available. In [76], Yuan and Nahrstedt propose a reservation approach in which future applications are scheduled in advance. However, this construction requires precise information about the future and is therefore somewhat inflexible.

Because of the computational complexity and the aforementioned inflexibility of the reservation approach, various simplifications have been proposed. In [2], Yuan et al. propose two heuristics to handle cases where information about future tasks is unknown. One of these, the *utility-greedy* approach, simply tries to maximize the instant utility at all times, ignoring the impact of the associated energy use on system runtime. It is optimal if the desired runtime is short.

The more common simplification, presented by Yuan as the *energy-greedy* heuristic, is to assume that the applications presently available will run from the present until the time the battery dies. This actually represents a subproblem of the original problem, and therefore the heuristic is optimal if the applications in fact do not change, and is a good heuristic if the character of the applications running on the system stays roughly the same. However, if the utility or energy demands of the applications change dramatically over time, it may result in significantly suboptimal allocations.

This subproblem (and by extension the energy-greedy heuristic) is essentially a constant power approach to the top-level allocation problem; at all times it limits power consumption to a value that allows the required lifetime to be achieved given the current energy supply. (The power constraint can vary in response to current energy availability as the optimization is repeated.)

**Theorem:** Given a sufficiently dense set of application configurations, the energy-greedy heuristic results in a near-constant system power.

**Proof:** To see that the energy-greedy approach attempts to equalize power consumption over time, we can consider its associated optimization problem. The energy-greedy heuristic maximizes the utility of the currently running applications subject to the power constraint

$$P_{avg} \leq \frac{E_{remain}}{T_{remain}}$$

Utility is a monotonically increasing function of power,[2] and we will always choose to use as much power as possible to achieve the greatest possible utility. As a result, the energy consumption of the system will be as close to $P_{avg}$ as possible given the available application configurations. If the set of application configurations is dense, the actual power will be close to $P_{avg}$, and when the maximum allowable power is calculated again the result will be near (but perhaps slightly higher than) $P_{avg}$.

### 6.4.1 Fixed-application optimization problem setup

For this subproblem, instead of maximizing the integral of utility over time for a possibly unknown collection of applications, we consider only the optimization of the utility provided by applications available to the system now. Likewise, instead of considering total energy consumption, we consider only the average power required to run the currently active applications. Once we have solved this problem, we will later return to the full optimization problem and show how we can use solutions to this inner problem to help us find a solution to the complete problem.

The subproblem's inputs are a list of applications and configurations. Each of the application configurations is labeled with a resource utilization and a quality metric. The optimization problem takes this list of application configurations as input, and returns an allocation that chooses appropriate application configurations that meet all constraints on resource availability and average power. This simplified problem can correspond to either of the two policies introduced in [2]. By setting the power constraint to be the energy reserve of the battery divided by desired lifetime, we can realize the *energy-greedy* heuristic. By setting the power constraint to be infinity, we can realize the *utility-greedy* heuristic.

[2]Although this is not true in general, any nonmonotonic points are always suboptimal and will therefore be ignored by the optimization process.

This problem can be mathematically specified as

$$\max_{freqs_i} \max_{confs_i} \sum_{app} U(app, conf_{app}) \tag{6.3}$$

$$\text{s.t.}$$

$$\sum_{app} P(app, conf_{app}, freq_{app}) \leq P_{avg}$$

$$\sum_{app} C(app, conf_{app}, freq_{app}) \leq 1$$

$$\sum_{app} N(app, conf_{app}, freq_{app}) \leq 1$$

In other words, the goal is to maximize the utility, subject to a constraint on total power and network and CPU utilization. This optimization problem takes the form of an NP-hard multidimensional, multiple-choice knapsack problem [73].

### 6.4.2 Subproblem solution and issues

Because the optimization problem reduces to the solution of an NP-hard knapsack problem, it must be solved either with an exponential-time brute-force algorithm, or using a suboptimal approximation algorithm such as the one presented in [73]. However, for problem sets involving reasonable numbers of multimedia applications running simultaneously (up to about two or three), a full search will not need to cover a huge number of possible combinations, and is likely to remain practical.

Another complexity issue that we encounter doing this optimization is that of CPU frequency optimization. In the problem formulation above, we optimize over both the configuration for each application and the CPU frequency that each application runs at. This requires (in general) evaluating the cross product of frequencies and application configurations, increasing the number of possible configurations that need to be evaluated for each application by a factor of the number possible CPU frequencies.

To simplify the problem, we can assume that all applications run at the same CPU frequency. The energy lost to this suboptimality is bounded by Jensen's inequality, and also largely recovered by the per-application adaptations we do on a frame-by-frame basis. Making this assumption permits us to do the entire optimization once for each CPU frequency, rather than making each inner optimization more complex, greatly reducing the computational complexity of the problem.

Jensen's inequality states that for any convex curve $f(x)$:

$$\frac{\sum f(x_i)}{n} \leq f\left(\frac{\sum(x_i)}{n}\right) \tag{6.4}$$

If we assume we can interpolate between any two operating frequencies by running the system at one part of the time and the other for the rest, the energy per cycle is a convex function of CPU frequency.[3] We can therefore apply Jensen's inequality and find that operating all applications at the same (possibly interpolated) frequency is better than running applications at different frequencies.

Therefore, if we restrict the applications to a single frequency, the extra energy consumed is limited by the difference between the chosen frequency and the ideal interpolated frequency. This difference will be less than or equal to the difference in energy consumed when all applications run at the chosen frequency, and the energy that would have been consumed if all applications ran at the next lower available frequency.

## 6.5 Lagrangian Energy-Greedy Heuristic

Traditionally, optimization problems such as the simplified problem described in the previous section are solved using knapsack solvers, or suboptimal, fast heuristics for near-optimal solutions. Lagrange optimization theory provides a different approach based on the replacement of a hard constraint with a Lagrange multiplier $\lambda$ [78,79]. In this section, we show how to apply Lagrangian techniques to solve the instantaneous allocation problem described in the previous section.

### 6.5.1 Lagrangian reformulations

The core idea of the Lagrangian approach to optimization is that solving constrained problems is difficult, and that sometimes this difficulty can be reduced by eliminating the constraints [77]. This is done by rewriting a constrained problem in the form

$$\max_{\bullet} \sum_i A_i(\cdot) \text{ s.t. } \sum_i B_i(\cdot) \leq C \tag{6.5}$$

into the form

$$\min_{\bullet} J(\lambda) = \sum_i -A_i(\cdot) + \lambda B_i(\cdot) \tag{6.6}$$

[3] Any points not on the convex hull would be suboptimal given the interpolation, and therefore not used.

where $\lambda$ represents a particular tradeoff between the term being maximized $A$ and the constrained term $B$. This formulation can in fact be generalized to an arbitrary number of constraints by introducing a separate Lagrange multiplier $\lambda_k$ for each constraint to be eliminated.

Lagrangian optimization theory gives us two important properties for this reformulation. First, it shows that if we solve the minimization problem Equation (6.6), the resulting allocation of the resource represented by $B_i$ to each component $i$ is optimal, if the total availability of the resource is equal to the total amount used; in other words, if $\sum_i B_i(\cdot) = C$, the resource allocation is optimal.

Second, it shows that we are likely to find a value of $\lambda$ that achieves the desired usage of the resource $B$. Specifically, if the utility $A_i(\cdot)$ is a convex function of the resource consumption $B_i(\cdot)$ for every component $i$, there exists some $\lambda$ which will achieve the optimal resource allocation. In fact, there will be a value of $\lambda$ that finds every point on the convex hull of the composite resource-utility curve we get from all possible combinations of allocations. Therefore, as long as the achieved utility is an *approximately* convex function of resource demands, it is likely that a near-optimal solution will be found.

### 6.5.2 The Lagrangian allocation problem

We can apply the Lagrangian technique to our allocation problem by realizing we have a utility function analogous to the $A(\cdot)$ shown in Equation (6.5), and several resource constraint functions analogous to $B(\cdot)$. We can therefore transform this problem into a Lagrange form, and by finding suitable values for the Lagrange multipliers remove the constraints on the optimization, yielding an unconstrained problem that admits fast solutions.

However, for our resource allocation problem it is difficult to use Lagrangian techniques to remove all of the constraints; even if the energy constraint is removed through Lagrange reformulation, the CPU and network constraints still apply. Although the Lagrangian approach can be used even for multidimensional problems, bisection searches with multiple $\lambda$'s have not been shown to be efficient or optimal. Also, the Lagrangian approach can only find points on the convex hull of the operating surface, and the resulting requirement for multidimensional near-convexity[4] is problematic. For this reason, we do not use the Lagrangian technique to remove the CPU and network constraints of the utility optimization problem.

[4] The problem must be close enough to convex that reasonable operating points can be found on the convex hull for any reasonable resource availability.

However, we can still use the Lagrangian formulation to unify utility and energy. To achieve this, the optimization problem stated in the previous section can be reformulated in terms of a Lagrange multiplier as follows:

$$\min_{\lambda} J(\lambda) \tag{6.7}$$

$$\text{where}$$

$$J(\lambda) = \min_{confs,\ freqs} \sum_{apps} \left[ -U(app, conf_{app}) + \lambda P(app, conf_{app}, freq_{app}) \right]$$

$$\text{s.t.}$$

$$\text{CPU constraint: } \sum_{apps} C(app, conf_{app}, freq_{app}) \leq 1$$

$$\text{Network constraint: } \sum_{apps} N(app, conf_{app}) \leq 1$$

An important result from Lagrange optimization theory is that the $\lambda$'s correspond to a slope on the convex hull [79], and that $\lambda$ is actually the *marginal efficiency* between the operating point and the adjacent point on the convex hull that uses less energy and achieves less utility. Therefore, every possible $\lambda$ corresponds to a minimum efficiency requirement, and through Lagrangian optimization theory, to a corresponding optimal configuration of applications. We find the set of application configurations that corresponds to a given value for $\lambda$ by minimizing $J(\lambda)$; the application configurations that minimize $J(\lambda)$ also optimize the utility for the resulting energy consumption. To satisfy a particular power or utility constraint, we must do a search over $\lambda$ to find the value that maximizes utility while obeying the energy constraint corresponding to the capacity of the battery.

### 6.5.3 Optimality of the Lagrange solution

If the power constraint $P_{max}$ happens to "hit" an operating point on the convex hull of the utility-power scatter, an argument analogous to a theorem presented in [77] can be used to prove that the reformulated problem and the original constrained optimizer have the same solution.

**Theorem**: If *conf** is the solution to the problem of Equation (6.7), it is also the solution to the problem of Equation (6.3) for the particular case of $P_{max} = P_{conf*}$.

**Proof**:

$$\begin{aligned}
J(best^*) &\leq J_\lambda(conf^*) \\
-U(best^*) + \lambda P(best^*) &\leq -U(conf^*) + \lambda P(conf^*) \\
-U(best^*) + U(conf^*) &\leq \lambda[P(best^*) - P(conf^*)] \\
-U(best^*) + U(conf^*) &\leq \lambda[P(best^*) - P_{max}]
\end{aligned}$$

Since $best^*$ is a valid solution to the constrained optimization problem,

$$P(best^*) \leq P_{max}$$

And therefore, since $\lambda \geq 0$,

$$-U(best^*) + U(conf^*) \leq 0$$

and therefore $conf^*$ satisfies the original optimization problem. In other words, if we solve the reformulated problem for some $\lambda \geq 0$ and get back a configuration that uses power $P_{\max}$, the solutions of the constrained and unconstrained problems are identical.

It can also be shown that as we sweep across values of $\lambda$, we trace out all the convex hull points on the utility-energy scatter. The fact that Lagrangian techniques find all points on the convex hull is widely cited in the literature [78–80], and was proven in [77]. The proof for the multicell case presented therein clearly applies when only one optimization "domain" is considered.

### 6.5.4 Computational complexity

Normally, we would use the Lagrange formulation to maximize $\lambda$ over a convex curve with no other constraints. If the selection of configurations for applications is independent and permit any combination of application configurations to be used, we can use a bisection search to find an appropriate value for $\lambda$; because $\lambda$ is the marginal utility at the operating point, this implies a configuration for each application. However, in our environment we cannot usefully eliminate all the constraints on the application configurations.[5]

[5] While it is possible to use more than one Lagrangian, we run into two problems doing this. First, we can only find points that are on the multidimensional convex hull; experiments have shown that this eliminates many possible configurations from consideration. Second, bisection searches for $\lambda$ are not known to be efficient or optimal if multiple Lagrange multipliers are used [77].

Because the application constraints are therefore not independent, marginal utilities for each application cannot be found, and therefore $J(\lambda)$ must be calculated by solving the minimization problem directly. The complexity of the search operation is equal to the cross product of all the configurations of all the applications in the system, and each available CPU frequency. However, we can again simplify the problem by setting the CPU frequency of all applications to be equal. By doing so, we reduce the search space to only the cross product of the application configurations, times the number of CPU frequencies, with a minimal increase in energy consumption.

Also, conventional fast-search techniques for solving the multidimensional, multichoice knapsack problem can be applied to estimate $J(\lambda)$ with reasonable results. This is especially valuable when the number of applications is high, as the complexity of a full search is higher and the suboptimality of doing a partial search is reduced.

The search for the value of $\lambda$ that maximizes utility while operating within the energy constraint adds complexity to the optimization problem, and as a result the Lagrange implementation requires more computation than the straight knapsack solution. Because $J(\lambda)$ is a convex function of $\lambda$, the search for $\lambda$ can be done using a fast bisection search that will converge within a small number of iterations [79]; our present implementation searches up to 18 points and finds $\lambda$ to precision of $2 \times 10^{-5}$ times the efficiency of the most efficient application configuration. However, we must optimize the application configurations to compute the minimum value for $J(\lambda)$ for each value of $\lambda$; computation of $J(\lambda)$ is reducible to solving the knapsack problem.[6] As a result, the complexity of the Lagrange search is higher than the computational complexity of a more conventional optimization algorithm. However, use of the Lagrange framework provides other advantages.

### 6.5.5 Interpretation: What is $\lambda$?

Normally, we would use a Lagrangian transformation to convert a constrained optimization problem to an unconstrained one. But due to the CPU and network constraints in our allocation problem, we cannot eliminate all of the constraints using a Lagrange formulation. In fact, reformulating the problem using an efficiency constraint adds additional complexity, as we

[6] It is also possible to search for $\lambda$ indirectly by searching the configuration space for the convex hull point that comes closest to using the available energy, and then solving for the $\lambda$ corresponding to that point.

must now do an outer loop to find an appropriate value of $\lambda$. This means that the knapsack problem must be solved several times, compared to just once for the original problem setup.

So why do the Lagrangian reformulation at all?

The gain is the intermediate parameter $\lambda$. In many cases, although the Lagrangian is a "synthesized" intermediate parameter, it has a real-world meaning. For example, in Frank Kelly's work on network pricing for elastic traffic [81], the Lagrangian values $\lambda_s$ represent the marginal or 'shadow' price of a unit of traffic on the corresponding network link. And in [82], the selected value for $\lambda$ represents the tradeoff between the energy consumed by an equalizer filter tap, and the amount of interference the filter tap can remove from the signal being received.

In the allocation problem we address here, the intermediate parameter $\lambda$ defines a tradeoff between the two optimization targets it connects—in this case, between utility and energy. A high $\lambda$ means that energy is at a premium, and that we should only use a configuration if it offers a particularly high utility in exchange for its energy consumption. A low $\lambda$, on the other hand, means that power can be spent relatively freely in exchange for modest amounts of utility. In fact, due to the construction of $J(\lambda)$, $\lambda$ is actually the minimum allowable slope between the selected point and the previous point on the utility-energy convex hull. This can be easily shown:

**Lemma**: $\lambda$ is the minimum permissible marginal utility for the set of application configurations that minimizes $J(\lambda)$.

**Proof**:

$$J(\lambda) = -U_{best} + P_{best}$$

where $U_{best}$ and $P_{best}$ correspond to the total utility and power consumed by the configuration minimizing $J(\lambda)$.

Because these values minimize $J(\lambda)$, perturbation can only increase $J(\lambda)$. Therefore,

$$\begin{aligned} J(\lambda) &\leq -(U - \delta slope) + \lambda(P - \delta) \\ J(\lambda) &\leq J(\lambda) + \delta slope - \lambda\delta \end{aligned}$$

where *slope* is the slope of the convex hull between the optimal point and the next point to the left, which uses less energy and generates less utility, and $\delta$ is the change in power going to that point. So

$$
\begin{aligned}
0 &\leq \delta slope - \lambda\delta \\
\lambda\delta &\leq \delta slope \\
\lambda &\leq slope
\end{aligned}
$$

Even with the constraints, we can achieve any particular tradeoff between utility and power by only considering system configurations that have *marginal efficiencies*—the change in utility over the change compared to the next lower-utility, lower-energy point on the convex hull—greater than or equal to a fixed number $\lambda$. Furthermore, once we fix a value for $\lambda$, we can continue to use it even if the applications running on the system change. The system will continue to run with the same tradeoff between utility and energy, which means that if similar applications replace the currently running applications they will achieve a similar total runtime and utility. Moreover, if we replace the applications with new ones that offer more utility for energy spent, energy consumption will increase to take advantage of the better opportunities to gain utility for the user; likewise, if new applications are less efficient, energy use will be reduced to conserve energy for the future. The marginal efficiency metric $\lambda$ therefore provides a mechanism which permits the actual power consumption of the system to vary in response to the changing workloads.

Because our constant as the workload changes is the efficiency metric $\lambda$ rather than power, energy, or utility, the system's power consumption can increase at one time to take advantage of the availability of high-efficiency tasks, and decrease at others if no high-efficiency tasks are available.

### 6.5.6 Optimality properties

It is important that although our restated optimization problem remains an NP-hard knapsack problem, it shares important optimality properties with the Lagrangian approach. These optimality properties provide us with a method of solving the full optimization problem in Equation (6.1), which we will introduce in a later section.

As proven in the theorem above, for any value of $\lambda$ the returned solution is optimal in that no other solution has both a larger utility, and a smaller total energy consumption.

The most powerful optimality property, however, is inherited from the Lagrange optimization theory. This is that a fixed $\lambda$ will correctly allocate energy to different applications, even as the workload changes. As long as the marginal utility remains constant, the allocation of energy to the various applications running at different times will achieve the optimal utility for the energy spent. We can therefore use the Lagrange optimization to optimize over changing application loads both in the context of known future tasks (much as is done in [76]), and in the case when task sets for the future are known only statistically. The choice of configurations for any particular set of applications is determined only by the CPU and network demands of the current application set, and a globally-precomputed value for $\lambda$. This fact leads us to a solution to the top-level problem in Equation (6.1), and will be explored in detail in the next two sections.

## 6.6 Optimization for Known Workloads

Although doing Lagrange allocation — that is, the use of marginal utility to pick an appropriate operating point — is serviceable for optimizing the system for running a fixed set of applications, it is not ideal. It optimizes over the convex hull of the utility/energy curve rather than the full scatter, and therefore may leave some portion of the energy unused, achieving a lower utility than otherwise could be be realized.

However, we have considered only the case where the same applications are running on the system from the initial start until the battery is exhausted. Normally, system workload varies as the user moves from one task to another. The true value of the Lagrange allocation method comes from its ability to handle cases where the workload varies over time.

In fact, if we use any fixed Lagrange multiplier $\lambda$ when we allocate utility and energy to the applications running on the system, the resulting system configurations will be optimal in that they will achieve the maximum possible utility for the amount of energy consumed. Also, as we accumulate more different workloads, the extra utility we can achieve by using operating points not on the utility-energy convex hull diminishes.

We can solve the varying-workload optimization problem using a straightforward extension to the Lagrangian solution to the fixed-workload case. Each time an application enters or leaves the system, we create a new "cell" in the Lagrange construction [77], that can be optimized independently given a fixed $\lambda$. Because each "cell" can be optimized independently, we do not need to consider the solutions for previous or future workloads when finding an optimal set of configurations for the current tasks.

### 6.6.1 Lagrange construction for allocation across workloads

Because workloads are nonoverlapping, the top-level problem setup in Equation (6.1) can be converted to to a Lagrangian optimization problem. This is because although each workload is subjected to CPU and network constraints that are not subsumed into the Lagrangian optimization, each workload is active for a fixed time period and its effects on CPU and network utilization do not extend past that time period. The only constraint that affects all workloads simultaneously is the energy constraint, which has been subsumed into the Lagrangian multiplier $\lambda$.

Each time an application enters or leaves the system, a workload is created, We can treat this new workload as an additional Lagrangian "cell" [77]. This "cell" can be optimized independently from the other cells using a precomputed value for $\lambda$, and we do not need to consider the solutions for previous or future workloads when finding the optimal set of configurations for the current tasks.

The Lagrangian minimization problem is formalized as follows:

$$\begin{aligned}
J(\lambda) &= \min_{confs, freqs} \int_t \sum_{apps} U(t, app, conf_{app}) + \lambda P(t, app, conf_{app}, freq_{app}) \\
&= \int_t \min_{confs_t,\, freq_t} \sum_{apps_t} U(t, app, conf_{app}) + \lambda P(t, app, conf_{app}, freq_{app}) \\
&= \sum_i time_i \times \min_{confs_i,\, freqs_i} \sum_{apps_i} U(app, conf_{app}) + \lambda P(app, conf_{app}, freq_{app}) \\
&= \sum_i time_i \times J_i(\lambda)
\end{aligned}$$

where $J_i(\lambda)$ is the Lagrange problem corresponding to the $i$th distinct workload. Each workload consists of a list of running applications $apps_i$, a list of corresponding application configurations $confs_i$, and operating frequencies $freqs_i$. $time_i$ is the length of time that the workload is active.

Because the time intervals corresponding to the different sets of applications being summed over are nonoverlapping, we can split the constraints on CPU and network into corresponding nonoverlapping intervals, each of which is optimized separately. The CPU and network constraints are therefore subsumed into the calculation of $J_i(\lambda)$, and there is no need to explicitly consider these constraints when searching for $\lambda$.

As a result, we can optimize the top-level problem, to within a convex-hull approximation, by doing a Lagrange search over $\lambda$ at the top level, while doing full searches to find $J_i(\lambda)$ for each set of applications independently. There is no need to search the cross product of applications running at different times.

### 6.6.2 Optimality properties

Because this remains a Lagrange formulation, we inherit all of the associated optimality properties [77]. Most importantly, for any value of $\lambda$, the amount of utility we get for the associated energy consumption is the maximum possible, and we can find any point on the convex hull of the composite utility-distortion curve. We can also still use a fast bisection search over $\lambda$ to find an appropriate value [79].

In other words, the Lagrange formulation permits us to reduce the complexity of the top-level allocation problem from a cross-product of all application configurations to a small number of allocations for each set of applications. Even if we are optimizing for a particular runtime, the resulting solution will be optimal to within a convex-hull approximation.

Furthermore, if we consider total (integrated over time) utility and we permit the system to achieve additional utility by slightly extending our runtime from the original goal, choosing convex hull points on the utility-energy curve is optimal as long as the time slices are short enough to be negligible compared to the system runtime. This directly follows from the optimality of the Lagrange (convex hull) solution for any runtime it finds.

**Theorem**: Total utility (integrated over time) from a point on the utility-energy convex hull is higher than the net utility from a point off the convex hull that provides the same or greater utility.

**Proof**: If we consider a point on the convex hull, and another point that provides more utility and is not on the convex hull, the efficiency (utility per unit energy) of the point on the convex hull will be greater than the efficiency of the point not on the convex hull. (Otherwise, the point not on the convex hull would also be on the convex hull, a contradiction.)

Therefore, as long as we can use any remaining energy to increase run time and achieve additional integrated utility, we will achieve more utility from the additional time than we would have by using the additional energy earlier.

These optimality properties lead to a surprising conclusion. If we allow the runtime of the system to extend indefinitely, the problem definition breaks down. Because there is no penalty for waiting, the optimizer will reject any applications that do not offer a very large utility return for energy consumed. These very efficient applications may appear only rarely, resulting in the system spending most of its time doing nothing at all. While this behavior is mathematically optimal, in practice it may not be considered desirable by a user.

## 6.7 Stochastic Problem and Solution

Another benefit of using the Lagrange formulation to solve the resource-allocation problem is that the optimality of the Lagrange formulation is retained, even if the future workload is known only probabilistically. Although in the stochastic case we can recompute utility and runtime only probabilistically, the Lagrange formulation assures that for whatever runtime we realize, the maximum possible average utility is achieved, and equivalently that for whatever average utility is achieved, then we realize the maximum possible runtime.

### 6.7.1 Stochastic problem statement

We introduce the stochastic problem as a variant of the top-level problem in which the workloads are not known in advance. Instead of having a list of workloads and the time periods that they are active, we have a list of potential or representative workloads, and probabilities that they are active at any given time instant.

We define the term $Pr(i)$ to be the probability of workload $i$—that is, a combination of applications we index using $i$—being active at any given time instant. In other words, it is the probability the system is running that particular *combination* of applications. We further assume independence of applications running at different times. While this assumption is patently false, if battery lifetime is long enough, the actual distribution of applications running during the lifetime of the battery will be close to the a priori probabilities. $P_{avg}$ is computed as

$$P_{avg} = \frac{E_{tot}}{runtime}$$

where $E_{tot}$ is the capacity of the system battery and *runtime* is the desired expected runtime.

With these assumptions, the stochastic problem can then be stated as

$$\max_{confs_i,\, freq_i} \sum_i Pr(i) \sum_{apps_i} U(app, conf_{i,app}) \tag{6.8}$$

$$\text{s.t.}$$

$$\sum_i Pr(i) \sum_{apps_i} P(app, conf_{i,app}, freq_{i,app}) \leq P_{avg}$$

$$\forall i, \sum_{apps_i} C(app, conf_{i,app}, freq_{i,app}) \leq 1$$

$$\forall i, \sum_{apps_i} N(app, conf_{i,app}) \leq 1$$

For the stochastic version of the allocation problem, we have simply replaced the integrals over time with summation over workloads and their associated probabilities. Note that if the distribution of future workloads is known exactly, this form is equivalent to the integrals over time in Equation (6.1). Instead of optimizing utility given a runtime constraint, we instead optimize expected utility given a constraint on expected average power. The use of expectation for both utility and power is somewhat flexible; especially on the power side it may be useful to optimize for, say, worst-case or 90th percentile runtime.

### 6.7.2 Stochastic problem solution

Since the stochastic problem setup shown in Equation (6.8) is only trivially different from the deterministic problem introduced in Equation (6.1), we can apply the same Lagrangian techniques we used in the previous section to solve the stochastic problem.

We convert the problem to Lagrange form by writing Equation (6.8) in terms of a Lagrange multiplier $\lambda$. We do this by replacing the optimality criterion with the usual Lagrange formulation:

$$J(\lambda) = \min_{confs,\, freqs} \sum_i Pr(i) \tag{6.9}$$

$$\sum_{apps_i} -U(app, conf_{i,app}) + \lambda P(app, conf_{i,app}, freq_{i,app})$$

$$\text{s.t.}$$

$$\forall i, \sum_{apps_i} C(app, conf_{i,app}, freq_{i,app}) \leq 1$$

$$\forall i, \sum_{apps_i} N(app, conf_{i,app}) \leq 1$$

By interchanging the summation and the minimization, we can rewrite this in terms of the single-application Lagrange weight:

$$\begin{aligned}
J(\lambda) &= \sum_i Pr(i) \min_{confs,\ freqs} \Bigg[ \sum_{apps_i} -U(app, conf_{i,app}) + \\
&\qquad \lambda P(app, conf_{i,app}, freq_{i,app}) \Bigg] \\
&= \sum_i Pr(i)\ J_i(\lambda)
\end{aligned}$$

To find the value of $\lambda$ that maximizes the expected utility (to within a convex-hull approximation) while ensuring that the expected running time of the system is at least some fixed value, we simply do a search over $\lambda$ to find the value that minimizes $J(\lambda)$. Because $J(\lambda)$ is expressed in terms of $J_i(\lambda)$, this search does not require evaluating cross products of different workloads; each workload is only optimized once per value of $\lambda$ checked.

The probability distribution of the workload is only used to estimate the average system power and hence the runtime; it is not used to determine the configuration used at any particular time. This limits the effect of inaccuracies in workload probability estimates. Although an inaccurate estimate will result in a runtime longer or shorter than desired, the system will still run efficiently. Hence, it is possible to design the system to allow the user to control $\lambda$ more directly, possibly providing an estimate of the resulting runtime (for either the current workload or estimated future workloads) for any input value for $\lambda$.

The optimal value of $\lambda$ depends only on the probability distribution of workloads that *may* run on the system; it does not depend on what applications are running at any particular time. Therefore, once an optimal $\lambda$ is chosen, it can be used for a long time—until the desired runtime changes or the battery is replaced or charged, or until the probability distribution that was used to compute $\lambda$ is no longer valid. Even as the workload changes, the value of $\lambda$ we use to compute the optimal allocation of resources for any given workload stays the same, as it represents the optimal division of energy between the current workload and the future.

### 6.7.3 Optimization with a base power load

So far, we have considered optimization only in the case where the applications are the only drain on the battery. However, in a mobile environment there will generally be some base load associated with system components, such as the display, that remain active even if no applications are running. This base load affects our optimization algorithm significantly, because increasing the runtime decreases the total energy availability by the corresponding base load.

The solution to the desired-runtime problem is the same: compute the energy availability after subtracting the base load for the desired runtime, and then allocate for the correct average power across the distribution of possible workloads.

However, the addition of a base load makes another optimization problem interesting: maximize the integral of utility over the run time of the system. This problem without a base load is trivial: operate the most efficient application in any possible workload at its most efficient (and hence lowest-utility) configuration, and disable all other applications. With a base load, it is no longer necessarily the case that the lowest-utility configuration is optimal, because when the system is operated at a low-power, low-utility configuration, the base load is likely to be the dominant power drain. In this case, the benefit of conserving energy for later use is diminished, and using energy more rapidly is justified.

This new problem can be stated as

$$\max_{confs_i, freqs_i} \left[ \sum_i Pr(i) \sum_{apps_i} U(app, conf_{i,app}) \right] \times runtime \tag{6.10}$$

$$\text{s.t.}$$

$$runtime \leq \frac{E_{tot}}{P_{avg}}$$

$$\forall i, \sum_{apps_i} C(app, conf_{i,app}, freq_{i,app}) \leq 1$$

$$\forall i, \sum_{apps_i} N(app, conf_{i,app}) \leq 1$$

where

$$P_{avg} = P_{base} + \sum_i Pr(i) \sum_{apps_i} P(app, conf_{i,app}, freq_{i,app})$$

This problem can also be solved using the Lagrangian formulation shown in Equation (6.9) by searching for an appropriate value of $\lambda$. In this case, we must compute the average power consumed for the value of $\lambda$ being evaluated, add the base load, and compute the lifetime of the system given the battery energy. We can then multiply the lifetime by the expected utility of the system to estimate the total utility. Since the total utility is a product of an increasing function of $\lambda$ (expected utility) and a decreasing function of $\lambda$ (expected runtime), it is convex, and we can optimize the resulting value using a bisection search on $\lambda$. The resulting solution will be the closest convex hull point to the optimal solution; however, the proof that the convex hull point is actually optimal no longer applies.

Although this stochastic allocation approach can be used to choose an appropriate value of $\lambda$ for a known list of workloads and expected runtime, it cannot be used for prescheduled task lists if the system must maximize utility in the presence of a base power load. The running time in this case is variable, and the workload distribution changes depending on the total run time of the system. Such varying workload distributions do not fit the fixed probability model we require for the stochastic algorithm.

### 6.7.4 Network variation

The Lagrangian approach can be easily extended to encompass variation of the network. This is done by adding the network state to the description of each workload when searching for an appropriate value of $\lambda$. In other words, we list the cross product of network states and workloads, setting the probability for each to be the probability of the workload times the probability of the network state. This effectively expands the number of workloads evaluated by a factor of the number of possible network states. Since the network state can generally be quantized into a small number of possible bandwidths, this extended problem will generally remain tractable.

## 6.8 Simulation Setup

We use the simulator developed in the previous section to evaluate the effectiveness of the Lagrangian resource allocation. Although the Lagrangian theory provides mechanisms whereby we can respond to variations in the network, we assume a fixed network that provides a constant bandwidth and per-byte energy demand. This is done to restrict this discussion to the performance of the Lagrangian allocation algorithm itself and not the nature and effects of the network variations.

### 6.8.1 Simulation environment

The network is modeled as having a bandwidth of 500 Kbyte/s and a per-byte energy cost of 1 $\mu$J. The CPU uses the same model of the Athlon XP-M 1700+ with voltage and frequency scaling used in previous sections. The desired runtime is set to 600 s, and the starting energy of the battery is varied to simulate environments under tighter and looser power constraints. The simulation runs for 600 s or until the initial energy supply is exhausted, whichever comes first. Parasitic power demands (such as the display) are not considered; it is assumed that the provided initial energy excludes any parasitic power that would be consumed during the requested running time.

The simulation environment does not presently charge the Lagrange optimization for the processing time and energy spent doing its one-time search for the Lagrange multiplier. The run time for the current implementation of the Lagrange multiplier search is approximately 2 s at full processor speed for the "realistic" workload, so it would increase the total energy consumption for the Lagrangian case by about 50 J. We do not charge this energy because in practice it would be amortized over a much longer runtime than the 600 s used in these simulations.

#### 6.8.1.1 Applications

For these simulations, we re-use the adaptive encoder application described in Chapter 2. The application is run on input streams with two different image sizes, CIF (352 x 288) and QCIF (176 x 144). For each image size, the resource requirements can be reduced at the cost of decreasing utility by decreasing the frame rate from the base of 10 (CIF) or 15 (QCIF) fps. The system also supports reducing the quality by increasing the quantizer step size for CIF encoding, although these configurations are relatively inefficient (in terms of utility per unit power consumed) and are therefore not selected by the optimizer.

Each operating mode allows application adaptation: 15 available compression modes when Q = 6, and 6 when Q = 12.

**Table 6.1** Application base utilities

| Resolution | Frame rate | Quantizer step size | Utility per second |
|---|---|---|---|
| CIF (352 x 288) | 10 fps | Q = 6 | 1.00 |
| | | Q = 12 | 0.80 |
| | 5 fps | Q = 6 | 0.60 |
| | | Q = 12 | 0.50 |
| | 3.3 fps | Q = 6 | 0.20 |
| | | Q = 12 | 0.15 |
| QCIF (176 x 144) | 15 fps | Q = 6 | 0.50 |
| | 10 fps | Q = 6 | 0.30 |
| | 5 fps | Q = 6 | 0.10 |

The base utilities for every possible configuration of the encoder are shown in Table 6.1. These numbers are expressed as a *rate*, in terms of utility per second. Each second that the application is running and set to a given configuration, it accumulates the utility shown in the table.

Because choosing meaningful values for the base utility would require extensive human trials, values were instead were assigned by hand. These particular values for utility were selected to ensure that the utility is a monotonic function of resource utilization and hence energy. They do not result in a convex energy/utility curve; this is intentional and intended to put the Lagrangian approach at a slight disadvantage.

In addition to the base utility, which is associated with the application itself, each time an application starts it is assigned a "weight" by the user. The weight connects the base utility of the application with the user's perception of its importance—it is a "utility mapping function." The implementation multiplies the weight assigned by the workload by the base utility rate of the application to find the actual utility rate for each potential application configuration. The higher the weight, the higher the resulting utility, and the more likely it will be that the application will be allocated enough energy, CPU time, and network bandwidth to operate at a high quality level.

### 6.8.2 Simulation workloads

We implement these simulations by defining two different prototype workloads, consisting of the CIF and QCIF versions of our adaptive encoder application. The first "realistic" workload is intended to represent a reasonable variation in desired applications and utility; the second is a "best case" workload intended to highlight the improvements in total utility that can come from allocating energy only to the most beneficial applications.

The prototype workloads list the possible application sets, the weight for each application, and the probability that this application set is active. We then generate the actual workload by choosing a workload from the prototype according to the associated probability distribution for each 30-second slice of a 600-second simulation run. The input stream is a composite of several MPEG test sequence, treated as a circular array. As part of the workload creation process a starting position for each application invocation is chosen randomly (with a uniform distribution) from the frames in this composite stream.

In all cases, the original probability distribution from which the actual workloads are drawn is used along with composite statistics about the application's resource demands to compute the value for $\lambda$ used for the Lagrangian optimization.

It is important to note that the global allocator is permitted to refuse any offered jobs, and that each job can run at one of several different quality/utility levels. This means that the actual energy consumption of an offered workload can vary down to zero, if none of the offered applications receives an energy allocation.

#### 6.8.2.1 "Realistic" workload

The "realistic" workload (Table 6.2) is intended to represent things a user could plausibly do with the computer. As we are limited by the fact that our adaptive application is an encoder, it is not particularly "realistic" in practice. However, unlike the "advantageous" workload it has not been designed to provide the Lagrangian optimization approach with a large advantage. We therefore expect the utility improvement we achieve with this workload to be more representative of the general case.

These workload probability distributions are synthetic and created to illustrate a point. However, one possible explanation for this type of workload is a remotely operated wireless camera controller (controlling several cameras) covering some type of event. CIF encoders (which consistently have higher priorities) could represent cameras being used for a live feed; QCIF encoders could represent cameras that are being previewed at the mixing board but are

**Table 6.2** "Realistic" workload

| Probability | Image size | Weight |
|---|---|---|
| 20% | CIF | 1.5 |
| 20% | QCIF | 0.8 |
| | CIF | 1.0 |
| 25% | CIF | 1.3 |
| | QCIF | 1.0 |
| 25% | QCIF | 1.0 |
| | QCIF | 0.5 |
| | QCIF | 0.5 |
| 10% | CIF | 1.0 |
| | QCIF | 0.7 |
| | QCIF | 0.7 |

not currently active. As cameras are turned on and off and sent to the feed, the workload changes.

#### 6.8.2.2 "Advantageous" workload

The "advantageous" workload (Table 6.3), on the other hand, represents a situation under which the Lagrangian optimization makes a large difference in the total utility of the system. It does not represent an upper bound (as the utility improvement given a suitably constructed workload availability is unbounded). It is instead intended to show that under certain circumstances, large utility improvements can be achieved.

**Table 6.3** "Advantageous" workload

| Probability | Image size | Weight |
|---|---|---|
| 20% | CIF | 100.0 |
| | CIF | 100.0 |
| 80% | CIF | 1.0 |
| | CIF | 1.0 |

This type of workload distribution could be found in a sensor network. The rare, high-value operations occur when the sensor has detected something of interest and the operator is likely to be actively viewing the sensor's output; the common, low-value operations occur when the system has not detected anything of interest and therefore is unlikely to be needed or monitored.

## 6.9 Simulation Results

We evaluate the performance of the Lagrange optimizer against the "Energy-greedy" heuristic described by Yuan et al. [2].

Figures 6.1 and 6.2 show the results of a proof-of-concept simulation of the Lagrangian allocator. Each set of graphs includes five rows of three graphs. The first four rows represent the same sequence of workloads; each workload sequence consists of a list of workloads, drawn randomly from the "realistic" or "advantageous" probability distributions of applications. New workloads are drawn for every 30-second slice, so there are 20 different workloads total represented in each graph. However, the system may shut down early and not run the last several workloads. The last row of graphs shows the average results across 10 realizations of the workload sequences, including the four shown as Workloads 1 through 4.

The leftmost column of graphs shows the total realized utility—in other words, the sum of the utility values multiplied by the running time and the weight of each application—over the 600 s the system is allowed to run. The middle column shows the amount of time that the system runs before it shuts down, either due to running out of time or exhausting its energy. The rightmost column shows the total energy consumption of the system. None of these totals include the time and energy that would be spent finding the optimal value of $\lambda$ as it is assumed to have been computed off-line. The overhead of allocating resources to each application entering and leaving the system is, however, included.

Each graph has a solid, darker line representing the results for the Lagrangian optimization and a dashed, lighter line representing the "energy-greedy" heuristic. The horizontal axis on all the graphs is the starting energy of the battery, expressed in terms of the average power permitted over the 600-second desired runtime; the starting energy in Joules is the value in watts shown times 600. The vertical axis on the "utility" graphs is utility units based on the application utility and weightings; on the "time" graphs it is seconds, and on the "energy" graphs it is once again in terms of power averaged over the desired runtime of 600 s. The performance is sampled across average power limits at every two watts from 5 to 25 W.

The minimum and maximum values across all 10 workload sequences is shown as error bars on the "average" graphs. The darker error bars correspond to the Lagrangian optimizer, and the lighter error bars correspond to the energy-greedy heuristic. Note that the minimum and

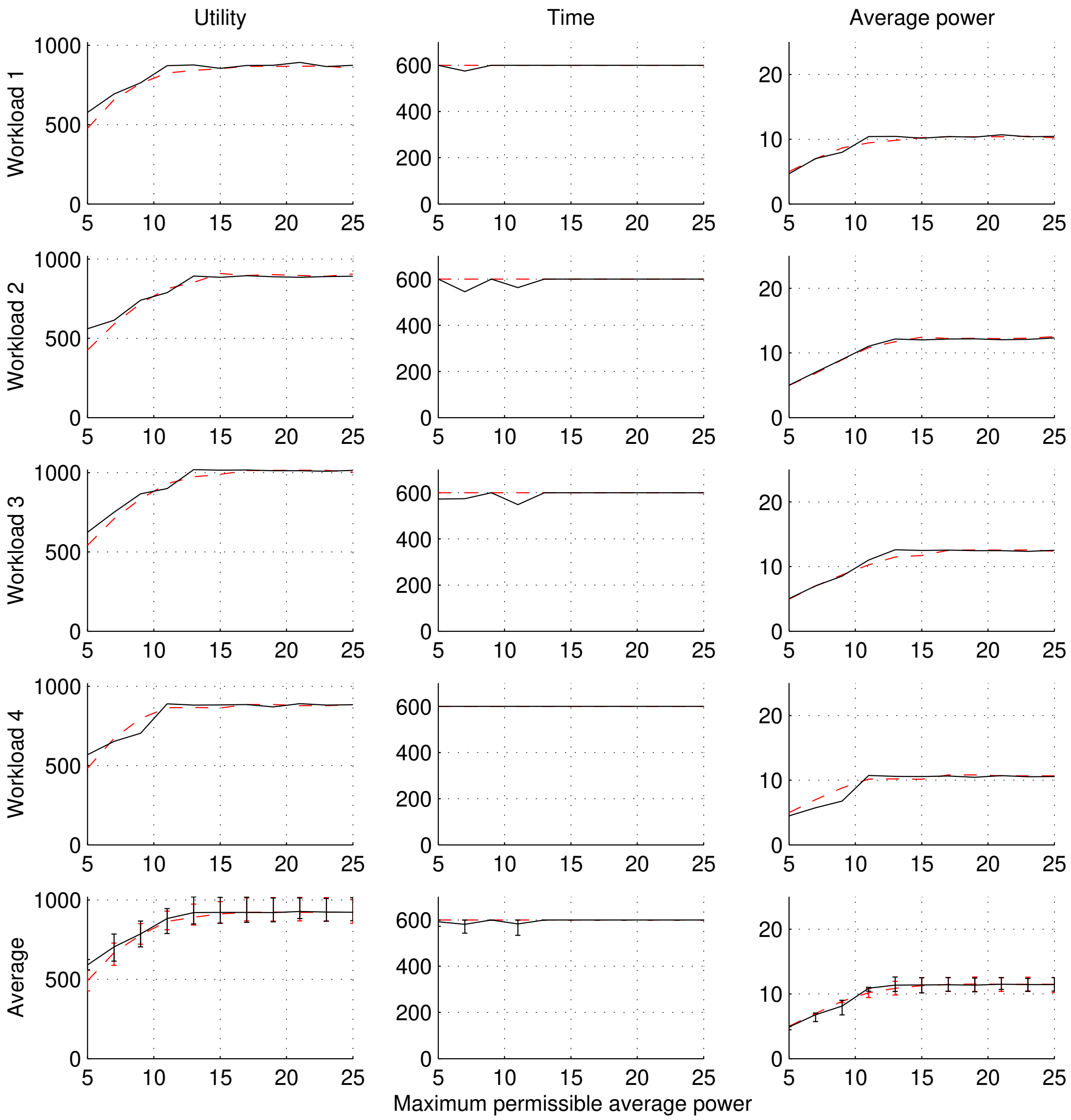



**Figure 6.1** "Realistic" workload. The "Workload" graphs show specific results for four workload sequences; all points on the graph for each row come from the same sequence of applications entering and exiting. The "Average" graphs show average and min/max (indicated by error bars) results across 10 workloads. Dashed/lighter lines are from the energy-greedy heuristic, solid/dark lines are from the Lagrange optimizer. Left column shows the total (summed) utility, middle column shows running time in seconds (limited to 600 s), right column shows average power in watts.

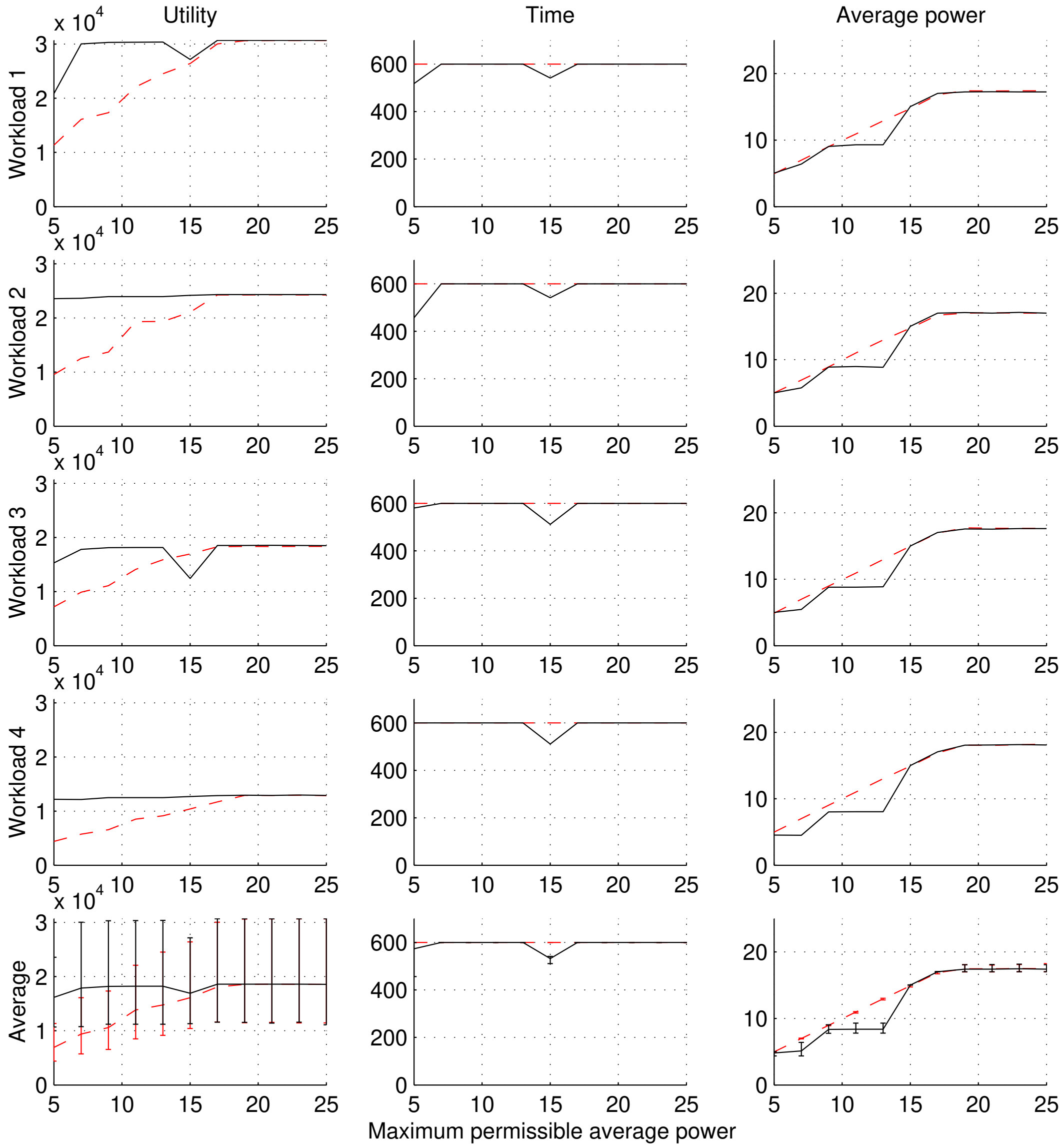



**Figure 6.2** "Advantageous" workload. The "Workload" graphs show specific results for four workload sequences; all points on the graph for each row come from the same sequence of applications entering and exiting. The "Average" graphs show average and min/max (indicated by error bars) results across 10 workloads. Dashed/lighter lines are from the energy-greedy heuristic, solid/dark lines are from the Lagrange optimizer. Left column shows the total (summed) utility, middle column shows running time in seconds (limited to 600 s), right column shows average power in watts.

maximum values for each starting energy are each selected independently, and do not represent any single workload.

### 6.9.1 "Realistic" workload

The results for the "realistic" workload are shown in Figure 6.1. We see that, on average, there is a significant increase in utility when the starting power is low, and no significant change in utility or energy consumption when the starting power is high. At an average power constraint of 5 W, we improve the average achieved utility by over 20% by pushing the power consumption from times that only low-utility tasks are running to other times when higher-utility tasks are available. However, once the starting energy is sufficient to allow the average power drain to exceed 15 W, there is no benefit from the use of the Lagrangian approach.

It can also be seen that in some cases, the total utility is reduced modestly. The worst loss of utility observed is 12%, and occurs due to under-using energy; this case is shown as "Workload 3." Here, at an average power limit of 9 W, only 77% of the original energy is used at the end of the end of the 600-second desired runtime. This occurs when the actual application selections have a lower utility than the prototype distribution. Although the applications to be run are selected from the probability distribution provided to the code that computes the optimal Lagrange multiplier, the workload list is short enough that significant variations from the mean distribution can occur.

### 6.9.2 "Advantageous" workload

The results for the "advantageous" workload are shown in Figure 6.2. Because this workload was constructed to show a large benefit from the Lagrangian optimization, we see an average improvement in utility of over 140% when the average power is limited to 5 W. (It is important to remember that a suitably constructed sequence could realize an arbitrarily large utility improvement.) As the average power increases, the benefit we get from the Lagrangian approach falls; at 13 W, the average improvement in utility is 23%. Above 17 W, there is no improvement because the Lagrangian optimizer and the energy-greedy heuristic both yield exactly the same configuration.

In fact, for several of the workloads, the utility curve is close to flat; for example, this is true of the second and fourth workload shown in Figure 6.2, The flat utility curve is because, in these cases, there is enough energy to run all the high-utility tasks at full quality (achieving the highest utility), and the low-utility tasks do not contribute significantly to the total utility.

At 15 W, we see a dip in running time across all the workloads, resulting results in noticeable reduction in utility for several of the workloads. This dip occurs because when we do the search for $\lambda$, the system estimates that if all applications are run at their highest possible utility (i.e., $\lambda$ is set to zero), the average power will be slightly less than 15 W. In reality, though, the power demand is slightly higher. This is partly, but not entirely, due to a systemic bias in the predictions: the power predictions made when $\lambda$ is calculated do not include energy used to allocate resources to applications as they enter and leave the system. (There is also some bias in the predicted cycle count, because the procedures used to create these tables do not accurately account for the per-application adaptation overhead.)

Because the system uses more energy than is predicted as it actually runs, allowing the applications to all run at maximum utility does not conserve enough energy to run until the end of the run time. Therefore, if a high-utility task appears at the end of the sequence of workloads, it will not run and the system will be unable to achieve the maximum possible utility. There are various solutions to this problem, including better accounting for various overheads and allocating a certain amount of "reserve" energy to ensure that the full runtime is achieved. We have not implemented these options here to demonstrate that while the Lagrangian theory works, a practical implementation should address these issues.

It is also possible for the workload to have an atypically high utility—that is, more high-utility (and therefore energy-consuming) workloads appear than the probability distribution indicates. In these cases, the system will terminate early due to energy exhaustion. This effect can be seen in Workload 1, where the total utility is much higher than the other workloads, but the system shuts down early if the battery holds only enough energy for an average power of 5 W.

## 6.10 Conclusion

In this chapter, we have shown that Lagrangian optimization techniques can be productively applied to the problem of optimizing the allocation of a fixed pool of energy across multiple applications as they enter and leave the system. This approach requires only that the *probabilities* of various workloads be known; foreknowledge of the actual schedule is not required.

Compared to existing constant-power optimization algorithms, the Lagrangian procedure can provide significant improvements in achievable utility when energy is at a premium. If system energy is significantly constrained, utility improvements of 20% or more are achievable.

We have also shown some of the problems that an actual implementation of this approach would encounter. First, the approach is sensitive to the accuracy of the probability distribution of the expected workload; mismatches result in consuming too much energy and terminating early, or consuming too little and achieving less than the best possible utility. Second, if there is no provision for periodically re-allocating the value of $\lambda$, these inaccuracies and any other inaccuracies in the predictions cannot be corrected.

Future work would require extending the implementation of the algorithm. First, the implementation currently assumes a fixed network bandwidth; optimizing across varying networks is important, but was not implemented in our proof-of-concept implementation. Second, the algorithm should dynamically update the workload probabilities based on the actual use of the system, and periodically update $\lambda$ as the energy availability and workload statistics change. This would largely solve the problems shown in the results, where sometimes the predictions result in over- or underestimation of the energy demands of the system for the desired runtime. Third, the Lagrangian algorithms should be integrated more tightly into the GRACE framework. Although the test system uses the GRACE infrastructure, it has not been tightly integrated, and the optimizer $\lambda$ is found and set outside the GRACE system.

# CHAPTER 7

# CONCLUSIONS AND FUTURE RESEARCH

## 7.1 Conclusions

The goal of my research was to design and implement the application-related components of the GRACE system. In this thesis, I have shown the design of the GRACE adaptation system, and provided results from a simulated implementation that show the effectiveness of our hierarchical approach.

Because my work centers on the application, I started by developing an adaptive application to run on our adaptive systems. This application is an H.263-style video encoder that allows trading off processing and network bandwidth through the use of an adaptive motion search, selective DCTs, and the transmission of uncoded macroblocks. The encoder permits the CPU load associated with encoding to be varied by an order of magnitude, and the network load to vary over several orders of magnitude; as a result, if the network bandwidth is large and the network power is low, the total energy of the CPU and network combined can be reduced to a small fraction of its original value. Because modern laptops fit this model— the CPU draws up to 20 W or more, and the network card draws no more than 2-3 W—much of this potential energy savings is actually achievable.

The adaptive application exposes adaptivity to the system, but cannot make allocation or configuration decisions on its own. To do this, we use the GRACE system, a framework that allocates resources and chooses configurations to minimize total energy consumption. The results in Chapter 5 show that the GRACE multilayer adaptation provides significant reductions in energy consumption of the system; we have shown energy savings of more than 50% under some conditions. Furthermore, we have shown that the fine-grained adaptations enabled by the hierarchical adaptation approach permit additional energy savings that cannot be achieved through the use of coarse-time-scale global adaptations alone, and demonstrated that in certain cases a 20% reduction in power can be achieved through the addition of per-application adaptation.

We have found that for laptops with 802.11-style networks, it is nearly universally advantageous to reduce the work done by the CPU by increasing the network utilization. This is primarily because the crossover point at which the network would draw more power than the CPU is far below current or likely future CPU power figures. As a result, the choices for the application configuration are primarily driven by responding to the network constraint. For fixed networks or networks where the network-bandwidth constraint is relatively loose, this means that near-optimal performance can be achieved through the use of global allocation alone.

For tighter network constraints, though, global allocation alone is problematic. Allocations must be made for the worst-case network bandwidth and stream complexity, but doing so reduces the ability of the GRACE system to decrease the CPU workload when network bandwidth is less tight. (The alternative, allocating for a more "average" case, results in large numbers of frame drops when the network bandwidth is reduced or stream complexity increases.) In these cases, the per-frame adaptation permits finer-grained decisions to be made, and enables energy savings by permitting CPU load to be reduced when extra network bandwidth is available.

We have also introduced a novel Lagrangian approach to the global allocation problem. Unlike previous heuristic algorithms, the Lagrangian framework allows for true optimization across varying workloads, even if the workloads are known only statistically. If the workload varies significantly, huge improvements can be realized by reserving more energy for times when high-utility tasks are available.

The Lagrangian approach to global allocation offers modest increases in utility for a "reasonable" workload distribution that might occur in a videoconference or video production environment when energy availability is severely restricted. However, it is better suited to workloads where there is a large disparity between "low priority" applications and "high priority" applications. If such a disparity exists, the Lagrangian approach can offer huge increases in system utility; the scenario we evaluated, with a 100 times disparity and an 80:20 ratio between "low-priority" and "high-priority" tasks, resulted in a utility increase of over a factor of two under severely energy-constrained conditions.

## 7.2 Future Work

This dissertation concentrates on the design of the GRACE system, and an simulated implementation of the GRACE architecture. But in addition to this simulation, the GRACE system as described in Chapter 3 has been implemented on a pair of IBM Thinkpad laptops. This prototype implementation supports CPU adaptation, network bandwidth estimation, and the GRACE multilevel adaptation scheme. Using this implementation of the GRACE system, we have achieved full-system energy savings of up to 12% as measured by a power meter attached to the DC input of the GRACE laptop [72].

Future work will concentrate on porting the utility allocation algorithm introduced in Chapter 6 into the actual implementation of the GRACE framework, and porting the GRACE transport protocol [75] to the implementation of the framework as well. By porting these two components, we will implement the entire GRACE system as originally envisioned.

Although we have shown the theoretical optimality properties of the Lagrangian optimization algorithm and demonstrated its effectiveness, several practical limitations remain. As part of integrating it into the GRACE test bed, we must address these limitations. Specifically, we must address the issues of periodic reallocation and creating and updating the workload-probability tables; the latter in particular is an interesting but difficult prediction problem that deserves significant study.

Also, the Lagrangian optimization algorithm may be well suited for sensor-network applications. The Lagrangian algorithm makes the largest difference when there is a large disparity between between frequent, "low-priority" tasks and infrequent, "high-priority" tasks. On a sensor network, when "interesting" things occur, the utility from spending energy and running high-quality algorithms is much higher than when nothing of relevance has been detected. Sensor networks are also often significantly energy-constrained and intended to last for a particular lifetime. Furthermore, the amount of time spent doing various operations is likely to be more predictable as there is less human input in their operation. Another path of future work would therefore extend the Lagrangian power allocation to remote sensors, possibly using the GRACE ideas of hierarchical adaptation and network and CPU load shifting as well.

# APPENDIX A

# ENCODER LIBRARY INTERFACES

## A.1 Encoder Interface

The original TMN H.263 code was designed as standalone encoder and decoder programs. They were not designed to be linked together into a single application, or into a larger application that supported a network transport. To make a workable test environment, we had to change the encoder and decoder to allow them to be linked into an application that supported error recovery and our cross-layer adaptation environment.

This has been done by collecting the encode-frame and decode-frame functions into a library called "tmn-lib." The library includes various functions to initialize the encoder, encode a frame, and set the various adaptable parameters.

### A.1.1 Core encoder functions

These functions provide the encoder basics: starting and stopping the encoder, and encoding one or more frames.

```
int enc_init(char *options);
int enc_frames(unsigned char *framebuf, unsigned char *recbuf,
int count, unsigned char *bits, int bufsize, int force_i);
int enc_close();
int enc_frame_size(void);

extern int pels;
extern int lines;
```

`enc_init` Initialize the encoder. This function takes a list of encoder parameters. The only important parameter that must be passed is the size of the stream to be encoded. This can be passed as either "-x 1" through "-x 5" representing SQCIF, QCIF, CIF, 4CIF, and 16CIF, or "-X xres yres" for an arbitrarily resolution. To change the encoding resolution, the encoder must be closed and re-opened.

enc_frames encodes one or more frames. It is provided with a pointer to an input buffer holding one or more frames (framebuf) along with a count of the number of frames to be encoded. It returns the encoded frames in the buffer pointed to by bits, which can hold at most bufsize bytes; the number of encoded bytes is returned by the function. If force_i is set, each of the frames being encoded is encoded without reference to past frames; if unset, the frame may or may not be an I-frame depending on the configuration setting of the encoder. Specifically, an I-frame will be encoded under the following conditions:

- The first frame after the encoder is initialized.
- If the fifth bit of enc_features is set.
- If the force_i flag is set.

enc_close closes the encoder and frees all of the memory allocated for the encoder state.

enc_frame_size returns the size (in bytes) of each frame, determined by the options passed to the encoder at initialization. Also, the encoder sets the global values pels and lines with the width and height of the image in pixels, respectively.

### A.1.2 Adaptation controls

```
void enc_Q(int qp, int qpi);
void enc_get_Q(int *qp, int *qpi);
int enc_set_seek(int newdist);
int enc_set_motion_thresh(int thresh);
int enc_set_dct_thresh(int thresh);
double enc_set_block_drop_ratio(double new_ratio);
int enc_features(int clearbits, int setbits);
```

enc_Q sets the quantization step size. The qp parameter is the step size used for P-frames. The qpi parameter is the step size used for I-frames.

enc_get_Q returns the current values for the quantization step sizes for P and I frames in the address pointed to by qp and qpi.

`enc_set_seek` sets the maximum distance from no motion that the motion-search algorithm will check. Valid inputs range from 0 to 15 pixels. If a negative input is provided, the previous value is returned.

`enc_set_motion_thresh` sets the motion-search threshold. If a negative input is provided, the previous value is returned.

`enc_set_dct_thresh` sets the DCT threshold. Again, if a negative input is provided, the previous value is returned.

`enc_set_block_drop_ratio` sets the ratio of blocks that are sent entirely uncoded. Valid values are 0 to 1, inclusive. If a negative input is provided, the previous value is returned.

`enc_features` updates a bitmap indicating which encoder features are enabled. Although several features are available, only one is tested. Setting the fifth bit tells the encoder to encode all frames as I-frames, and clearing it tells it to use P frames. Internally, the current value is AND'd with `clearbits` and OR'd with `setbits`.

### A.1.3 Encoder state manipulation

```
int get_prev_images(unsigned char *image, unsigned char *recon);
int set_prev_images(unsigned char *image, unsigned char *recon);
```

These functions save and restore the state of the encoder. To save the state of the encoder, allocate two buffers of the size returned by `enc_frame_size` (in bytes), and pass pointers as `image` and `recon` to `get_prev_image()`. If these same pointers are then later passed back to `set_prev_images()`, the state of the encoder is reset to the saved state.

This feature is used to ensure that the encoder and decoder remain synchronized even if frames are lost. If a frame loss occurs, the application can restore the state to match the state of the encoder before the missing frame was encoded.

## A.2 Decoder Interface

The decoder works by accepting a buffer containing one or more encoded frames as input. It then decodes the frame into an image buffer (in YUV format), and provides the number of bytes read from the input. This allows the decoder to automatically split an incoming stream into frames without requiring the application structure to understand the H.263 stream format.

### A.2.1 Core decoder functions

```
int decode_init(char *args, unsigned char *firstheader,
int headerlen, int *imagex, int *imagey);
int decode_frame( unsigned char *imageptr,
unsigned char *streamptr, int streamlen );
int decode_close( );
```

`decode_init`: Initialize the decoder.

To open the decoder, we must first load the header of the first frame and make it available. This is so the encoder can read the size out of the stream, and initialize its internal buffers appropriately. The `firstheader` parameter is a pointer to the first frame's header; `headerlen` is the amount of valid data it points to.

`imagex` and `imagey` return the size of the stream's data, which is used to create a window and properly size buffers for saving and restoring the decoder state. Any image buffers should be allocated with a size of `(((*imagex)*(*imagey))*3)/2` bytes.

Once again, `args` points to a string giving arguments. However in this case, no useful arguments exist; this parameter should be an empty string.

`decode_frame`: Decode a frame. This takes as input a pointer to the decoded image, and a pointer to the input and a length of the valid input. It decodes one frame and saves it in the decoded-image buffer, and returns the number of bytes read.

`decode_close`: Close the decoder and deallocate any buffers that have been allocated.

### A.2.2 Decoder state manipulation

```
void decode_get_refframe(unsigned char *buf);
void decode_set_refframe(unsigned char *buf);
```

These functions save and restore the state of the decoder (specifically, the reference frame) in a buffer provided by the application. Obviously, the state of the decoder when decoding the frame must correspond to the state of the encoder when it was encoded, or the frame will be decoded incorrectly.

`decode_get_refframe` puts the current state of the decoder in an image-sized buffer pointed to by `buf`. `decode_set_refframe` retrieves the image from the buffer pointed to by `buf`, and copies it into the decoder.

# REFERENCES


[1] S. Adve et al., "The Illinois GRACE project: Global resource adaptation through cooperation," in *Proceedings of the Workshop on Self-Healing, Adaptive and Self-Managed Systems (SHAMAN)*, New York, NY, June 2002.

[2] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets, "Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems," in *Multimedia Computing and Networking*, Jan. 2003.

[3] A. Fox et al., "Adapting to network and client variation using infrastructural proxies: Lessons and perspectives," *IEEE Personal Communications*, vol. 5, no. 4, pp. 10–19, Aug. 1998.

[4] E. de Lara, D. Wallach, and W. Zwaenepoel, "Puppeteer: Component-based adaptation for mobile computing," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001, pp. 159–170.

[5] B. Noble, "System support for mobile, adaptive applications," *IEEE Personal Communications*, vol. 7, no. 1, pp. 44–49, 2000.

[6] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *Proceedings of the Symposium on Operating Systems Principles*, 1999, pp. 48–63.

[7] J. Flinn et al., "Reducing the energy usage of office applications," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001, pp. 252–272.

[8] S. Narayanswamy et al., "A low-power lightweight unit to provide ubiquitous information access: Application and network support for InfoPad," *IEEE Personal Communications*, vol. 3, no. 2, pp. 4–17, Apr. 1996.

[9] R. Puri and K. Ramchandran, "PRISM: An uplink-friendly multimedia coding paradigm," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 4, Hong Kong, Apr. 2003, pp. 856–859.

[10] D. Kim and D. Shin, "Energy-based adaptive DCT/IDCT for video coding," in *Proceedings of the 2003 International Conference on Multimedia*, vol. 1, July 2003, pp. 557–560.

[11] D.-G. Lee and S. Dey, "Adaptive and energy efficient wavelet image compression for mobile multimedia data services," in *Proceedings of the IEEE International Conference on Communications*, vol. 4, Apr. 2002, pp. 2484–2490.

[12] A. Said and W. A. Peralman, "A new fast and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, pp. 243–250, 1996.

[13] B.-J. Kim and W. A. Peralman, "An embedded wavelet coder using three-dimensional set partitioning in hierarchical trees (SPIHT)," in *Proceedings of the Data Compression Conference*, 1997, pp. 251–260.

[14] S. Choi and J. W. Woods, "MC 3-D subband coding of video," *IEEE Transactions on Image Processing*, vol. 8, pp. 155–167, Feb. 199.

[15] P. Chen, K. Hanke, T. Rusert, and J. W. Woods, "Improvements to the MC-EZBC scalable video coder," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, Barcelona, Spain, Sept. 2003, pp. 81–84.

[16] C. J. Hughes, J. Srinivasan, and S. V. Adve, "Saving energy with architectural and frequency adaptations for multimedia applications," in *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001, pp. 250–261.

[17] D. Dwyer, S. Ha, J.-R. Li, and V. Bharghavan, "An adaptive transport protocol for multimedia communication," in *IEEE Multimedia Systems*, Austin, Texas, 1998, pp. 23–32.

[18] R. Han and D. Messerschmitt, "A progressively reliable transport protocol," *Multimedia Systems*, vol. 7, no. 2, pp. 141–156, 1999.

[19] P. Karn, "Toward new link layer protocols," *QEX*, pp. 3–10, 1994.

[20] P. Chou, A. Mohr, A. Wang, and S. Mehrotra, "FEC and pseudo-ARQ for receiver-driven layered multicast of audio and video," in *Proceedings of the Data Compression Conference*, 1999, pp. 440–449.

[21] J. Nonnenmacher, E. Biersack, and D. Towsley, "Parity-based loss recovery for reliable multicast transmission," in *IEEE/ACM Transactions on Networking*, vol. 6, no. 4, 1997, pp. 349–361.

[22] L. Rizzo and L. Vicisano, "A reliable multicast data distribution protocol based on software FEC techniques," in *Fifth IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS) '97*, 1997, pp. 116–125.

[23] A. Chockalingam, M. Zorzi, and V. Tralli, "Wireless TCP performance with link layer FEC/ARQ," in *Proceedings of the IEEE International Conference on Communications*, 1998, pp. 1212–1216.

[24] D. G. Sachs et al., "Hybrid ARQ for robust video streaming over wireless LANs," in *International Conference on Information Technology: Coding and Computing*, 2001, pp. 317–321.

[25] A. Majumda, D. Sachs, I. Kozintsev, K. Ramchandran, and M. Yeung, "Multicast and unicast real-time video streaming over wireless LANs," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 6, pp. 524–534, June 2002.

[26] A. F. Harris, C. Sengul, R. Kravets, and P. Ratanchandani, "Energy-efficient transmission policies for multimedia in multi-hop wireless networks," in *The Sixth IFIP IEEE International Conference on Mobile and Wireless Communication Networks (MWCN)*, Paris, France, Oct. 2004.

[27] T. Pering, T. Burd, and R. Brodersen, "Voltage scheduling in the lpARM microprocessor system," in *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, Rapallo Italy, July 2000, pp. 96–101.

[28] P. Pillai and K. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001, pp. 89–102.

[29] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time CPU scheduling for mobile multimedia systems," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, Oct. 2003, pp. 149–163.

[30] G. M. Davis and J. M. Danskin, "Joint source-channel coding for image transmission over lossy packet networks," in *SPIE Conference on Wavelet Applications of Digital Image Processing XIX*, 1996.

[31] K. Ramchandran, A. Ortega, K. Uz, and M. Vetterli, "Multiresolution broadcast for digital HDTV using joint source/channel coding," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 6–23, Jan. 1993.

[32] S. B. Z. Azami, O. Rioul, and P. Duhamel, "Performance bounds for joint source-channel coding of uniform memoryless sources using a binary decomposition," in *Proceedings of the European Workshop on Emerging Techniques for Communication Terminals*, 1997, pp. 259–263.

[33] B. Belzer, J. D. Zillasenor, and B. Girod, "Joint source-channel coding of images with trellis coded quantization and convolutional codes 1995," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, 1995, pp. 85–88.

[34] H. Man, F. Kossentini, and M. Smith, "Progressive image coding for noisy channels," *IEEE Signal Processing Letters*, vol. 4, pp. 8–11, August 1997.

[35] J. Lu, A. Nosratinia, and B. Aazhang, "Progressive source-channel coding of images over bursty error channels," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, 1998, pp. 127–131.

[36] M. P. C. Fossorier, Z. Xiong, and K. Zeger, "Joint source-channel image coding for a power constrained noisy channel," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, 1998, pp. 122–126.

[37] M. Brystrom and J. W. Modestino, "Combined source channel coding for transmission of video over a slow-fading Rician channel," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, 1998, pp. 147–151.

[38] T.-H. Lan and A. H. Tewfik, "Power optimized mode selection for h.263 video coding and wireless communications," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, 1998, pp. 113–117.

[39] H. Zheng and K. J. R. Liu, "Image and video transmission over a wireless channel: A subband modulation approach," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, 1998, pp. 132–136.

[40] S. Aramvith, I.-M. Pao, and M.-T. Sun, "A rate-control scheme for video transport over wireless channels," *IEEE Transactions on Circuits and Systems*, vol. 11, pp. 569–580, May 2001.

[41] I.-M. Kim, H.-M. Kim, and D. G. Sachs, "Power-distortion optimized mode selection for transmission of VBR videos in CDMA systems," *IEEE Transactions on Communication*, vol. 51, no. 4, pp. 525–529, Apr. 2003.

[42] R. Kravets, K. Calvert, and K. Schwan, "Payoff adaptation of communication for distributed interactive applications," *Journal on High Speed Networking: Special Issue on Multimedia Communications*, vol. 7, no. 3, pp. 301–318, 1998.

[43] S. Appadwedula, D. L. Jones, K. Ramchandran, and L. Qian, "Joint source-channel matching for wireless image transmission," in *Proceedings of the International Conference on Image Processing*, 1998, pp. 137–141.

[44] S. Appadwedula, M. Goel, N. Shanbhag, D. Jones, and K. Ramchandran, "Total system energy minimization for wireless image transmission," *Journal of VLSI Signal Processing Systems*, vol. 27, no. 1/2, pp. 99–117, February 2001.

[45] L. Qian, D. L. Jones, K. Ramchandran, and S. Appadwedula, "A general joint source-channel matching method for wireless video transmission," in *Proceedings of the Data Compression Conference*, 1999, pp. 414–423.

[46] L. Qian, "Joint source-channel video transmission," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2001.

[47] K.-W. Lee et al., "An integrated source coding and congestion control framework for video streaming in the internet," in *INFOCOM*, vol. 2, 2000, pp. 747–756.

[48] A. E. Mohr, E. A. Riskin, and R. E. Ladner, "Unequal loss protection: Graceful degradation of image quality over packet erasure channels through forward error correction," *IEEE Journal on Selected Areas in Communication*, vol. 18, no. 6, pp. 819–829, 2000.

[49] P. Chou and Z. Miao, "Rate-distortion optimized streaming of packetized media," Microsoft Research, Tech. Rep. MSR-TR-2001-35, 2001.

[50] D. G. Sachs, A. Raghavan, and K. Ramchandran, "Wireless image transmission using multiple-description based concatenated codes," in *SPIE Conference on Image and Video Communications and Processing*, Jan. 2000.

[51] D. G. Sachs, "Wireless image transmission over fading channels using multiple description concatenated codes," M.S. thesis, University of Illinois at Urbana-Champaign, May 2000.

[52] K. Nahrstedt, S. H. Shah, and K. Chen, "Cross-layer architectures for bandwidth management in wireless networks," in *Resource Management in Wireless Networking*, M. Cardei et al., Eds. New York: Springer, 2005, pp. 41–62.

[53] V. Bharghavan, K. Lee, S. Lu, S. Ha, J. Li, and D. Dwyer, "The TIMELY adaptive resource management architecture," *IEEE Personal Communications*, vol. 5, no. 4, pp. 20–31, Aug 1998.

[54] S. H. Shah, K. Chen, and K. Nahrstedt, "Dynamic bandwidth management in single-hop ad-hoc wireless networks," *Mobile Networks and Applications*, vol. 10, no. 1–2, pp. 199–217, Feb. 2005.

[55] P. Sudame and B. Badrinath, "On providing support for protocol adaptation in mobile wireless networks," *Mobile Networks and Applicatons*, vol. 6, no. 1, pp. 43–55, 2001.

[56] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, "A scalable solution to the multi-resource QoS problem," in *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Dec. 1999, pp. 315–326.

[57] E. de Lara, D. Wallach, and W. Zwaenepoel, "HATS: Hierarchical adaptive transmission scheduling for multi-application adaptation," in *Proceedings of the SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2002.

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

[59] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat, "ECOSystem: Managing energy as a first class operating system resource," in *Proc. of 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002, pp. 123–132.

[60] C. Rusu, R. Melhem, and D. Mosse, "Maximizing the system value while satisfying time and energy constraints," in *Proceedings of the 23rd Real-Time Systems Symposium (RTSS 02)*, Austin, TX, Dec. 2002.

[61] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat, "Currentcy: A unifying abstraction for expressing energy management policies," in *USENIX 2003 Annual Technical Conference*, San Antonio, TX, June 2003, pp. 43–56.

[62] K. Gopalan and T. Chiueh, "Multi-resource allocation and scheduling for periodic soft real-time applications," in *Proceedings of the SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2002.

[63] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst, "A platform supporting coordinated adaptation in mobile systems," in *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, June 2003, pp. 128–137.

[64] D. G. Sachs, S. Adve, and D. L. Jones, "Cross-layer adaptive video coding to reduce energy on general-purpose processors," in *Proceedings of the IEEE International Conference on Image Processing*, vol. 3, Barcelona, Spain, Sept. 2003, pp. 109–112.

[65] V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM Reference Manual version 1.0," Department of Electrical and Computer Engineering, Rice University, Tech. Rep. 9705, August 1997.

[66] V. Vardhan et al., "Integrating fine-grained application adaptation with global adaptation for saving energy," in *Proceedings of the 2nd International Workshop on Power-Aware Real-time Computing (PARC)*, Jersey City, NJ, Sept. 2005.

[67] D. G. Sachs et al., "GRACE: A hierarchical adaptation framework for saving energy," University of Illinois at Urbana-Champaign, Department of Computer Science, Tech. Rep. UIUCDCS-R-2004-2409, Feb. 2004.

[68] D. G. Sachs et al., "GRACE: A cross-layer adaptation framework for saving energy," *IEEE Computer Magazine*, special issue on Power-Aware Computing, pp. 50–51, Dec. 2003.

[69] Telenor Research, "TMN H.263 encoder and decoder, version 1.7," 1996, http://www.xs4all.nl/~roalt/telenor.tar.gz.

[70] J. R. Jain and A. K. Jain, "Displacement measurement and its application in interframe image coding," *IEEE Transactions on Communications*, vol. 29, no. 12, pp. 1799–1808, Dec. 1981.

[71] C. Hughes et al., "Variability in the execution of multimedia applications and implications for architecture," in *Proceedings of the International Symposium on Computer Architecture*, 2001, pp. 254–265.

[72] A. F. Harris et al., "The Illinois GRACE project: Energy-efficient cross-layer system design," presented at MobiCom, Cologne, Germany, 2005.

[73] M. Moser, D. Jokanovic, and N. Shiratori, "An algorithm for the multidimensional multiple-choice knapsack problem," *IEICE Transactions on Fundamentals*, vol. E80-A, no. 3, pp. 582–589, Mar. 1997.

[74] H. Thon, "Going mobile a la AMD: Low-voltage Athlon XP-M vs. Pentium-M (Banias)," 2003, http://www.tomshardware.com/mobile/20030507/lifebook-02.html.

[75] A. F. Harris, "Cross-layer energy-efficient mobile system design," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Mar. 2006.

[76] W. Yuan and K. Nahrstedt, "ReCalendar: Calendaring and scheduling applications with CPU and energy resource guarantees for mobile devices," in *Proceedings of the IEEE Pervasive Computing and Communications (PerCom03)*, Mar. 2003.

[77] H. Everett, "Generalized Lagrange multiplier method for solving problems of optimum allocation of resources," *Operations Research*, vol. 11, no. 3, pp. 399–418, May/Jun 1963.

[78] B. S. Krongold, K. Ramchandran, and D. L. Jones, "Computationally efficient optimal power allocation algorithms for multicarrier communication systems," *IEEE Transactions on Communications*, vol. 48, no. 1, pp. 23–27, January 2000.

[79] K. Ramchandran and M. Vetterli, "Best wavelet packet bases in a rate-distortion sense," *IEEE Transactions on Image Processing*, vol. 2, no. 2, pp. 160–175, 1993.

[80] Y. Shoham and A. Gersho, "Efficient bit allocation for an arbitrary set of quantizers," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 9, pp. 1445–1453, Sep 1988.

[81] F. Kelly, "Charging and rate control for elastic traffic," *European Transactions on Telecommunications*, vol. 8, pp. 33–37, 1997.

[82] M. Goel and N. R. Shanbhag, "Dynamic algorithm transforms for low-power reconfigurable adaptive equalizers," *IEEE Transactions on Signal Processing*, vol. 47, no. 10, pp. 2821–2832, Oct. 1999.

# AUTHOR'S BIOGRAPHY

Daniel Grobe Sachs was born on January 5, 1976, in Chicago, IL. After graduating from York Community High School in Elmhurst, IL, he went to the University of Illinois at Urbana-Champaign for his undergraduate education, receiving a B.S. from the Department of Computer Science in 1998. He graduated with departmental highest honors and received the prestigious Bronze Tablet award for his academic achievement.

After receiving his B.S, Daniel remained at the University to enter graduate school in the Department of Electrical and Computer Engineering, where he received an M.S. degree in 2000 under the direction of advisor Douglas L. Jones and Kannan Ramchandran. He completed his Ph.D. in 2006 under the direction of research advisor Douglas L. Jones and project principal investigator Sarita Adve.

In addition to his academic achievements, Daniel has been a summer employee of Motorola, Intel, and MIT Lincoln Laboratory, and spent several years as a student contractor at the U.S. Army Construction Engineering Laboratory. He has also participated in a research exchange program at the University of Tokyo in Tokyo, Japan.