GRACE-2: Integrating Fine-Grained Application Adaptation with Global Adaptation for Saving Energy*

Vibhore Vardhan
Daniel G. Sachs
Wanghong Yuan
Albert F. Harris
Sarita V. Adve
Douglas L. Jones
Robin H. Kravets
Klara Nahrstedt
Department of Computer Science
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, USA
E-mail: grace@cs.uiuc.edu

Abstract:
Energy efficiency has become a primary design criterion for mobile multimedia devices. Prior work has proposed saving energy through coordinated adaptation in multiple system layers, in response to changing application demands and system resources. The scope and frequency of adaptation pose a fundamental conflict in such systems. The Illinois GRACE project addresses this conflict through a hierarchical solution which combines (1) infrequent (expensive) *global* adaptation that optimizes energy for all applications in the system and (2) frequent (cheap) *per-application* (or *per-app*) adaptation that optimizes for a single application at a time. This paper demonstrates the benefits of the hierarchical adaptation through a second-generation prototype, GRACE-2. Specifically, it shows that in a network bandwidth constrained environment, per-app application adaptation yields significant energy benefits over and above global adaptation.

Keywords: power management, mobile devices, cross-layer adaptation, hierarchical adaptation, multimedia applications, resource management

Reference to this paper should be made as follows: Vardhan, V., Sachs, D.G., Yuan, W., Harris, A.F., Adve, S.V., Jones, D.L., Kravets, R.H. and Nahrstedt, K. (2007) 'GRACE-2: Integrating Fine-Grained Application Adaptation with Global Adaptation for Saving Energy', Int. J. Embedded Systems †

Biographical notes: Vibhore Vardhan is a Software Systems Architect at Texas Instruments Inc., USA. His research interests are in high-performance low-power embedded system design. He received his M.S. in 2004 from University of Illinois.

*This work is supported in part by the National Science Foundation under Grant No. CCR-0205638 and a gift from Texas Instruments.

†This work is an extension of our paper in the 2nd International Workshop on Power-Aware Real-Time Computing (30). Major additions include a detailed description of the GRACE-2 implementation (Section 4), workloads and resource constraint scenarios evaluated (Section 5), GRACE-2 overheads (Section 6), complete results for global and per-app adaptation (Section 7.1), analysis of the results

Copyright © 200x Inderscience Enterprises Ltd.

1 Introduction

Mobile devices primarily running soft real-time multimedia applications are becoming an increasingly important computing platform. Such systems are often limited by their battery life, and saving energy is a primary design goal. A widely used energy saving technique is to adapt the system in response to changing application demands and system resources. Researchers have proposed such adaptations in all layers of the system; e.g., hardware, application, operating system, and network. Recent work has demonstrated significant energy benefits in systems that employ coordinated multiple adaptive system layers or cross-layer adaptation (33; 32).

Such systems must employ intelligent control algorithms that determine when and what adaptations to invoke, to exploit the full potential of the underlying adaptations. These algorithms must balance the conflicting demands of adaptation scope and frequency. On one hand, an algorithm that considers all applications and adaptive system layers, referred to as global, is likely to save more energy than a more limited scope algorithm (e.g., considering only one application at a time). On the other hand, global algorithms are also likely to be more expensive since they must optimize across the cross-product of all configurations of all adaptive layers, considering the demands of all (possibly adaptive) applications on these configurations.

Previous cross-layer adaptation work, therefore, performs global adaptation relatively infrequently (e.g., when an application enters or leaves the system (33; 32)). This infrequent invocation in turn reduces the system's responsiveness to change, potentially sacrificing energy benefits. Other work performs adaptations more frequently, but assumes only one application in the system (28) or only a single adaptive layer (9).

To balance the conflict of frequency vs. scope, the Illinois GRACE project (Global Resource Adaptation through CoopEration) takes a *hierarchical approach* that invokes expensive global adaptation occasionally, and inexpensive limited-scope adaptations frequently (27; 33; 32). GRACE uses three adaptation levels, exploiting the natural frame boundaries in periodic real-time multimedia applications (Figure 1 (27)). *Global* adaptation considers all applications and system layers together, but only occurs at large system changes (e.g., application entry or exit). *Per-application* adaptation (or *per-app*) considers one application at a time and is invoked every frame, adapting all system layers to that application's current demands. *Internal* adaptation adapts only a single system layer (possibly considering several applications) and may be invoked several times per application frame. All adaptation levels are tightly coupled by ensuring that the limited-scope adaptations respect the resource allocations made by global adaptation. The different adaptation levels may or may not consider the same adaptations; they are distinguished by the granularity at which they consider an adaptation (e.g., both global and per-app levels may consider dynamic voltage and frequency scaling or DVFS for CPU adaptation).

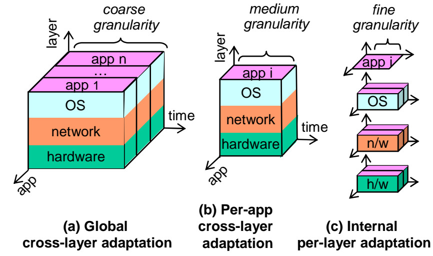


Figure 1: GRACE adaptation hierarchy. (We do not yet adapt the network.)

We previously reported on the first GRACE prototype, GRACE-1, with adaptations in the CPU (DVFS), application (frame rate and dithering), and soft real-time scheduler (CPU time allocation) (33; 32). GRACE-1's focus was on cross-layer *global* adaptation, for which it showed significant energy benefits. It reported a few experiments with hierarchical adaptation in the CPU and scheduler, but showed only modest benefits over global adaptation when running multiple applications.

This work focuses on the benefits of hierarchical adaptation in a mobile multimedia system, and reports results from the second generation prototype, GRACE-2. Our main contribution is to show that *per-app application adaptation provides significant benefits over and above global adaptation when network bandwidth is constrained.* These benefits occur with and without per-app CPU adaptation. Notably, the benefits with both per-app application and per-app CPU adaptation are often more than additive. In contrast, GRACE-1 neither provided per-app application adaptation nor implemented a network constraint, and is thus unable to obtain GRACE-2's benefits. Further, GRACE-1's hierarchical adaptation had to be redesigned to incorporate per-app application adaptation because it implicitly assumed a fixed application configuration between global adaptations.

GRACE-2 is implemented on a Pentium M based laptop running Linux 2.6.8-1. As illustrated in Table 1, GRACE-2 implements global adaptations in the CPU, application, and soft real-time scheduler; per-app adaptation in the CPU and application; and internal adaptation in the scheduler. It respects the constraints of CPU utilization and network bandwidth, while minimizing CPU and network transmission energy. All aspects of the system are fully implemented except for network communication. We report both the measured energy savings for the entire system and modeled energy savings for just the CPU and network (we could not isolate the CPU energy through measurements).

We emphasize that the individual adaptations in GRACE-2 are not our focus, and have been previously proposed. Our focus is on their hierarchical control, and specifically on per-app application adaptation.

(Section 7.2), and complete results for system-wide energy savings (Section 7.3).

Objective: Minimize CPU and network transmission energy
Constraints: CPU time, network bandwidth

Layer	Adaptation	Hierarchy level		
		Global	Per-app	Int.
CPU	Dynamic voltage and frequency scaling (DVFS)	yes	yes	no
Application	Drop DCT and motion estimation computations based on adaptive thresholds	yes	yes	no
Scheduler	Change CPU time, network bandwidth budget	yes	no	yes

Table 1: Adaptations supported in GRACE-2

To our knowledge, this work is the first to demonstrate the benefits (energy savings) from per-app application adaptation over and above global adaptation. It is also the first to demonstrate significant benefits from hierarchical adaptation on a real multimedia system implementing multiple applications, adaptations, and constraints. Section 8 further discusses related work.

2 Layer Adaptations and Models

2.1 CPU

Adaptations: We study dynamic voltage and frequency scaling (DVFS). Our Pentium M CPU supports five frequencies {600, 800, 1000, 1200, 1300 MHz} and corresponding voltages {956, 1260, 1292, 1356, 1388 mV} (15).

To partially alleviate the limitations of the small number of discrete DVFS points supported, we emulate a continuous set of DVFS points as follows (16). If we need to run at an unsupported frequency, f, we run at the supported frequency just below f (say f_l) for some number of cycles (say c_l) and the supported frequency just above f (say f_h) for the remaining cycles (say c_h). If c cycles need to be executed, then $c_l + c_h = c$ and $\frac{c}{f} = \frac{c_l}{f_l} + \frac{c_h}{f_h}$

Energy model: We report energy measurements from the actual system. However, we could not isolate the CPU energy from the rest of the measured system energy. To better understand the impact of our adaptations on the CPU energy and to provide a CPU energy model to the adaptation control algorithms, we use the following: Energy = Power × Execution Time, where we approximate power at frequency f and voltage V by dynamic power $\propto V^2 \times f$.

We derive the proportionality constant using published numbers for the maximum Pentium M power. The above model does not incorporate leakage (static) power or the effect of application-specific clock gating (as is the case in much of the DVFS literature). These are difficult to incorporate analytically and do not affect the overall trends in the impact of per-app adaptation. This is substantiated by our measured (entire system) energy numbers which do include all effects.

For future systems where leakage is expected to dominate, GRACE will need to incorporate leakage-driven adaptations and energy models for effective energy savings. For example, running application frames quickly and shutting down the CPU until the next frame may save more leakage energy than the savings from reduced voltage with DVFS. Similarly, power gating different CPU structures will also save leakage energy. The GRACE framework can easily incorporate such adaptive configurations with corresponding energy models (e.g., obtained through profiling (14), vendor-specified estimates of leakage power, etc.). The fundamental tradeoff between processor and network energy will still remain and GRACE can take advantage of it in just the same way. It may appear that the linear power savings expected out of leakage-driven adaptations may be significantly lower than the traditionally expected quadratic power savings from dynamic power driven DVFS. We note, however, that our experimental system is already in the CMOS regime where frequency reductions result in sub-linear voltage reductions and DVFS savings are already closer to linear than quadratic. Thus, we expect our results here to reflect trends in the leakage dominated future as well.

2.2 Network (non-adaptive)

We assume a non-adaptive (simulated) network layer with fixed available bandwidth. We model network transmission energy using a fixed energy/byte cost: Network Energy = EnergyPerByte × BytesTransmitted (5). Table 2 summarizes energy per byte for different bandwidth values in an IEEE 802.11b wireless network, based on the energy consumption of a Cisco Aironet 350 series PC card (5). Although here we only model network transmission energy, our model can be enhanced to include fixed costs such as idle energy. This will not affect our current results because the Pentium M CPU on our laptop uses nearly 10 times more power than the wireless network card.

We use different bandwidth values to model different constraints in the system. If the value selected is between two values in Table 2 (possible since not all the bandwidth of the channel is available to one node), we assume the transmission cost of the higher bandwidth. We believe our network configurations represent reasonable scenarios seen in practice. Responding to variations in network bandwidth with an adaptive network layer is part of our ongoing work.

Bandwidth (Mbps)	2	5.5	11
Energy per byte (e^{-6} J)	4	2	.08

Table 2: Network bandwidth and energy/byte.

2.3 Applications

We consider periodic soft real-time applications or tasks. An application releases a job or a *frame* at the beginning of each period. We study workloads consisting of various combinations of speech and video encoders and decoders (Section 5). Our H.263 video encoder is adaptive while the other applications are non-adaptive.

Adaptations in the H.263 video encoder: We use the adaptations proposed in (28) (in the context of a system with a single application, and without global adaptation). Since these are not our focus, we only summarize them next and refer to (28) for details.

The adaptations trade-off CPU computation (i.e., CPU energy) for the number of bytes transmitted (i.e., network transmission energy), to minimize the total CPU+network transmission energy. The appropriate trade-off varies dynamically, depending on the video stream, the system load, and the ratio of network energy per byte to CPU energy per cycle (which depends on the chosen CPU frequency).

The adaptations work at the granularity of a single video frame. They enable dropping certain DCT (discrete cosine transform) computations and motion searches based on a threshold (set by the adaptation control algorithm) for the corresponding frame. The net effect is that, by changing the thresholds, the control algorithm can vary the bit rate and the computation cycles for a frame by about a factor of two. These adaptations can potentially reduce the PSNR (pseudo signal to noise ratio) of the stream, but this is compensated for by adjusting the quantizer step size. Thus, the adaptive encoder can be scaled between a highly compute-intensive but lower bit rate configuration to a less compute-intensive higher bit rate configuration, *without affecting the quality of the decoded video.*

We study four DCT and four motion-search thresholds, resulting in a configuration space of sixteen different encoder configurations.

Deadline misses and frame drops: A frame that does not complete computation or transmission of all its bytes by the end of the ensuing period is said to miss its deadline, with one exception. For video encoders, if a frame finishes its computation within 1ms of its period, we do not count it as a miss. We find these delays do not accumulate (the misses are not clustered). If the video encoder misses its deadline for one frame, the encoding/transmission for that frame continues in the next period, borrowing from the budget of the next frame. If it misses the deadline for two frames in a row, then the next frame is entirely dropped (i.e., incurs no computation or network transmission), enabling the encoder to catch up on its previous frame overruns. We have not (yet) modified the other applications to drop frames.

Since we use soft real-time applications, we assume that we may miss the deadline for or drop a total of up to 5% of all frames, without affecting quality. Although strictly speaking, missing a deadline by a small interval and dropping an entire frame have different effects on quality, we do not distinguish between the two and seek to limit both of these effects to a total of 5%.

2.4 OS Scheduler

We assume an earliest-deadline-first (EDF) soft real-time scheduler for CPU time and network bandwidth. The scheduler is responsible for enforcing budget allocations for both CPU time and network bandwidth. To reduce deadline misses due to imperfect predictions of resource demands, the scheduler performs an internal adaptation called budget sharing (3). Briefly, this allows an application to reclaim unused budget from previous applications' underruns. The EDF CPU scheduler maintains a record of all unused budgets and their expiration times (i.e., the deadline for the job that released the budget). When an application is scheduled, the scheduler first tries to exhaust any unused budget before charging the elapsed cycles to the application. The unused budget can be given to an application only if the expiration time of the budget is less than the deadline for the application (3). We similarly exploit network bandwidth sharing between applications. Unless stated otherwise, *budget sharing is used in all systems studied here.*

3 Adaptation Control Algorithms

3.1 Global Control

Overview: We use a global control algorithm similar to that in (32), but extended to incorporate a network bandwidth constraint. The algorithm is invoked on large changes in the system; e.g., when an application enters or exits. As input, the algorithm receives the resource requirements (CPU utilization, network bandwidth, CPU+network energy) for each combination of application and CPU configuration. The algorithm must then choose, for each application, the combination of the application and CPU configuration such that (i) the total CPU+network energy is minimized, and (ii) the resource requirements for all the applications (running with the chosen configurations) are met.

More formally, for application i, let Period_i be its period and C_i be a chosen CPU and application configuration combination. Let Energy_{i,C_i} be the energy consumed, Time_{i,C_i} be the CPU time taken, and Bytes_{i,C_i} be the network bytes required by a frame of application i with configuration C_i. Let there be a total of N_{apps} applications in the system and let B be the total network bandwidth (assumed to be fixed). Then the global algorithm must choose the CPU and application configuration C_i for each application i to:

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

subject to EDF scheduling and bandwidth constraints:

$$\sum_{i=1}^{N_{apps}} \frac{\text{Time}_{i,C_i}}{\text{Period}_i} \leq 1 \quad \text{and} \quad \sum_{i=1}^{N_{apps}} \frac{\text{Bytes}_{i,C_i}}{\text{Period}_i} \leq B$$

Solving the optimization: The above optimization problem is a multi-dimensional multiple-choice knapsack problem (MMKP) (20) and is known to be NP-hard. For the purpose of determining energy savings, we solve this problem using a brute force exhaustive search approach (with one modification below), to give global control the best showing. This approach is impractically expensive for a real system. When reporting the overhead for global, we use a more practical, but possibly sub-optimal heuristic approach based on Lagrangian techniques (20). (We found the energy savings of both approaches to be comparable for the scenarios studied here.)

To reduce the complexity of both solution approaches, we choose the same frequency (CPU configuration) for all applications. We justify this simplification with Jensen's inequality (18): if the CPU energy per unit time (power) is a convex function of frequency, then the best frequency setting is a single point for all applications (if the CPU does not support this single point, then a combination of adjacent supported frequencies is best). We note that the conditions required for optimality may not hold for some situations; e.g., with a mix of compute- and memory-intensive applications, since the CPU power versus frequency relationship depends on memory behavior. As discussed below, our applications are not memory intensive and the same-frequency rule yields near-optimal results. For other applications and systems, other solution approaches will be required, making global adaptation even more expensive. Our use of the heuristic enables us to solve the MMKP problem separately for each supported frequency. We then pick the frequency that provides the minimum energy with the chosen application configurations at that frequency.

After the above process, it is possible that the chosen application configurations and frequency do not exhaust all the CPU utilization and network bandwidth. In that case, the leftover resources are divided among the applications in proportion to their current allocation. This leftover CPU utilization allows a further reduction in frequency. If the resulting frequency is not directly supported, the continuous DVFS emulation discussed in Section 2.1 is used.

Predicting resource requirements: The global algorithm requires predicted resource usage of a frame (Energy_{i,C_i}, Time_{i,C_i}, and Bytes_{i,C_i} in the optimization equations). These predictions must be representative of all frames until the next global adaptation is invoked. Following previous work on resource allocation and scheduling for soft real-time multimedia applications (4; 32), we use profiling of several frames to determine the resource usage. (In our experiments, since our streams are relatively short and since we would like to give global control the best showing, we profiled the entire stream off-line.)

To reduce the amount of profiling, we leverage findings from (13). Specifically, for our applications, the number of execution cycles for a given frame for a given application configuration is roughly independent of frequency; therefore, execution time scales roughly linearly with frequency.[1] Thus, by profiling each application configuration at a single CPU frequency, we are able to estimate the execution time (and the number of bytes) at all frequencies. These estimates also allow estimation of energy using the models in Section 2. (For more general applications, frequency-dependent estimates of execution time and energy would need to be deduced – a combination of more profiling and interpolation or alternate models could be used, but was not required for our work.)

Since we assume a 5% deadline miss rate is acceptable, we use the execution time (and bytes) from the frame that falls in the 95th percentile of all profiled frames. For energy, we are concerned with minimization and not meeting a constraint. We therefore use the average time and bytes from the profiled frames as input to the energy models.

3.2 Per-App Control

The per-app control algorithm (derived from (28)) is invoked at the start of a frame with the following inputs: (1) the resource allocation for the frame and (2) the resource requirements for the frame for each application configuration.

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

Predicting resource requirements: As for the global algorithm, estimating the execution cycles and bytes for a frame enables estimating all its resource requirements (execution time, bandwidth, and energy). Unlike global control, per-app control requires predicting resource usage for only the next frame.

For non-adaptive applications, we use a common history-based heuristic technique that uses execution information of the last few frames to predict the behavior of the next frame (14). There is a tradeoff between using history from too many frames (which may not capture enough short-term variation in resource usage) and too few frames (which may result in unnecessary response to random one-time fluctuations). We experimentally determined that the average of the execution cycles and bytes of the last five frames to predict these quantities for the next frame gave adequate predictability for our system. For the adaptive application, the history of the past few frames may be for different application configurations, and cannot be used

[1] This is because these applications generally hit in the cache and do not see much memory stall time (13).

directly to predict the behavior of the next frame for yet other configurations. We therefore use an off-line profiling based prediction technique proposed by Sachs et al. as follows (28).

The technique generates an execution cycle predictor off-line by repeatedly encoding one or more sequences (for a fixed hardware frequency), randomly changing the encoder configuration at each frame. This off-line run generates several points for every pair of (previous, next) encoder configurations, mapping the number of cycles in the previous frame to those in the next frame. The predictor is generated by fitting a function in the least-squared error sense, for every pair of (previous, next) configurations. A byte count predictor is similarly generated. To avoid deadline misses, we conservatively add an adaptive leeway into the predicted values for both execution cycles and bytes. Improving the predictors for adaptive applications is part of our ongoing work.

When the per-app adaptation is invoked, it determines the cycle count and byte count for each application configuration for the next frame by using the appropriate predictor, given the knowledge of the previous frame's application configuration, actual cycle count, and actual byte count.

3.3 Integrating Global and Per-App Control

A system that runs with only global control uses the frequency and application configurations as chosen by the global algorithm. In a system that additionally incorporates per-app control, the global algorithm's choice of configuration is only used to determine the resource allocation for each application. This resource allocation is fed as input to the per-app control algorithm. The latter then determines the appropriate configurations for the next frame based on its predictions of the resource usage of that frame and its allocation. Since the per-app controller makes a prediction only for the next frame (using information from the last few, in our case five, frames), it is likely that its prediction is better than that of the global algorithm (which must make a conservative prediction that will accommodate the resource usage of all frames until the next global adaptation). Therefore, the per-app controller is likely to better utilize the resources that were allocated to its application by the global algorithm. Figure 2 summarizes the integrated system. As shown, the only interaction between the global and per-app controller is that the former gives the resource allocation to the latter.

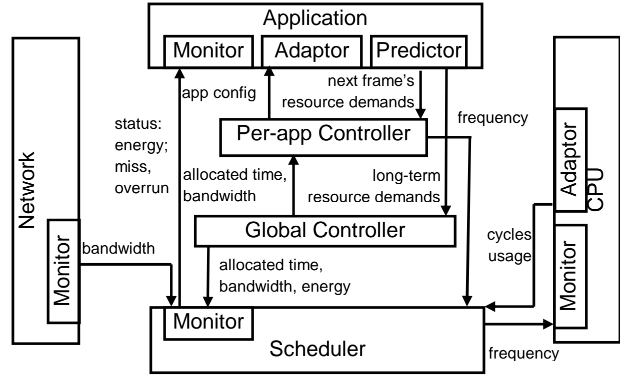


Figure 2: Integrated global and per-app control.

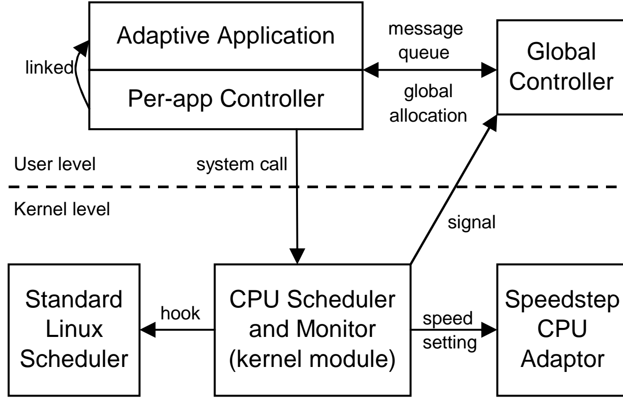


Figure 3: Software architecture. The implementation uses an IBM ThinkPad R40 laptop running the Linux kernel 2.6.8-1.

4 Implementation

We have implemented all aspects of the system studied except for the network communication (which is replaced with file I/O). Our implementation is on an IBM ThinkPad R40 laptop running the Linux kernel 2.6.8-1. The laptop has a single Intel Pentium M processor, which features Intel's Enhanced SpeedStep technology with the DVFS points summarized in Section 2.1. The processor can be made to transition between DVFS points at run time by the operating system. We implement our operating system components as a set of patches that hook into the Linux kernel.

Figure 3 gives an overview of the GRACE-2 software architecture. The architecture builds on that developed for GRACE-1 (33; 32), but with significant additions and changes in the implementation. Major differences include the addition of the per-app controller, support for continuous DVFS and budget sharing in the CPU scheduler, changes to the application interface and the CPU scheduler to incorporate per-app adaptation, and a different algorithm for the global controller to incorporate network constraints. We discuss the main components of the implementation next.

4.1 Global controller

The global controller is implemented as a separate user-level process because (1) its computation involves double precision floating point variables, which is currently not supported in the Linux kernel module, and (2) a user-level

global controller can run at a lower priority than the applications, ensuring that the expensive global optimization process does not supersede applications. The global and per-app controllers communicate via a message queue.

4.2 Per-app controller

The per-app controller is designed as a generic function that can be linked with the application at compile time. This has two advantages over making it part of the global controller: (1) the cycles it uses are charged to the corresponding application by the CPU scheduler, and (2) the application can ensure that per-app adaptation occurs at the start of every frame. The advantage over implementing it in the kernel is in the reduced number of system calls. However, a disadvantage of linking with the application is that it is vulnerable to malicious applications, making it non-trustworthy. We can circumvent this problem by sending global allocations to the CPU scheduler, and having the scheduler enforce these allocations.

4.3 CPU scheduler

We use an EDF based Soft Real-Time (SRT) CPU scheduler (Section 2.4). (As mentioned above, the scheduler builds on the GRACE-1 implementation (33), but is significantly enhanced for per-app adaptation, continuous DVFS, and budget sharing.)

Invocation of the scheduler and GRACE-2 system calls: The scheduler is invoked either when a timer it started expires or when an application makes a system call.

The scheduler may set the timer for several reasons. For example, before starting a new application frame, it sets a timer to expire when the application's budget runs out, to enable handling overruns. At the end of an application frame, it sets a timer to expire at the start of a new period for the application, to schedule its next frame. Per-app control requires a low overhead, high resolution timer, so we use the High Res Posix timers (1).

The application may invoke the scheduler for various reasons, through five system calls:

EnterSrt is invoked when the application first joins the system. The CPU scheduler initializes its data structures for the new application, inserts it into the SRT task list, and signals the global controller.

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

If the frequency is not supported, the CPU scheduler calculates the continuous DVFS values to emulate the frequency (Section 2.1). It invokes the CPU adaptor to set the CPU speed to the lower continuous DVFS frequency, and sets a timer to expire at the end of the low frequency interval. When the timer expires, the scheduler invokes the CPU monitor to get the resource usage, and the CPU adaptor to set the frequency to the higher continuous DVFS frequency.

FinishJob is invoked when the application finishes its frame. The CPU scheduler gets the resource usage (elapsed cycles, energy) from the CPU monitor, checks for deadline miss, and sends the resource usage and miss status information back to the application. The monitor checks the cycle usage by using the *rdtscll* function in the Linux kernel. It estimates the CPU energy using the model in Section 2.

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

ExitSrt is invoked when the application is done with all its frames. The scheduler removes the application from the SRT list, cleans up related data structures, and signals the global controller.

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

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

4.4 Application

Applications communicate with the global controller using a message queue, the per-app controller using function calls, and the CPU scheduler via system calls. There are four global controller calls: AddTask (add application to the global list), DeleteTask (remove application from the global list), GlobalAddConfigs (send different configurations, along with long-term resource demands[2]), and GlobalGetAllocation (get allocated resources). There are two per-app controller calls: FrameResourceDemands (send the next frame's resource demands for different configurations) and FrameGetConfig (get the application configuration for the next frame). There are five CPU scheduler calls as discussed in Section 4.3.

We believe the above additions can be made rather easily by application developers as the points of insertion are well defined: beginning and end of the application (for the calls to the global controller and the EnterSrt and ExitSrt calls to the scheduler) and the beginning and end of each frame (for the calls to the per-app controller and remaining calls to the scheduler). Looking forward, recent open multimedia standards such as OpenMAX (17) signify a trend where information needed by GRACE will become available (if not via the application directly then via calls to the middleware).

There are two ways of handling non-GRACE and legacy applications. The first approach is to run them in a non-real time partition. As long as that partition is judiciously allocated, the non-GRACE application would not be worse off than without GRACE. Alternatively, we can use passive monitoring at the OS level to find the resource demands and task boundaries of legacy applications (9). The GRACE-2 interfaces will need to be enhanced to accommodate such monitoring.

5 Experimental Methodology

Energy measurement: We use an Agilent 66319D sampling power supply to measure the energy consumed by the entire system. The measurements were done with the display brightness set to level 3 (0 is minimum). The wireless card was turned off, the laptop battery was removed, and the only applications running were from the experimental workload. All other parts of the system (e.g., hard drive) were on. The network energy used was calculated using the model in Section 2.2, and was added to the above measured energy to give the total system energy in Section 7.3.

The form-factor and packaging of a laptop makes it difficult to measure the energy usage of individual components such as CPU, GPU, and memory. Since we cannot isolate the CPU energy in our measurements and since the CPU and the network are the targets of our energy adaptations, our first set of results (Section 7.1) are based on modeled CPU (+network) energy, using the model in Section 2.1.

Applications and input streams: We study workloads consisting of various combinations of an H.263 video encoder and decoder, and a speech encoder and decoder (from Speex project (31)). The video encoder is adaptive as discussed in Section 2.3 while the other applications are non-adaptive. Table 3 summarizes the input streams for these applications. The video streams are standard H.263 test sequences and are freely available on the Internet. They have been chosen to represent a spectrum in inter-frame computation variability (the first three sequences have lower variability compared to the next three). We use QCIF size frames for the video encoder and CIF size for the video decoder.[3] The audio streams are in 16-bit PCM format and were also downloaded from the Internet.

Workloads: We evaluate our system with four distinct combinations of the above applications to represent real-world workloads. Two of these workloads are run on two different streams each. This gives a total of six evaluated workloads, summarized in Table 4. Workloads 1 and 2 consist of two video encoders representing a remote sensing application where two video streams need to be encoded and transmitted simultaneously (e.g., Mars rover). Workload 1 is run with low variability video sequences while workload 2 uses high variability sequences. Workloads 3 and 4 represent audio-less video-teleconferencing, with a video encoder and a video decoder, running low and high variability sequences respectively. Workload 5 is a video-teleconferencing setup with audio and video, and consists of a video encoder, audio encoder, video decoder, and audio decoder. Finally, workload 6 represents a setup where the user is involved in a video-teleconference while also watching another streaming video. It could also be considered to represent a case where the video teleconference is between three sites, with each site sending one stream and receiving two streams. Thus, the workload consists of one video and one audio encoder and two video and audio decoders.

Resource constraints or scenarios: To study the effect of different types of resource constraints (i.e., system load and/or resource availability), we use different periods (frame rates) for our workloads and different values of the available network bandwidth. We create four scenarios of resource constraints, depending on whether the CPU or network is constrained or not:

Scenario 1, only CPU constrained or C*:* We set the application period so that the application configurations[4] that do the most computation (i.e., the most compression for the video encoder) are unable to run on our system (i.e., they would require a higher frequency than that supported). The network does not pose a constraint in this scenario – we set enough available bandwidth to send/receive the bytes produced by the application config-

[2] As discussed earlier, in our implementation, the long-term resource demands are obtained through profiling.

[3] Our current platform cannot provide real-time CIF encoding at 30 fps; we therefore use QCIF for the encoder. We used CIF for the decoder (assuming the streaming host has enough computation power) because the computation demand for QCIF decoding is very low and we wanted another application somewhat comparable to the video encoder.

[4] Multiple configurations apply only to the adaptive video encoder.

uration that does the least compression.

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

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

Scenario 4, unconstrained U*:* In this case, we pick the period and bandwidth such that none of the application configurations are either CPU or network constrained.

Table 5 summarizes the workloads, their periods, and available bandwidth for each scenario that we study. (For example, N.1 implies workload 1 from Table 4 in the "only network constrained" scenario.) Since workloads 3 to 6 require relatively low computation, they cannot be CPU constrained on our platform and so do not have entries under the C or B category. For simplicity, for a given scenario and workload, we use the same period for both applications, but for generality, the applications start with an arbitrary lag between them. Note that the speex codecs are run with a 20ms period, as specified in the speex codec documentation (31). Each run includes between 150 to 500 frames for each application.

6 Overheads

We next summarize the overheads from various parts of our implementation (measured using a methodology similar to that from GRACE-1 (33)). The overheads are reported in terms of the number of CPU cycles (which is virtually independent of frequency). For comparison, note that the number of CPU cycles for encoding a typical video frame is of the order 10 to 25 million cycles.

Global vs. Per-app Control: Figure 4 compares the cost for global and per-app control. For global, we measured the elapsed CPU cycles for the global optimization algorithm by Moser et al. (Section 3.1). To study how the optimizer scales with the number of applications, we report results for systems containing one to ten applications. The system with ten applications may represent, for example, a teleconference system involving five sites (a video and an audio decoder for each of the four remote sites, and a video encoder and an audio encoder for the local site). Note that our numbers do not include any profiling cost incurred for making predictions for long-term resource usage for the global optimizer (discussed further below).

To measure the cost of per-app control, for each frame of the foreman sequence, we measured the elapsed cycles for the per-app control algorithm (Section 3.2). We report

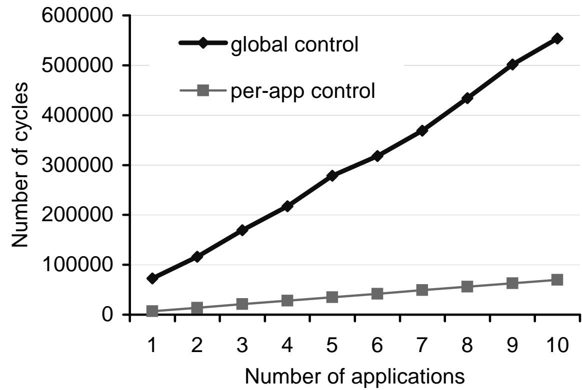


Figure 4: Overhead for global and per-app control.

the (per-frame) elapsed cycles averaged over the entire sequence and multipled by the total number of applications in the system (to obtain the per-invocation cost for all applications). Note that this measurement includes the full cost of the adaptation, including the cost of predicting the resource usage for the next frame.

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

First, we note that our global algorithm is optimized for the system we study. Specifically, we do not explore the full cross-product of the space of CPU and application configurations – we are able to assume a common frequency for all applications because of the special frequency-energy curve. However, this relationship may not be true for other adaptations such as architecture adaptations that are becoming increasingly common in hardware (14). Further, we also do not consider an adaptive network layer, which will further increase the complexity of the global algorithm. As the number of possible adaptive layers, adaptive components within each layer, and the number of adaptive states within each component increases, the overhead of the global optimizer will increase much faster than that of the per-app controller.

Finally, when considering the overhead of global control, we must also consider overheads for the required prediction of the long-term resource usage. In our system, we perform global adaptation when an application joins or leaves the system, which is a relatively rare event. Therefore, the profiling required for predictions can be done on-line (while running the system in sub-optimal configurations); the time spent profiling is a negligible fraction of the overall time that an application runs. However, for more frequent global adaptation, on-line profiling at sub-optimal

Video	Description	Audio	Description
salesman	talking head	lpcqutfe	sentence read by a boy
paris	talking heads	female	sentence read by a woman
carphone	talking head	male	sentence read by a man
foreman	talking head	clinton	speech by Clinton
football	football game		
buggy	buggy race		

Table 3: Input streams.

#	**Applications**	**Inputs**
1	video (encode, encode)	salesman, carphone
2	video (encode, encode)	foreman, buggy
3	video (encode, decode)	carphone, paris
4	video (encode, decode)	buggy, foreman
5	video (encode, decode) audio (encode, decode)	carphone, paris clinton, lpcqutfe
6	video (encode, decode, decode) audio (encode, decode, decode)	foreman, carphone, football female, clinton, male

Table 4: Workloads evaluated.

Constraint	Only **C**PU		Only **N**etwork					
Workload	C.1	C.2	N.1	N.2	N.3	N.4	N.5	N.6
Period (Fps)	33	30	30	20	30	30	30	30
Bandwidth (Mbps)	11	11	1.2	2	2	4.4	2.1	6.7
Constraint	**B**oth		**U**nconstrained					
Workload	B.1	B.2	U.1	U.2	U.3	U.4	U.5	U.6
Period (Fps)	33	30	30	26	30	30	30	30
Bandwidth (Mbps)	2	3.3	11	11	11	11	11	11

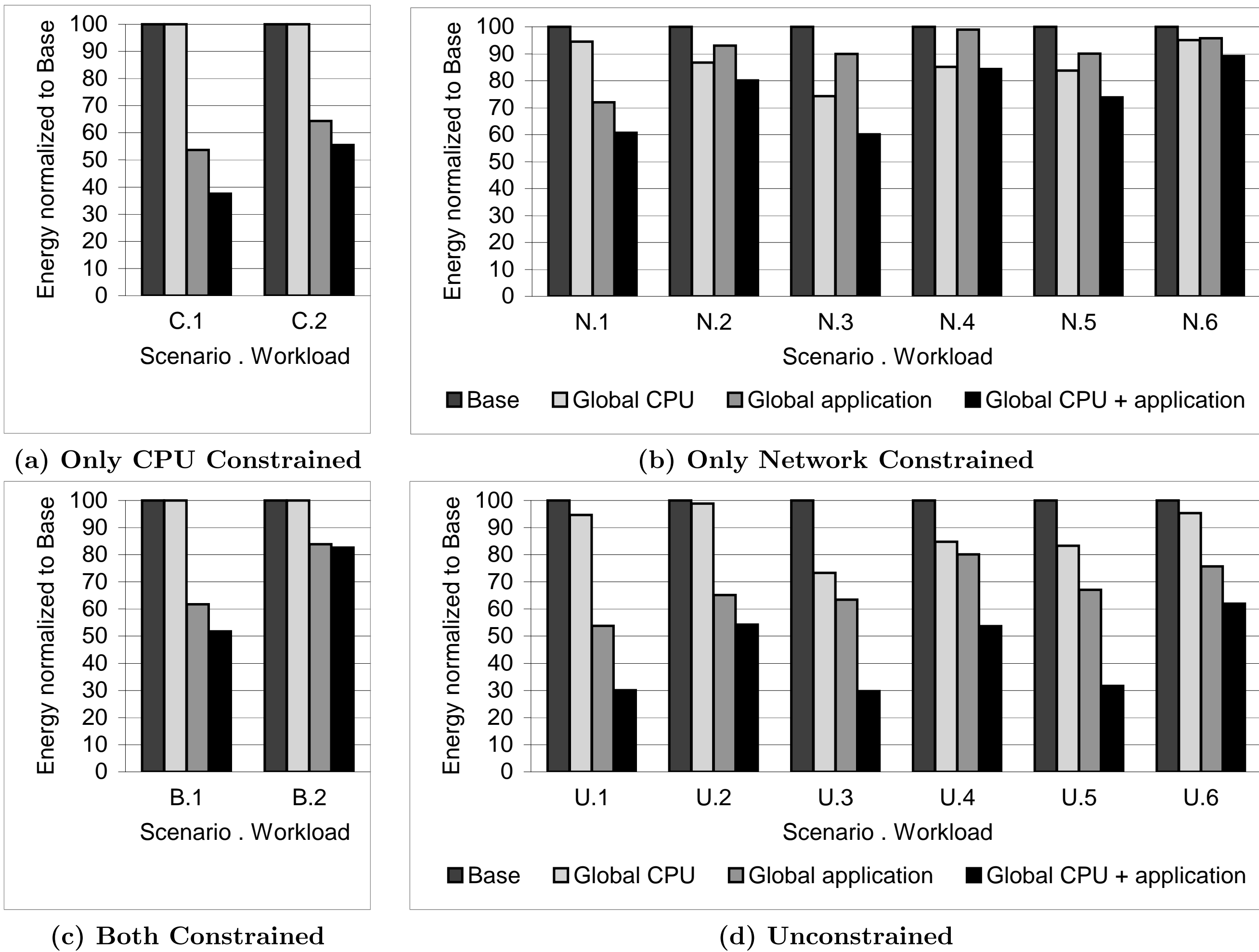
Table 5: Scenarios evaluated.

configurations can be too expensive. We cannot directly use past history because we only have the history for the application configuration that was chosen for a frame; the optimizer needs to make predictions for all the configurations. We could potentially use the same predictors as used in the per-app adaptation to predict the behavior of the next frame, and keep track of the outputs of these predictors over several frames. Whether this is feasible requires a study of how well these predictors perform for a span of several frames. Our results show that per-app adaptation is much simpler, and gives significant benefits over streams of several hundred frames.

Other overheads: We measured the average cycles used by each of the 5 system calls made by the video encoder while running foreman. Each call took less than 2,700 cycles, which is negligible overhead (e.g., less than 0.1% of encoding a video frame). The SRT scheduler requires less than 500 cycles per application. The high resolution timer it uses requires between 1,000 to 1,500 cycles for set up. We also found the budget sharing overhead to be negligible (consistent with (3)). Thus, the total scheduler overhead is small. For DVFS, the Pentium M processor decouples the voltage and frequency transition, thereby allowing voltage to be changed while executing instructions. The DVFS overhead is around 10 us (15) (except for transition to 600 MHz, where we found the overhead to be around 400 us), making intra-frame frequency transition feasible for many applications.

7 Energy Savings

This section quantifies the energy benefits of hierarchical adaptation. Section 7.1 presents the energy savings in the CPU and network subsystem since those are the targets of this work. Section 7.2 provides detailed analysis of these results. Section 7.3 presents the savings for the entire system. Section 7.4 quantifies the benefits of budget sharing. Since the primary benefit of budget sharing is in reducing missed deadlines, we discuss all deadline misses in Section 7.4 .



(a) Only CPU Constrained

(b) Only Network Constrained

(c) Both Constrained

(d) Unconstrained

Figure 5: CPU+network energy benefits from global adaptation for different resource constraints. For each workload, the leftmost bar shows energy for a non-adaptive base system. The next three bars show energy for global CPU adaptation, global application adaptation, and global CPU+application adaptation respectively. All adaptive systems include internal scheduler adaptation. The energy for each system is normalized to the base system (leftmost bar).

7.1 CPU and Network Energy Savings

Benefits of global adaptation: For reference, we first briefly summarize the benefits of global adaptation over the non-adaptive base system (Base). For each scenario/workload, Figure 5 gives the normalized energy consumption of three systems with global adaptation – global CPU adaptation, global application adaptation, and global CPU+application adaptation. All adaptive systems include internal scheduler adaptation. The energy is normalized to that of Base (which is assumed to be 100 units).

In the CPU constrained scenarios (C and B), Base and global CPU adaptation are unable to meet the computation requirements of the base configuration of the video encoder. Global application adaptation, and global CPU+application adaptation, however, change the application configuration to use less computation, and are able to successfully run these cases. Their energy consumption shown is normalized with respect to the energy consumed by Base running in “best effort” mode.

Overall, we see benefits from both CPU and application adaptation, with the best savings coming from the combination. In the network constrained cases (N and B), global CPU adaptation, global application adaptation, and global CPU+application adaptation respectively save an average of 10%, 14%, and 27% over Base. For the remaining 8 scenarios (C and U), global CPU adaptation, global application adaptation, and both global CPU and global application adaptation respectively save an average of 9%, 35%, and 56% over Base.

To see how global adaptation saves energy, Table 6 shows the configurations that global CPU+application adaptation chooses (the application configurations are roughly ordered by the amount of compression performed – configuration 0 is the highest compression). Note that for workloads 3 to 6, only the video encoder supports multiple application configurations. We find that global chooses a variety of configurations depending on the resource constraints.

Constraint	Only CPU		Only network					
Workload	C.1	C.2	N.1	N.2	N.3	N.4	N.5	N.6
CPU(MHz)	718	960	876	943	704	932	797	1174
App 1	15	15	11	1	1	1	1	1
App 2	15	15	1	2	-	-	-	-
Constraint	Both		Unconstrained					
Workload	B.1	B.2	U.1	U.2	U.3	U.4	U.5	U.6
CPU(MHz)	841	1260	646	832	485	710	578	909
App 1	15	1	15	15	15	15	15	15
App 2	7	3	15	15	-	-	-	-

Table 6: Configurations chosen by global CPU+application. Since only the video encoders are adaptive, the number of applications supporting multiple configurations in a workload is only one (workloads 3 to 6) or two (workloads 1,2).

Comparing the brute force and the more practical optimizer, we find that the practical solver provides very similar energy benefits. Nevertheless, since our focus is on the benefits of per-app adaptation, we henceforth use the brute force optimizer to give global the best showing.

Benefits of per-app adaptation: Figure 6 illustrates the energy benefits in the CPU-network subsystem of per-app application adaptation. For each workload, the leftmost bar shows a system with global adaptation in the application, CPU, and scheduler. The next three bars show systems that incorporate this global adaptation and additionally have per-app CPU adaptation (second bar), per-app application adaptation (third bar), and both per-app application and per-app CPU adaptation (the last bar, which represents GRACE-2). The energy of all systems is normalized to that consumed by the system with only global adaptation (the first bar). We find that GRACE-2 consumes less than or virtually the same energy as a system with only global adaptation for all the scenarios and workloads.[5] The magnitude and source of the benefits depends on the magnitude and nature of the resource constraints in the system. The largest benefits from GRACE-2 over the global-only system come in the network constrained cases (scenarios N and B). For the 8 such cases studied here, the energy savings range from 18% to 36%, with an average of 27%. The savings in the other 8 cases (scenarios C and U) are a more modest 0% to 11% with an average of 6%. We next discuss the contributions of the CPU and application adaptations to these benefits.

Adding per-app adaptation in the CPU to a system with global adaptation provides discernible benefits in all cases (except U.3 and U.5 as discussed previously). The benefits from CPU adaptation are modest relative to those seen for DVFS in prior work due to the sub-linear relationship between frequency and voltage reductions in current processors (Section 2.1).

Figure 6 shows that adding per-app application adaptation to a system with global adaptation can result in significant energy benefits. The benefits remain significant regardless of whether the base global system contains per-app CPU adaptation (second bar) or not. Relative to a system with only global adaptation, the energy savings from adding per-app application adaptation range from 6% to 18% with an average of 13% for the network constrained scenarios (N and B). Relative to a system with both global and per-app CPU adaptation, the energy savings from adding per-app application adaptation range from 12% to 32% with an average of 22% across the N and B scenarios.

It is noteworthy that adding only per-app CPU adaptation to global adaptation gives modest benefits. In contrast, combining CPU and application adaptation at the per-app level gives more than additive benefits in some cases, resulting in quite significant overall savings of per-app adaptation relative to a system with only global adaptation.

7.2 Analysis

Next, we analyze the reasons for the above results in each of the scenarios in more detail. Figure 7 shows the network, CPU, and total energy for each application configuration for a specific frame of one of the video encoders in workload 2 for each of the four scenarios. The application configurations are ordered in increasing order of bytes generated. Only those configurations where the CPU cycles for the configuration decrease with increasing bytes are shown, since the remaining configurations are clearly sub-optimal. Configurations that do not meet the required constraints are also not shown. (For this reason, the same number on the x-axis may represent different actual configurations in the four graphs.) On each curve for total energy, we mark the application configuration chosen by the global-only system and the GRACE-2 system, along with the frequency chosen.

[5] In a few cases (U.3 and U.5), GRACE-2 is very slightly worse than the global-only system. In these cases, the global-only system already picks the lowest energy configuration supported by the system; i.e., the lowest CPU frequency and the lowest compression video encoder configuration. Thus, GRACE-2 cannot do any better than GRACE-1. For a few frames, the resource usage predictors of GRACE-2 turn out to be slightly more conservative, making it pick slightly higher frequency than GRACE-1 and showing very slightly higher overall energy. For other frames, GRACE-2's resource usage predictions are (correctly) lower than GRACE-1's, but the system does not support a lower frequency to convert these better predictions into energy savings.

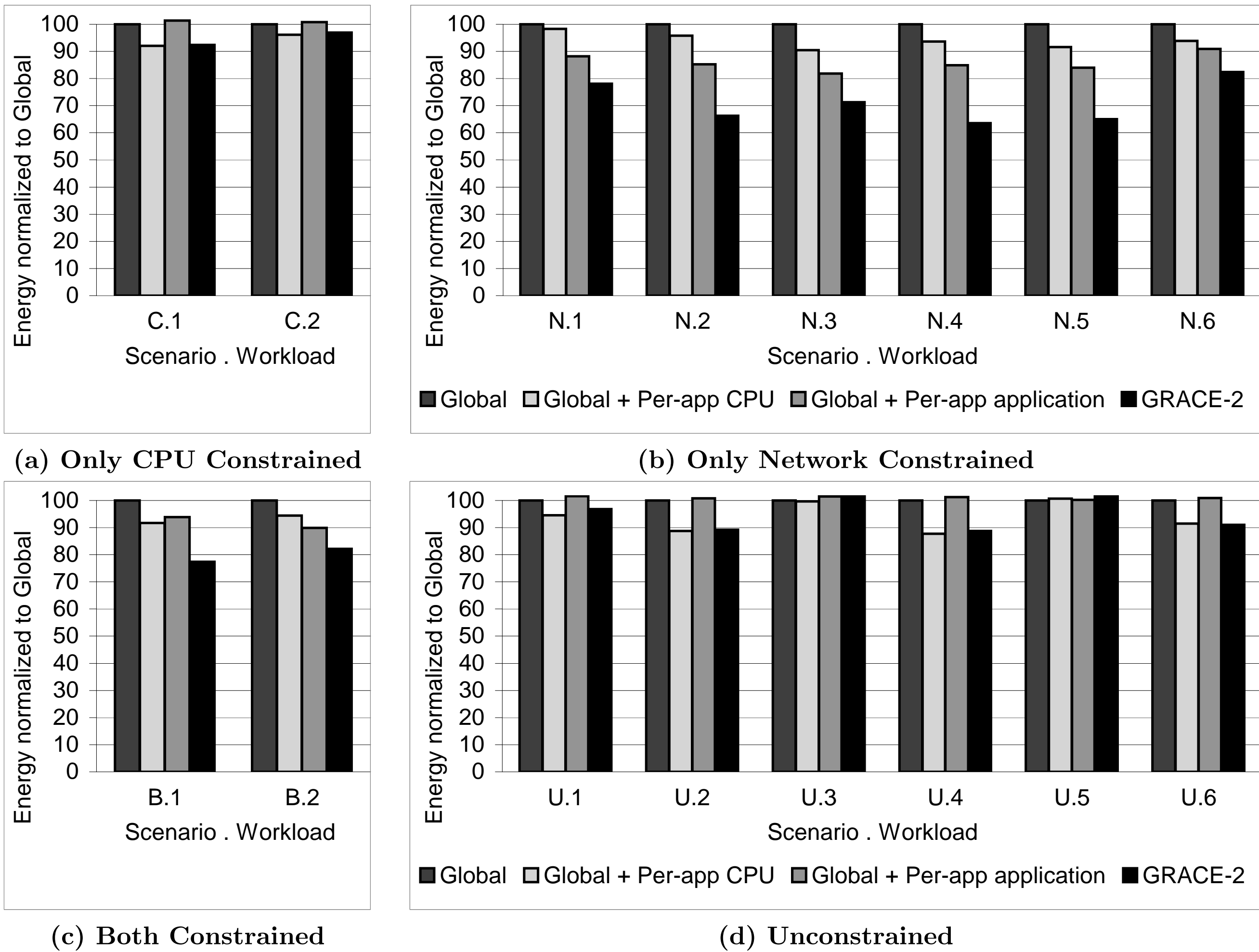


Figure 6: CPU+network energy benefits from per-app application adaptation for different resource constraints. For each workload, the leftmost bar shows energy for a system with global adaptation in the CPU, application, and scheduler. The next three bars include this global adaptation as well as per-app CPU adaptation, per-app application adaptation, and both per-app CPU and per-app application adaptation (i.e., GRACE-2) respectively. The energy for each system is normalized to the system with only global adaptation (leftmost bar).

Recall that network energy is simply the product of (bandwidth dependent) energy/byte and bytes generated. For CPU energy, we first need to determine the frequency at which the frame will complete the required cycles within the time allocated by the global adaptation. Network energy increases going from left to right due to increasing byte count, while CPU energy decreases. In our graphs, for the most part, CPU energy is dominant, and so we find that the total energy curve primarily follows the CPU energy.

We can now analyze the four cases. We start with the unconstrained case (part (d)). Both the global-only system and the GRACE-2 system are able to pick the configuration with the least computation (rightmost), and so the most energy efficient. Thus, GRACE-2 does not benefit from application adaptation, compared to the global-only system. However, GRACE-2 does benefit from CPU adaptation because of its ability to better predict the cycle count and use a lower frequency.

Next consider the network constrained case (part (b)). The minimal energy configuration is the rightmost one shown on the graph, and picked by GRACE-2. However, the global-only system is not able to pick that configuration because its estimate of the byte count for that configuration is too high for the bandwidth available. The global-only system is forced to make a conservative byte count estimate for this frame because its estimate must be high enough to accommodate the bandwidth requirement of all (or at least 95% of) future frames (until the next global adaptation). Thus, the global-only system is forced to pick a configuration that is less energy efficient than needed for the frame shown in the figure. GRACE-2 on the other hand must make a prediction that is adequate for just the frame under consideration. It is able to correctly predict the low byte count requirement for that frame, and correctly determine that the bandwidth requirement of the

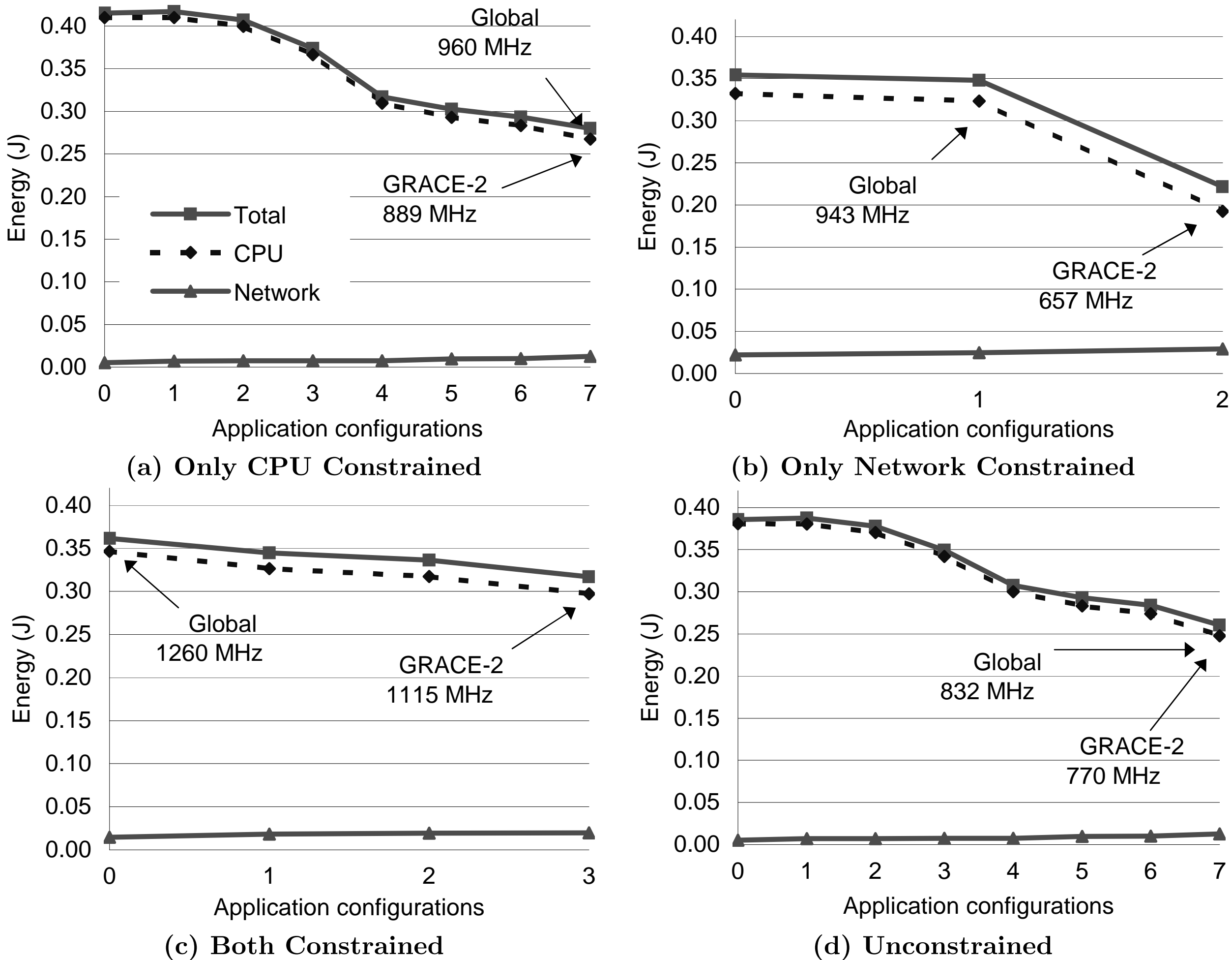


Figure 7: Analysis of the results.

rightmost configuration will be met with the given network constraint. This configuration has much lower energy (since it performs less compression and takes less CPU energy), enabling significant savings from GRACE-2. The other cases can be similarly analyzed.

7.3 System-Wide Energy Savings

We next discuss (measured) system-wide energy savings of GRACE-2 over a system with only global adaptation.[6] Figure 8 shows the system-wide energy for the global-only system and for GRACE-2, both normalized to that for Base. Again, we find that the addition of per-app adaptation to global adaptation is most beneficial in the network constrained scenarios (N and B). For these cases, we found that GRACE-2's per-app adaptation provides a system-wide energy benefit of 7% to 14% with an average of 10% (relative to only global adaptation). (For reference, the savings of the global-only system over Base for the N and B scenarios is 5% to 22%, average 13%.)

These savings are significant, considering that they are for the entire system including the display, disk, power-supply loss, and memory system; they are actual measured values; and they come from only adaptation of the CPU and application. (As reference, the one workload with multiple applications reported for GRACE-1 showed system-wide savings from hierarchical adaptation of only 3.8%, relative to global adaptation (33).)

[6]As explained in Section 7.2, the network energy is modeled, but is a very small part of the system energy.

7.4 Deadline Misses and Budget Sharing

The main benefit of budget sharing (i.e., the internal scheduler adaptation described in Section 2.4) is in reducing the number of deadline misses (including frame drops). Budget sharing has negligible (< 1%) effect on energy. GRACE-2 shows acceptable deadline misses (within 5%) for each application in each scenario/workload studied. Without budget sharing, the deadline miss ratios are high (up to 23%) for several cases. Thus, budget sharing is effective and critical for our system.

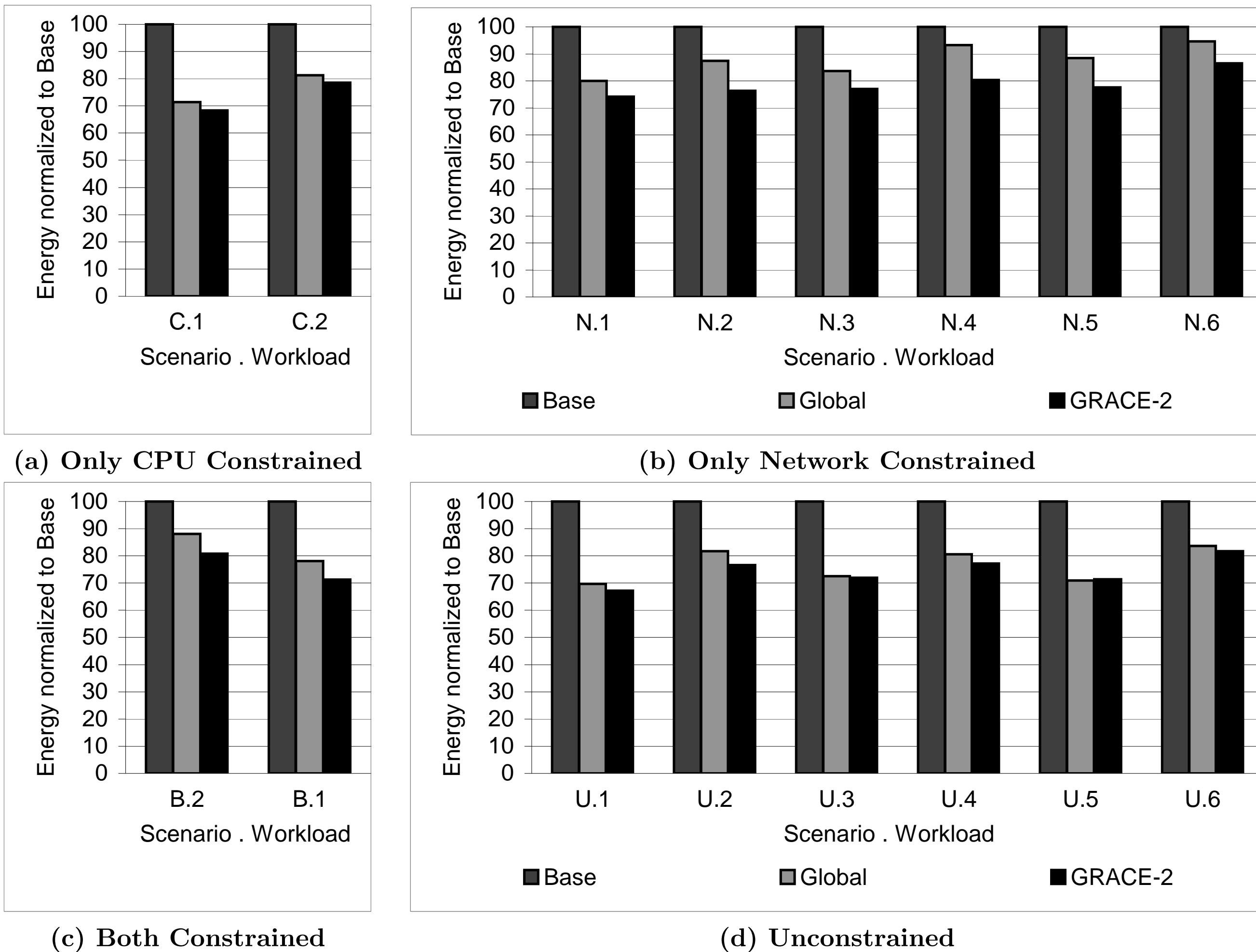


Figure 8: System-wide energy savings for different resource constraints.

8 Related Work

There has been a large amount of work on energy and bandwidth driven adaptations and resource allocation that is relevant to this work. This includes CPU adaptation with and without coordination with a real-time scheduler (e.g., (2; 9; 22; 23; 25; 29; 34)), adaptation of one or more applications with and without OS/middleware support (e.g., (7; 8; 10; 11; 19; 21; 24)), and single-layer or cross-layer adaptation or resource allocation with only global control supporting multiple applications (e.g., (12; 35; 26)) or only per-app control supporting a single application (e.g., (28)). The focus of this work, however, is on hierarchical adaptation control in a cross-layer adaptive system, and more specifically on fine-grained (per-app) application adaptation. None of the above systems exhibit this property.

The systems most closely related to the hierarchical adaptation of GRACE-2 are Fugue (6) and GRACE-1 (33; 32), which we discuss in more detail next.

Fugue proposed adaptation at multiple time scales for wireless video (6). This is one of the key features of GRACE-2's hierarchical control. However, Fugue differs from GRACE-2 in the following important ways. First, it considers only one application running. Second, it is based on the insight that different types of adaptations work on different time scales; e.g., application quality control must occur at a coarser time scale than network transmission power control. GRACE-2's global and per-app controllers consider the same set of adaptations, but for different purposes – the former uses them for resource allocation among multiple applications while the latter does the actual adaptation. Incorporating adaptations that inherently work at different time scales can be viewed as an orthogonal issue – our system incorporates these as well, but that is not the focus of this work.

The goal of GRACE-1 was to demonstrate the benefits of coordinated cross-layer adaptation (33; 32). GRACE-1 therefore primarily focused on global adaptation in the CPU, application, and scheduler, and included only very preliminary support and experimental evaluation for hierarchical adaptation (using only internal CPU scheduler related adaptations). Further, GRACE-1 considered only the resource constraint of execution time, and did not consider any network bandwidth related resource usage. The

lack of any network awareness resulted in very modest benefits from the hierarchical adaptation support in GRACE-1 while running multiple applications. The fundamental difference between the GRACE-1 and GRACE-2 systems is that GRACE-2 is network-aware – it adds a network bandwidth constraint in the global and per-application controller and considers global and per-app application adaptations that are driven by the tradeoff in CPU time and network bandwidth usage. Network awareness is critical for the mobile environment where this work is targeted and results in different conclusions. Specifically, our results show that network awareness gives significant energy benefits from hierarchical adaptation in GRACE-2, providing the first demonstration of large energy benefits of hierarchical adaptation on a multimedia system implementing multiple applications, adaptations, and constraints.

9 Conclusions

The GRACE project balances the scope and frequency of energy saving adaptations in multiple layers through a hierarchical approach, where expensive and infrequent global adaptation allocates resources among applications based on long-term predictions, and inexpensive per-app control seeks to make the energy-optimal use of these resources through localized short-term predictions and cross-layer adaptations.

This paper presents results from the second generation prototype, GRACE-2. Specifically, it shows that per-app application adaptation provides significant benefits over and above global adaptation when the network bandwidth is constrained. These benefits are seen both with and without per-app CPU adaptation. For example, the energy savings in the CPU+network from adding per-app application adaptation to a system with global adaptation and per-app CPU adaptation were seen to be up to 32% (average 22%). Interestingly, when both per-app CPU and per-app application adaptation are added to a system with global adaptation, the combined benefits are more than additive.

To our knowledge, this work is the first to demonstrate the benefits from per-app application adaptation control over and above global control. It is also the first to demonstrate significant benefits from hierarchical adaptation on a real multimedia system implementing multiple applications, adaptations, and constraints. Given the low overhead of per-app control and the relatively low added system implementation complexity over a system with global control, the benefits achieved seem worthwhile to exploit.

Our ongoing work is incorporating an adaptive network layer that responds to variations in network bandwidth, and is also exploring other possible application adaptations including those that affect user perception. There are several other interesting avenues of future work. We would like to explore adaptations and resource tradeoffs for other compute- and network-intensive applications that would be important for mobile workloads; e.g., graphics. We would also like to integrate other adaptations within the CPU (e.g., architectural adaptations) and adaptations of other hardware components as well (e.g., main memory, display, and disk). Recent work has shown how to perform joint adaptation of closely coupled hardware components such as processor and main memory for general-purpose applications (Li et al.), and we wish to integrate that work with our cross-layer adaptation framework. We would also like to explore adaptation with multithreaded/synchronized applications and on multicore systems; these are currently open problems. We also wish to extend our implementation in various ways to make it more complete; e.g., incorporate applications where performance may not be proportional to frequency (these will require alternate performance models and/or additional profile data); incorporate invoking global adaptation when an application undergoes a large (and infrequent) change in its resource requirement or in the network bandwidth available; and run non-real time, non-GRACE applications in a non-real time partition.

REFERENCES

[1] Anzinger, G. (2001). High res posix timers. http://sourceforge.net/projects/high-res-timers/.

[2] Aydin, H. et al. (2001). Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proc. of Real-Time Systems Symposium*, pages 95–105.

[3] Caccamo, M. et al. (2000). Capacity sharing for overrun control. In *Proc. of Real-Time Systems Symposium*, pages 295–304.

[4] Chu, H. H. and Nahrstedt, K. (1999). CPU service classes for multimedia applications. In *Proc. of IEEE Int. Conf. on Multimedia Computing and Systems*, pages 296–301.

[5] Cisco (2004). Cisco Aironet 350 Series Client Adapters Datasheet. http://www.cisco.com/en/US/products/hw/wireless/ps4555/ps448/.

[6] Corner, M. et al. (2001). Fugue: time scales of adaptation in mobile video. In *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, pages 75–87.

[7] de Lara, E. et al. (2002). HATS: hierarchical adaptive transmission scheduling for multi-application adaptation. In *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, pages 100–114.

[8] Efstratiou, C. et al. (2003). A platform supporting coordinated adaptation in mobile systems. In *Proc. of 4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 128–137.

[9] Flautner, K. and Mudge, T. (2002). Vertigo: Automatic performance-setting for linux. In *Proc. of Symposium on Operating Systems Design and Implementation*, pages 105–116.

[10] Flinn, J. et al. (2001). Reducing the energy usage of office applications. In *Proc. of Middleware*, pages 252–272.

[11] Flinn, J. and Satyanarayanan, M. (1999). Power-Scope: A tool for proling the energy usage of mobile applications. In *Proc. of 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10.

[12] Gopalan, K. and Chiueh, T. (2002). Multi-resource allocation and scheduling for periodic soft real-time applications. In *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, pages 34–45.

[13] Hughes, C., Kaul, P., Adve, S., Jain, R., Park, C., and Srinivasan, J. (2001a). Variability in the execution of multimedia applications and implications for architecture. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*, pages 254–265.

[14] Hughes, C., Srinivasan, J., and Adve, S. (2001b). Saving energy with architectural and frequency adaptations for multimedia applications. In *Proc. of 34th Intl. Symp. on Microarchitecture*.

[15] Intel (2003). Intel Pentium M Processor Datasheet. http://www.intel.com/design/mobile/datashts/25261203.pdf.

[16] Ishihara, T. and Yasuura, H. (1998). Voltage scheduling problem for dynamically variable voltage processors. In *Proc of Intl. Symp. on Low Power Electronics and Design*, pages 197–202.

[17] Khronos (2006). OpenMAX - The Standard for Media Library Portability. http://khronos.org/openmax/.

[18] Krantz, S., Kress, S., and Kress, R. (1999). *Jensen's Inequality*. Birkhauser.

[Li et al.] Li, X., Gupta, R., Adve, S. V., and Zhou, Y. Cross-component energy management: Joint adaptation of processor and memory. *ACM Trans. on Architecture and Code Optimization*. Accepted subject to minor revisions.

[19] Mesarina, M. and Turner, Y. (2002). Reduced energy decoding of MPEG streams. In *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, pages 202–213.

[20] Moser, M. et al. (1997). An algorithm for the multidimensional multiple-choice knapsack problem. In *IEICE Trans. on Fundamentals of Electronics*, pages 582–589.

[21] Noble, B. et al. (1997). Agile application-aware adaptation for mobility. In *Proc. of Symposium on Operating Systems Principles*, pages 276–287.

[22] Pering, T. et al. (2000). Voltage scheduling in the lpARM microprocessor system. In *Proc of Intl. Symp. on Low Power Electronics and Design*, pages 96–101.

[23] Pillai, P. and Shin, K. G. (2001). Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of Symposium on Operating Systems Principle*, pages 89–102.

[24] Poellabauer, C. et al. (2002). Cooperative run-time management of adaptive applications and distributed resources. In *Proc. of 10th ACM Multimedia Conference*, pages 402–411.

[25] Quan, G. and Hu, X. (2001). Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Design Automation Conference*, pages 828–833.

[26] Rusu, C. et al. (2002). Maximizing the system value while satisfying time and energy constraints. In *Proc. of Real-Time Systems Symposium*, pages 246–257.

[27] Sachs et al. (2003a). GRACE: A Cross-Layer Adaptation Framework for Saving Energy. *SIDEBAR in IEEE Computer*, pages 50–51.

[28] Sachs, D. et al. (2003b). Adaptive video encoding to reduce energy on general-purpose processors. In *Proc. of Intl. Conference on Image Processing*.

[29] Simunic, T. et al. (2001). Dynamic voltage scaling and power management for portable systems. In *Design Automation Conference*, pages 524–529.

[30] Vardhan, V. et al. (2005). Integrating fine-grained application adaptation with global adaptation for saving energy. In *Proc. of the 2nd International Workshop on Power-Aware Real-Time Computing (PARC)*.

[31] Xiph.org (2003). Speex. http://www.speex.org/.

[32] Yuan, W. et al. (2003). Design and evaluation of cross-layer adaptation framework for mobile multimedia systems. In *Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, pages 1–13.

[33] Yuan, W. et al. (2006). GRACE: Cross-Layer Adaptation for Multimedia Quality and Battery Energy. *IEEE Trans. on Mobile Computing*.

[34] Yuan, W. and Nahrstedt, K. (2003). Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proc. of Symposium on Operating Systems Principle*, pages 149–163.

[35] Zeng, H., Fan, X., Ellis, C., Lebeck, A., and Vahdat, A. (2002). ECOSystem: Managing energy as a first class operating system resource. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132.