# Energy Driven Application Self-Adaptation at Run-time

Jorgen Peddersen, Sri Parameswaran
School of Computer Science and Engineering, National ICT Australia
The University of New South Wales, Sydney, NSW 2052, Australia
{jorgenp,sridevan}@cse.unsw.edu.au

UNSW-CSE-TR-0619
September 2006

**Abstract**

Until recently, there has been a lack of methods to trade-off energy use for quality of service at run-time in stand-alone embedded systems. Such systems are motivated by the need to increase the apparent available battery energy of portable devices, with minimal compromise in quality. The available systems either drew too much power or added considerable overheads due to task swapping. In this paper we demonstrate a feasible method to perform these trade-offs. This work has been enabled by a low-impact power/energy estimating processor which utilizes counters to estimate power and energy consumption at run-time. Techniques are shown that modify multimedia applications to differ the fidelity of their output to optimize the energy/quality trade-off. Two adaptation algorithms are applied to multimedia applications demonstrating the efficacy of the method. The method increases code size by 1% and execution time by 0.02%, yet is able to produce an output which is acceptable and processes up to double the number of frames.

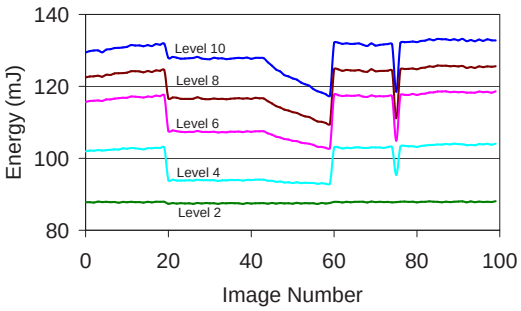



Figure 1: Quality Comparison

# 1 Introduction

Power and energy consumption are becoming the most dominant bottlenecks constraining today's embedded systems. As more systems become mobile, the reliance on energy consumption is particularly important due to battery constraints and heat dissipation. Batteries have a very limited amount of energy they can supply and when power peaks over certain levels, the capacity of the battery drops more rapidly than usual[1]. Recently, many methods have been designed to manage power at all levels of design. These methods can be split into static and dynamic methods.

Static power management involves predicting, simulating or profiling applications to record their performance and optimize power/performance trade-offs to cater for the data set. All decisions are made before run-time, so worst case conditions are typically used to guarantee constraints are met.

However, many applications currently executed on embedded systems cannot be analyzed statically due to the input data being unknown prior to run-time. For example, multimedia and network applications alter their operation significantly depending on what sort of data is used. Due to such altering conditions, dynamic methods are needed that use run-time feedback to adapt an application while it is executing to suit the situation.

Figure 1 demonstrates the energy use of various images encoded through a JPEG encoder at different quality levels (higher values indicate a higher quality level). These images were taken from several image sequences, so consecutive images are similar and energy consumption is also similar for these images. However, there are some places where consecutive images differ significantly (19→20, 59→60 and 74→75→76) and changes in the energy consumption can be seen between the different image types. Note also that as the quality level decreases, the energy consumption decreases. This observation enables a tradeoff between qual-

ity and energy consumption.

Embedded systems which adapt dynamically can be advantageously deployed in numerous environments. Military devices (e.g. reconnaissance devices) often have to be left alone in the field to operate for set time periods before the energy supply can be replenished. During this time, the unit should perform at the highest possible quality while ensuring it does not exhaust its energy supply. Wireless networking nodes must be able to adapt to the traffic they are seeing and to the quality of the radio communications link, to similarly meet run-time deadlines and constraints.

Until recently, solutions to counter the run-time adaptation problem have been hindered by the lack of feasible methods to feedback information of the power and energy consumption of a system. New approaches have been proposed that allow an embedded system to estimate its own power consumption with minimal impact[2].

Run-time application adaptation has emerged as one solution to the problem of optimizing quality while conserving power and energy within acceptable levels at run-time. Application adaptation benefits greatly from the new availability of run-time power/energy consumption.

Previous approaches to solve problems of run-time power management have used Operating Systems (OS) to handle application adaptation and switching. These OS- based techniques often switch versions of an application in and out, causing large delays due to context changes[3]. OS-based approaches also use periodic reading points to gather data about executed energy and power. Hence, they are less aware of when key events in an application such as frame boundaries occur. Therefore, they are less able to attribute energy calculations to an individual frame of an image than an application based approach.

Our solution solves the problems of OS-based approaches by providing a methodology to design applications that alter their own functionality to suit the operating conditions. Applications that are parameterized already contain much of the necessary code to provide this run-time adaptation; the application is only modified to detect the operating conditions and change the parameters to suit.

This paper analyzes several methods that can be used to provide true run-time application adaptation. The methods can be applied to existing software to provide adaptation with minor impact to the run-time and code size. We also propose some adaptation algorithms to optimize the quality/energy trade-off that provide highly useful results with minimal run-time impact. We also give examples of several multimedia algorithms that utilize the adaptation procedures to perform dynamic power optimizations.

Thus, the contributions of this paper are:

- For the first time, we show a method to trade-off quality of service for energy with fine-grained low-overhead adaptations. This method is enabled by:
  - Identification of a system of techniques an application can utilize to adapt itself to dynamically optimize the quality/energy trade-off.
  - Proposal of two new algorithms for on-line run-time self adaptation of applications.

The remainder of this paper is organized as follows. Section 2 discusses previous work on the topic of run-time application adaptation. Section 3 gives some background information on the work which enables the methodology of this paper which is given in Section 4. Section 5 provides some example applications which have been modified to verify the applicability of the methodology while Section 6 concludes the paper.

## 2 Related Work

Reduction of power/energy consumption of embedded systems has been a major field of research for the past few decades[4], covering both software and hardware approaches. Some of the proposed hardware techniques include Dynamic Voltage Scaling[4, 5] and Adaptive Body Biasing[6, 7, 8]. Proposed software methods include Power-Aware Task Scheduling[9, 10], Power Macro Modeling[11] and Application Adaptation[12].

Where data input significantly influences energy/power consumption and is not known a priori, only dynamic run-time techniques can adequately manage the power consumption of systems. Several of these schemes require run-time values for power and energy consumption as an input to the system[12, 13, 14]. However, run-time power and energy consumption values have not been feasibly available at run-time until recently[2].

Powerscope[3] is a system designed to measure run-time power consumption and is used as a tool for power-aware computing. It was designed to be used with the tool Odyssey[12] to allow run-time application adaptation trade-offs to manage power consumption of a run-time system. However, this measurement procedure is power hungry and is not feasible for stand-alone embedded systems.

Odyssey is applied to mobile embedded systems in [15] and [16]. These works discuss several adaptation techniques (not energy driven) that can be performed at run-time and highlight the need for ‘agility’ when making adaptable mobile embedded systems. Agility refers to an application performing adaptation quickly with little overhead to minimize the costs of adaptation. These works also discuss

some fidelity/deadline based trade-offs that can be made to applications, and OS based/aided approaches to manage them.

Our work differs from the previous work by using run-time energy information to drive adaptations and by modifying the application to perform adaptation rather than relying on an operating system. This increases the 'agility' of the application, providing a solution that has little impact upon the performance of the system due to low hardware and software overheads.

# 3 Background

Dynamic power optimization techniques have been stymied by the lack of methods to provide run-time feedback of power and energy consumption. Recent work[2] has introduced CLIPPER, a methodology for accurately estimating power consumption in parallel with execution and feeding it back to the system. This low impact provision of power consumption data allows a new breed of dynamic power optimization techniques to be created. CLIPPER provides an energy calculation loop which can be called to provide an estimate of energy use since the last access with an error of less than 2%. The processor thus created exhibited a 4.9% increase in chip area and 3% increase in average power.

CLIPPER processors use counters attached via memory mapped I/O to count power consuming events that can be seen on the control signals. Contributions of events to the power consumption of the system are found by macro modeling [17]. Software loops are executed at run-time to read the counters and then use the contributions with the number of events to calculate power and energy consumption. These loops are executed fairly infrequently, (see Section 5) so they have minor impact upon the executing code.

For the purposes of this paper, we have further updated the integer only processor of [2] to also perform floating point instructions. This results in the addition of several events to measure power consumption of floating point instructions. Appropriate energy contributions for floating point events were estimated from a floating point functional unit to complete the energy model.

As the processor was originally based on the PISA SimpleScalar processor[18], we modified the SimpleScalar simulator to emulate the processor, adding pipeline stalls and cache simulators to create a cycle accurate model. The energy estimation counters described in [2] were also modeled by the simulator to allow quick calculation of energy consumption.

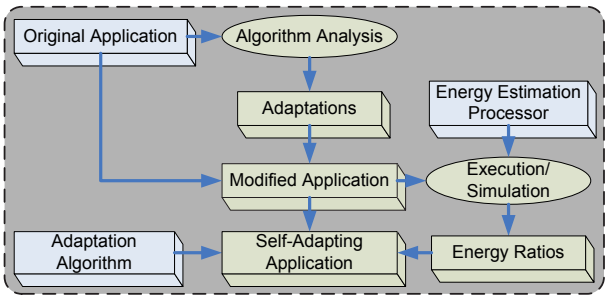



Figure 2: Methodology

# 4 Methodology

We define an environment for power measurement and run-time adaptation upon which we will build our application adaptation algorithms. This environment makes use of the function *calc_energy()* which reads the energy estimation counters and returns energy usage since the last call. Additionally, we add a function *calc_level(int energy)* which calculates the quality level at which further operation should perform based on the energy amount of the current and previous frames.

Figure 2 demonstrates the basic procedure for creating a self-adapting application from a given one. The algorithms of the application are first analyzed to discover which adaptation techniques (discussed below) can be applied. This will yield a set of adaptations which can be used to create quality levels. The application then needs to be simulated (or executed) on the target processor at each quality level to find a relative energy ratio for each quality level (see Section 4.2). These ratios are needed by a run-time adaptation algorithm that allows the application to choose its quality level while executing. The chosen adaptation algorithm and energy ratios are then added to the application to produce an application that is able to adapt its operation at run-time to suit energy constraints. The following sections discuss this procedure in greater detail.

## 4.1 Adaptation Techniques

This section describes several techniques to enable run-time application self adaptation. A self-adapting application is able to make use of a combination of these approaches to change its operation without requiring a context switch. The set of techniques described here is not exhaustive, although the methods allow much of the work to be done with little effort and can solve most adaptation problems (as demonstrated in Section 5).

The adaptation techniques described here rely on periodically receiving the energy calculation provided by the energy estimating processor. This is typically performed by executing the *calc_energy()* function whenever a recurring point in the main loop is reached (e.g. between each frame of a multimedia application). Adaptation algorithms(such as those described in section 4.2) compare the power level of current and previous frames with thresholds to determine whether adaptation is required.

When the algorithm determines a quality change is needed, the adaptation techniques discussed here utilize global variables and pointers to allow the application to adapt its operation to suit the current operating conditions. These variables can either be defined in global scope, or located in a *struct* (or similar variant) which is passed to all functions executed by the program. Examples in the following sections demonstrate loops in the main function of a program to clearly illustrate the techniques. In reality however, the adaptation code can be inserted in any suitable part of the application.

Most adaptations exploit parameters that already exist in the code as will be seen below. Many applications are already written to support different quality choices that can be made, which are usually input into the application at the start of execution. The adaptation techniques make use of these choices as well as demonstrating faster and simpler ways to achieve them. By utilizing these techniques, low impact application self-adaptation is made available.

### Conditional Operation

The most common technique for self-adaptation is conditional operation. This includes if/else chains and switch statements that use globally visual variables to allow the conditional statement/s to determine what code should be executed. This method is commonly used by designers when an application is written to utilize input parameters. Changing the variables controlling the conditional operation at run-time allows the application to change quality levels.

Figure 3 demonstrates how this system can be used in the energy aware environment, with adaptation code in **bold**. In this example, a global variable that stores the current mode to be used is stored in *i*. The *if-else* statement allows differing code to be run depending on the quality needed to meet constraints. Obviously, this method also applies to using other conditional statements such as *switch* statements.

Modifying existing code to add this functionality usually adds no further impact apart from the assignment of the variable to choose operating parameters. Applications that provide command line or configuration parameters to alter operation of the code are usually already written in this way.

```
int i = HIGH;

int main(void) {
   while(!end) {
      int energy;
      ...
      if (i == HIGH) { ... }
      else { ... }
      ...
      energy = calc_energy();
      i = (calc_level(energy)) ?  LOW : HIGH;
   }
}
```

Figure 3: Using Conditional Operators to adapt code

**Function Selection**

Function selection refers to the process of selecting from a set of functionally equivalent (yet operationally distinct) functions. In C, the most useful way to represent these is with function pointers. This technique is useful for applications that can select from a range of functions that perform a similar task with varying quality (e.g. Huffman encoding or quantization). As long as input and output parameter types remain the same, a function pointer can be defined and assigned at run-time to point to the correct function. Whenever that task is executed in code, the function pointer assures the currently selected function that implements the task is executed.

Figure 4 demonstrates a sample code segment written in C that makes use of function selection in the energy-aware environment described earlier. In this case, two functions, *f_low()* and *f_high()*, can be used interchangeably to perform the same action within the main loop with comparatively low or high energy consumption. The function pointer *func* is used to point to the desired function and can be updated dynamically.

Adding function selection to existing code inflicts very minor impact on code size and execution time (apart from the code size overhead of having various versions of the same function). Updating a function pointer is like updating any other pointer. Function calls are similar but will use a register to determine location of jumps rather than fixed addresses. Figure 4 demonstrates how the function call itself does not change; only the function definition changes (to a function pointer).

```
/* int func(int a, int *b, ...); */
int (*func)(int a, int *b, ...);
int f_low(int a, int *b, ...){ ...  }
int f_high(int a, int *b, ...){ ...  }

int main(void) {
   func = f_high;
   while(!end) {
      int energy;
      int i, j, k, ...;
      ...
      i = func(j, *k, ...)
      ...
      energy = calc_energy();
      func=(calc_level(energy)) ?  f_low:f_high;
   }
}
```

Figure 4: Using Function Selection to adapt code

### Loop Conditioning

Loop conditioning directly affects the conditions for exiting an executing loop. This is performed by changing the number of iterations that are executed or by changing a particular limit that the algorithm is trying to reach. This technique is useful for changing output size and for constraint based algorithms.

```
int limit = HIGH;

int main(void) {
   while(!end) {
      int i, energy;
      ...
      for (i = 0; i < limit; i++) { ... }
      ...
      energy = calc_energy();
      limit = (calc_level(energy)) ?  LOW:HIGH;
   }
}
```

Figure 5: Using Loop Conditioning to adapt code

Figure 5 demonstrates loop conditioning where the iteration limit is changing. In this case, a global variable is used to alter the number of times a loop is executed. The variable *limit* determines how many times the loop runs.

Most loops are already written in ways to support loop conditioning. This is due to the fact that most code uses loop conditioning to allow one piece of code to operate under variable input parameters. Adding adaptations exploits this design decision to allow the parameter to vary throughout execution rather than being set once and used repeatedly in the same way. Only the code to change the value of the variable needs to be added.

### Control Flow Adaptation

Control flow adaptation uses the ability of software loops to be exited early. One clear method of providing this in C is the *break* statement. Many applications that apply heuristic approaches use an algorithm that repeats or recurses to an iteration limit; as this limit increases, the quality of the result increases accordingly.

```
int main(void) {
   while(!end) {
      int energy;
      ...
      calc_energy(); /* clear energy counters */
      energy=0;
      for (;;) { /* do heuristic until break */
         ...
         energy += calc_energy();
         if (calc_level(energy)) break;
      }
   }
}
```

Figure 6: Using Control Flow Adaptation

Using this technique, an energy aware system can calculate an energy value for each iteration and automatically end the heuristic algorithm when that allocated energy is exhausted. Figure 6 demonstrates an example of this approach. When accumulated energy has increased to meet an allocated threshold (see Section 4.2), the power management code breaks the current loop to move to the next frame.

Using this method to modify existing code allows loop conditions to remain the same; the only change being the addition of the conditional break statement. When using nested loops, some more code is needed to exit all loops (provided by loop conditioning). This technique is also useful for recursive code where reaching the threshold causes the function to return.

**Non-C Adaptations**

Languages other than C allow for further adaptations not discussed here. Similarly, not all coding languages will allow the adaptations previously mentioned. For example, certain object oriented (OO) languages do not provide pointers to reference memory (e.g. Java). However, these languages allow quick morphing of object types by dynamic typecasting to provide similar functionality. In Java, an object can be created as an interface with several implementations, each with differing quality/energy characteristics. When a different implementation is required, the object is dynamically cast to an alternate type and all code that uses the object will automatically use the new version. This allows for quick changes of quality with very little run-time overhead. Techniques for other languages are possible, but are not discussed here due to length restrictions.

## 4.2 Adaptation Algorithms

We have designed two application adaptation algorithms to make use of the techniques outlined above to provide run-time optimization of the quality/energy tradeoff. Applications are modeled as having specific quality levels that are created by combining several of the techniques mentioned above. Target applications are executed at each quality level over a wide range of input to test relative energy levels for each quality level (the simulation step of Figure 2). A table storing these relative energy ratios for each quality level is utilized by the algorithms when making decisions.

Algorithm 1 shows the first adaptation algorithm, dubbed simple adaptation, which uses the quality level energy ratios to determine which quality level should be chosen at each adaptation point. The number of images that remain to be encoded is $N$, $R$ is the remaining energy to encode those images, $L$ is the current quality level with maximum $MAX$, and $levels$ is the array storing the relative energy consumption of each quality level. Using the energy for the current and/or previous frames($E$), the amount of remaining energy per image($R/N$) and the relative energy levels($levels$), the algorithm predicts the best quality level for future images($l$) to meet the quota and energy constraints. This method attempts to find the optimal quality level at which to encode frames, although this comes at the expense of occasional large jumps in the quality of the output between frames, which may be off-putting to the user.

The second algorithm, provided as Algorithm 2, smoothes the changes in the quality of the application. This algorithm uses a modified version of adaptive differential adaptation (as used in ADPCM) which initially only allows small changes in quality to be made when a change is needed. If the constraints are still not met

**Algorithm 1** Simple Application Adaptation Algorithm

$N \leftarrow$ Number of frames to encode
$R \leftarrow$ Total available energy
$L \leftarrow MAX$
**while** $N > 0$ **do**
  Encode frame at level $L$
  $E \leftarrow calc\_energy()$;  // Get energy of encoded frame
  $N \leftarrow N - 1$;
  $R \leftarrow R - E$;
  **if** $E > \frac{R}{N}$ **then**
    // Energy is too high, reduce level until it meets constraint
    **if** $L \neq 0$ **then**
      $l \leftarrow L - 1$;  // Start form next level down
      **while** $l > 0$ **and** $E \times \frac{levels[l]}{levels[L]} > \frac{R}{N}$ **do**
        $l \leftarrow l - 1$;
  **else**
    // Energy is OK, increase level if it will stay under constraint
    **while** $l < MAX$ **and** $E \times \frac{levels[l+1]}{levels[L]} < \frac{R}{N}$ **do**
      $l \leftarrow l + 1$;
  $L \leftarrow l$;  // Set level for further frames

on the next frame, increasingly larger changes are allowed. The algorithm continues in this fashion smoothing the transitions between quality levels and is less susceptible to uncharacteristic energy usage by particular frames.

# 5 Verification

We have modified several applications to make use of the adaptations methods described in this paper to provide dynamic optimization of energy consumption. These applications were tested executing on the energy estimation processor described in [2].

## 5.1 Experimental Setup

Figure 7 demonstrates the procedure used to acquire results for each application. The self-adapting application is given a total energy constraint and simulated to provide the energy use, time spent and the output data which has been varied in quality to meet the energy and quota constraints. Analysis of the performance of each application is discussed in the remainder of this section.

**Algorithm 2** Adaptive Differential Adaptation Algorithm

$change \leftarrow 0$
$N \leftarrow$ Number of frames to encode
$R \leftarrow$ Total available energy
$L \leftarrow MAX$
**while** $N > 0$ **do**
  Encode frame at level $L$
  $E \leftarrow calc\_energy()$;  // Get energy of encoded frame
  $N \leftarrow N - 1$;
  $R \leftarrow R - E$;
  **if** $E > \frac{R}{N}$ **then**
    // Energy is too high, reduce it in increasing steps
    **if** $change < 0$ **then**
      $change \leftarrow change - 1$
    **else**
      $change \leftarrow -1$
    **if** $L + change < 0$ **then**
      $change \leftarrow L$
  **else**
    // Energy is less, increase level in increasing steps
    **if** $change > 0$ **then**
      $change \leftarrow change + 1$
    **else**
      $change \leftarrow 1$
    **if** $L + change > MAX$ **then**
      $change \leftarrow MAX - L$
    // If change puts us over the constraint, reduce
    **while** $change > 0$ **and** $E \times \frac{levels[L+change]}{levels[L]} > \frac{R}{N}$ **do**
      $change \leftarrow change - 1$;
  $L \leftarrow L + change$;  // Set level for further frames

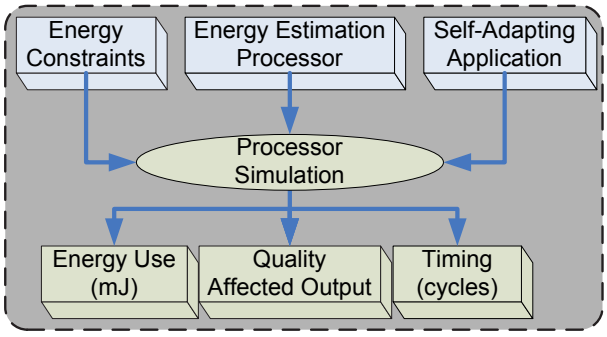
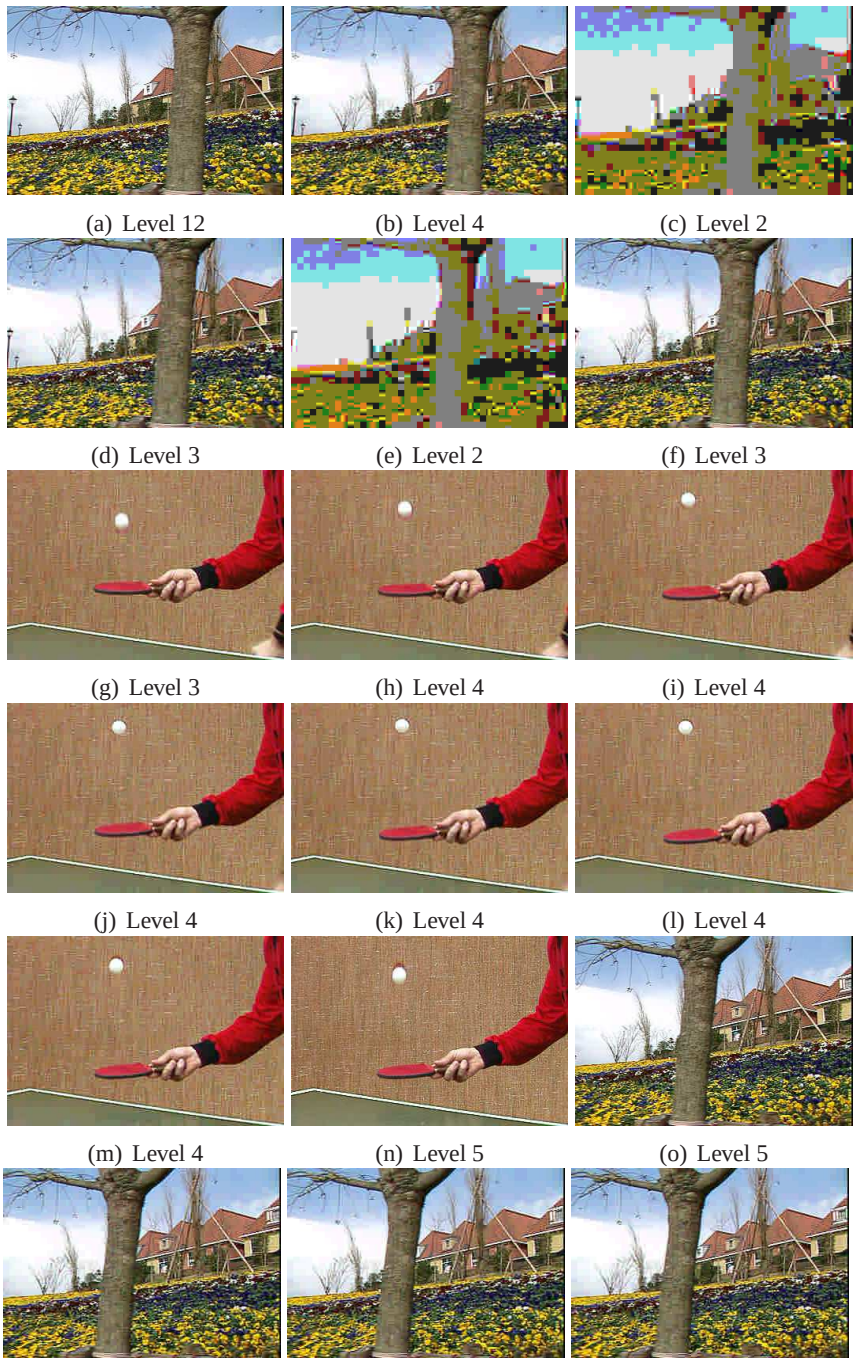



Figure 7: Verification Setup



(a) Level 12 (b) Level 4 (c) Level 2

(d) Level 3 (e) Level 2 (f) Level 3

(g) Level 3 (h) Level 4 (i) Level 4

(j) Level 4 (k) Level 4 (l) Level 4

(m) Level 4 (n) Level 5 (o) Level 5

(p) Level 4 (q) Level 4 (r) Level 4

Figure 8: Output of Sample Application with Image Quality Levels

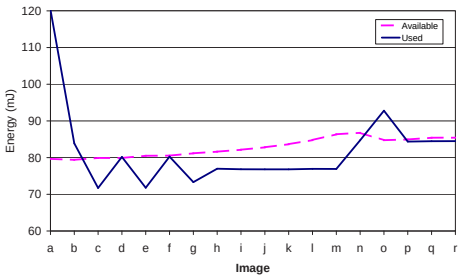



Figure 9: Energy Usage with Constraints

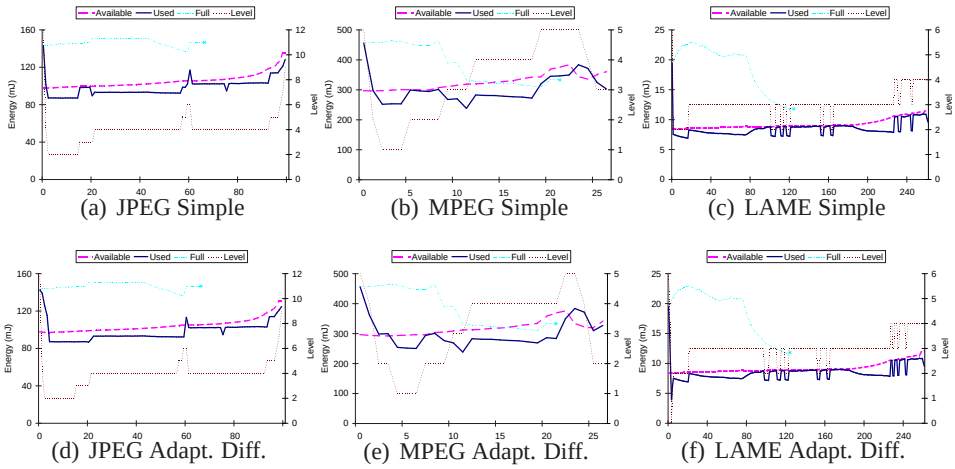
## 5.2 JPEG

The JPEG application from the Mibench testbench suite[19] was first modified to encode several images in a stream. The adaptation algorithms use only the previous frame to make predictions for further images. The use of the previous frame is justified due to the observation that images in a stream rarely change dramatically between frames.

The JPEG application was allowed to adapt itself in several ways. First, the compression amount is changed to control the lossiness of the algorithm (energy reduces as lossiness increases). Second, the function used to to perform the DCT was changed between a slow accurate version and a fast inaccurate version. Third, the lowest quality levels allow the color depth of encoded images to be modified to encode in black and white. By applying these quality adaptations in various configurations, thirteen quality levels were created.

Figure 8 shows the output of a sample execution of the Motion JPEG benchmark with the simple adaptation algorithm. Figure 9 shows both the energy use of each image in the stream (solid line) and the available energy per image for future images (dashed line). The first image is computed at maximum quality (level 12), and further images use the algorithm to change the quality level. Figure 9 demonstrates how the algorithm finds the correct power level at which to encode images to maximize quality while attempting to stay under the available energy consumption level. Also note the garden images take more energy to encode than the table tennis images at equivalent levels (see transitions from f→g and n→o). The algorithm automatically compensates for these different image types, altering the level for future images when it detects the change in energy.

Figures 10(a) and 10(d) demonstrate operation of the simple and adaptive differential adaptation algorithms respectively. Available (dashed line), Used (solid line) and Full (dash-dotted line) on the left axis refer to the available energy, used energy and full quality energy for each image. The full quality line ends when the initial energy constraint is exhausted. The dotted line is plotted on the right axis




(a) JPEG Simple (b) MPEG Simple (c) LAME Simple

(d) JPEG Adapt. Diff. (e) MPEG Adapt. Diff. (f) LAME Adapt. Diff.

Figure 10: Energy Graphs for Adaptation Algorithms

and represents the quality level the algorithm chose after each image. Input to the application was a sequence of 100 slowly changing images with occasional cuts between image types (like in the previous example). Note that both algorithms attempt to keep energy use as close as possible to the available value without exceeding it. The graphs also show that the adaptive differential algorithm produces slower transitions between quality levels (especially visible at the start). This results in a more visually appealing output as large quality jumps are more detectable in a stream of images than small ones. However, these slow transitions cause early images to consume more energy than is available, which causes later images to be encoded at reduced quality (e.g. frames 57-60 and frames approaching 100). Note that encoding frames at full quality would have caused energy to be exhausted during the $67^{th}$ frame.

## 5.3 MPEG

The MPEG application from the Mediabench testbench suite[20] has to remember more than the previous frame to decide on its adaptations. MPEG encodes each frame of a stream in one of three ways: no prediction (I frames); backward prediction (P frames); and bi-directional prediction (B frames). Prediction in this case refers to whether a frame can point to locations in previous and/or future frames in the stream. The adaptation algorithms uses each of the most recent I, P and B frames to test whether constraints are being met.

The MPEG adaptations were designed to alter the power of P and B frames;

none of the applied adaptations significantly affect the operation of I frames. When the adaptation algorithm decides a change is required, the stored P and B frame energy values are altered to reflect the new quality level (using the pre-computed energy ratios). Six quality levels were created by using combinations of two adaptations. The first adaptation type varies the search space used during forward and backward prediction. The second reduces calculation by ignoring every second macro block when performing motion prediction; the skipped blocks are set to use the same prediction as a neighboring block.

Figure 10(b) and 10(e) demonstrate the algorithms working with MPEG on an input stream of images (see the last paragraph of Section 5.2 to explain the graphs). As the three frame types each have different power characteristics, the values for used energy consumption is actually the average remaining energy per frame, which is useful for comparison purposes. Notice again that the predicted usage closely matches the total available energy usage without significantly going over. The slower transitions to improve frame by frame fidelity can also be seen for the adaptive differential version. Remembering values for each image type works well to predict the total power, but has consequences which sometimes cause errors. e.g. frames 23-24 of both algorithms where an unexpected change in energy consumption of an I frame caused power to be too high for a few frames. Note that both algorithms were able to automatically correct to bring energy consumption back under the constraint before the end of the input set. Encoding at full power causes the algorithm to deplete its energy supply during the $22^{nd}$ frame of 27.

### 5.4 LAME

LAME is an MP3 encoding application from Mibench[19]. As audio data is much more subject to sudden change in input than video, we used a running average value derived from the leaky capacitor model to compare with the prediction constraint. This algorithm works as follows:

$$ave(n) = \{\, E\,(1)(1-q).ave(n-1) + q.E(n) \qquad n\ = 1n > 1$$

where $E(n)$ and $ave(n)$ are the energy and average energy for the $n^{th}$ sample respectively and q is a constant between 0 and 1 that approximates the number of values being averaged over (no. of values $\approx$ 1/q). Whenever an adaptation is applied the average is also updated via the quality ratios to reflect the new quality level.

The adaptations designed for LAME varied some of the quality settings already existent in the application. These settings affect operation such as Huffman coding, quantization, noise shaping and psychoacoustics. In addition, certain quality levels

Table 1: Comparison of Adaptation Algorithms

| Benchmark | | Code Size (bytes) | Increase (bytes) | % | Total cycles ($*10^6$) | Adapt. cycles ($*10^3$) | % |
|---|---|---|---|---|---|---|---|
| JPEG | O. | 223,184 | | | | | |
| | S. | 224,352 | 1168 | 0.52% | 15,858 | 362.8 | 0.0023% |
| | A. | 224,384 | 1200 | 0.54% | 15,832 | 364.4 | 0.0023% |
| MPEG | O. | 215,120 | | | | | |
| | S. | 217,248 | 2128 | 0.99% | 12,924 | 80.7 | 0.0006% |
| | A. | 217,136 | 2016 | 0.94% | 13,016 | 81.5 | 0.0006% |
| LAME | O. | 377,424 | | | | | |
| | S. | 380,704 | 3280 | 0.87% | 3,034 | 647.6 | 0.0213% |
| | A. | 380,144 | 2720 | 0.72% | 3,034 | 674.9 | 0.0223% |

also force the use of side stereo encoding (which provides better bandwidth at the cost of extra energy).

Figures 10(c) and 10(f) demonstrate the simple and adaptive differential energy optimization algorithms working with the modified LAME application (See Section 5.2 for explanation). These two graphs appear similar as the averaging causes values to change smoothly, causing the algorithms to react in similar ways. Once more, both algorithms accurately modify energy consumption to meet the available energy constraint. For LAME, full quality results in energy running out after 125 frames, less than half the number of frames that were requested.

### 5.5 Impact

Table 1 shows the impact on code size and execution time for each of the applications presented above. Code size compares the original code size before addition of the adaptations, power estimation loop and adaptation algorithms. Execution time measures the amount of time spent performing non-application operations (adaptation algorithms and power estimation) for the above examples. Column 1 lists the benchmark and adaptation method used. In this column, 'O.' stands for the original algorithm, 'S.' stands for the simple adaptation method and 'A.' refers to the adaptive differential adaptation method. Column 2 lists the number of bytes of code size used by each of the benchmarks. Column 3 lists the increase in code size seen in the benchmarks modified for adaptation. Column 4 represents this increase as a percentage. Column 5 shows the total number of executed cycles, while Column 6 shows the number of these cycles that were spent executing power estimation and adaptation code. Column 7 represents the percentage of cycles spent executing this

non-application type code. From the table, it can be seen that the added code to perform energy estimation and power optimization operates for a minor fraction of the total run-time (worst case of 0.0223%). Code size introduced by the application adaptation and power estimation code is also small with MPEG showing the largest proportion which is still less than 1%. This demonstrates that using these procedures will usually not require the instruction memory to be scaled up to a larger size.

# 6 Conclusion

This paper has provided a methodology for energy driven self application self adaptation which has been enabled by recent breakthrough technologies in the estimation of run-time power consumption. The methodology provides several adaptation techniques and adaptation algorithms that have proved extremely effective for several multimedia applications without greatly impacting upon the operation of the system ($<$0.02% cycles). The presented techniques are widely applicable to a range of common programming methods and have much less impact and greater cognisance than previous OS-based approaches. This methodology benefits many environments in the embedded domain including military and consumer multimedia and network applications.

# References


[1] M Pedram and Q Wu. Design Considerations for Battery-powered Electronics. In *Proceedings of DAC*, 1999.

[2] J Peddersen and S Parameswaran. CLIPPER: Counter-based Low Impact Processor Power Estimation at Run-time. Technical Report UNSW-CSE-TR-0618, The University of New South Wales, 2006.

[3] J Flinn and M Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, 1999.

[4] J Rabaey and M Pedram, editors. *Low Power Design Methodologies*. Kluwer Academic Publishers, 1996.

[5] A Andrei, M Schmitz, P Eles, Z Peng, and B Al-Hashimi. Quasi-Static Voltage Scaling for Energy Minimization with Time Constraints. In *Proceedings of DATE*, 2005.

[6] Y Le, J Luo, and N Jha. Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Heterogeneous Distributed Real-Time Embedded Systems. In *Proceedings of the International Conference on Computer-Aided Design*, 2003.

[7] Steven Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Lower Power Microprocessors Under Dynamic Workloads. In *Proceedings of the International Conference on Computer-Aided Design*, 2002.

[8] J Tschanz, S Narendra, R Nair, and V De. Effectiveness of Adaptive Supply Voltage and Body Bias for Reducing Impact of Parameter Variations in Low Power and High Performance Microprocessors. In *IEEE Journal of Solid-State Circuits*, volume 38. 2003.

[9] H Aydin, R Melhem, D Mosse, and P Mejia-Alvarez. Power-Aware Scheduling for Periodic Real-Time Tasks. In *IEEE Transcations on Computers*, volume 53, pages 584–600. May 2004.

[10] J Wang, B Ravindran, and T Martin. A Power-Aware, Best-Effort Real-Time Task Scheduling Algorithm. In *IEEE Workshop on Software Technologies for Future Embedded Systems*, 2003.

[11] A Raghunathan, S Dey, and N Jha. High-Level Macro-Modeling and Estimation Techniques for Switching Activity and Power Consumption. In *IEEE Transactions on Very Large Scale Integration Systems*, volume 11. Aug 2003.

[12] J Flinn and M Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *Proceedings of the ACM SOSP*, 1999.

[13] R McGowen. Adaptive Designs for Power and Thermal Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, 2005.

[14] V Raghunathan, C Pereira, M Srivastava, and R Gupta. Energy-Aware Wireless Systems with Adaptive Power-Fidelity Tradeoffs. In *IEEE Transactions on Very Large Scale Integration Systems*, volume 13, pages 211–225. Feb 2005.

[15] BD Noble et al. Agile application-aware adaptation for mobility. In *Proceedings of the ACM SOSP*, 1997.

[16] BD Noble and M Satyanarayanan. Experience with Adaptive Mobile Applications in Odyssey. In *Mobile Netwroks and Applications*, volume 4. 1999.

[17] F Bellosa. The Case for Event Driven Energy Accounting. Technical Report TR-I4-01-07, University of Erlangen, 2001.

[18] D Berger and T Austin. *SimpleScalar Tool Set, Version 2.0*. http://www.simplescalar.com/docs/users_guide_v2.pdf, 1997.

[19] M Guthaus, J Ringenburg, D Ernst, T Austin, T Mudge, and R Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE Annual Workshop on Workload Characterization*, 2001.

[20] C Lee, M Potkonjak, and W Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 1997.