Dynamic Frequency Scaling With Buffer Insertion for Mixed Workloads

Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—This paper presents a method to reduce the energy of interactive systems for mixed workloads: multimedia applications that require constant output rates and sporadic jobs that need prompt responses. The authors' method divides multimedia programs into stages and inserts data buffers between them. Data buffering has three purposes: 1) to support constant output rates; 2) to allow frequency scaling for energy reduction; and 3) to shorten the response times of sporadic jobs. The authors construct frequency-assignment graphs. Each vertex represents the current state of the buffers and the frequencies of the processor. The authors develop an efficient graph-walk algorithm that assigns frequencies to reduce energy. The same method can be applied to perform voltage scaling and the combination of frequency and voltage scaling. The authors' experimental results on a StrongARM-based computer show that four discrete frequencies are sufficient to achieve nearly maximum energy saving. The method reduces the power consumption of an MPEG program by 46%. The authors also demonstrate a case that shortens the response time of a sporadic job by 55%.**

***Index Terms*—Frequency scaling, multimedia, power reduction.**

I. Introduction

PORTABLE computers, like iPAQ, are increasingly popular. Such systems can execute multimedia programs that require consistent audio and video output rates to maintain satisfactory quality of service. Meanwhile, these systems continue accepting user inputs that need prompt responses. In sum, they execute mixed workloads. Most of these systems operate on batteries and require low power consumption to keep long the operational time between the recharging of batteries. This paper presents a method to reduce power consumption for mixed workloads.

Most processors use CMOS-based circuits; they consume power mainly during switching from logic true to false, or vice versa. The switching power is proportional to the clock frequency and the square of the supply voltage. Therefore, lowering clock frequencies and/or supply voltages reduces power [33]. This is called *dynamic frequency* (*voltage*) *scaling* [7], [13]. Scaling may have a negative impact on timing-sensitive programs, for example, by failing to meet the output rate for a multimedia program. How to scale frequencies and voltages while meeting timing constraints has been an active research topic in recent years [1], [19], [25], [27], [29], [28], [30], [31], [35], [36].

This paper proposes a software-based technique to reduce power by dynamic frequency scaling on processors that have only finite frequencies. Our method inserts data buffers in a multimedia program. Data are processed and stored in the buffers when the processor runs at a higher frequency. Later, the processor runs at a lower frequency to reduce power and data are taken from the buffers to maintain the same output rate. Before the buffers become empty, the processor begins to run at a higher frequency again. Inserting data buffers provides opportunities to reduce power consumption. Buffering can also shorten the response time of a sporadic job. If there are enough data in the buffers, the processor can handle a sporadic job without affecting the output rate of the multimedia program. Our method computes the optimal assignments of processor frequencies by traversing a finite graph. In this graph, each vertex represents the current state of the buffers, the processor frequencies, and how the buffers are filled (or drained). We present an efficient method to compute optimal solutions by graph walking. The same method can be applied to voltage scaling or the combination of frequency and voltage scaling.

This method was implemented on a StrongARM-based hand-held computer. Our experimental results show that inserting buffers can achieve nearly optimal power saving with only a few discrete frequencies. Our method reduces the power consumption of an MPEG program by 46%, after the nonscaleable base power is excluded. We also present a case that reduces the response time by 55% with negligible increase in energy consumption.

II. Background

A. Jobs and Constraints

A program can be decomposed into smaller units, called "jobs" [8]. For example, an MPEG player program can be divided into two jobs: processing and displaying images. The processing job may be further divided into smaller units, including reading the file and decoding the data. If a job must execute before another job, there is a *precedence constraint* between these two jobs. Precedence constraints are determined by the structure of a program. For instance, an MPEG player has to decode an image before displaying it. We use "$\rightarrow$" to represent precedence constraints. If job j_a has to execute

Manuscript received November 15, 2001; revised April 3, 2002. This work was supported in part by the MARCO/Defense Advanced Research Projects Agency Gigascale Silicon Research Center and by the National Science Foundation under Contract CCR-9901190. This paper was recommended by Associate Editor R. Gupta.

Y.-H. Lu was with Stanford University, Stanford, CA 94305 USA. He is now with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907-1285 USA.

L. Benini is with the Department of Electronics and Computer Science, University of Bologna, Bologna 40136 Italy.

G. De Micheli is with the Department of Electrical Engineering and the Department of Computer Science, Stanford University, Stanford, CA 94305-9030 USA.

Digital Object Identifier 10.1109/TCAD.2002.804087

0278-0070/02$17.00 © 2002 IEEE

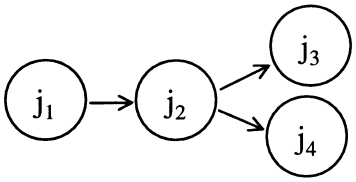


Fig. 1. Precedence constraints form a directed acyclic graph.

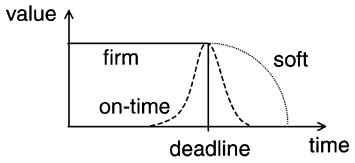


Fig. 2. Three types of deadlines.

before job j_b, their relationship is expressed as $j_a \rightarrow j_b$. For example, suppose j_{decode} and j_{display} are the jobs to decode and display an MPEG image. The precedence relationship is $j_{\text{decode}} \rightarrow j_{\text{display}}$. Precedence constraints are often represented as a *directed acyclic graph* (DAG). Fig. 1 is an example of precedence constraints: j_1 has to execute before j_2 and j_2 has to execute before j_3 and j_4. Precedence constraints also occur because of the sequential relationship of data. The first frame of an MPEG video must be displayed before the second frame.

A *timing constraint* requires that a job finish within a given duration. Timing constraints can be classified according to three categories: firm, soft, and on-time [8]. Fig. 2 illustrates the differences between these constraints. Suppose there is a "value" if a job finishes before the deadline. For a firm deadline, the value drops sharply if the job finishes after the deadline. Examples of firm deadlines are flight control systems: finishing a job after the deadline can lead to severe damage or even loss of lives. For a soft deadline, the value decreases more smoothly after the deadline. If a job has an on-time constraint, it should finish near the deadline, neither too early nor too late. Playing an MPEG movie requires a consistent frame rate (number of frames per second). In other words, j_{display} has to execute repetitively at a constant rate. The time between $j_{i,\,\text{display}}$ and $j_{i+1,\,\text{display}}$ should be a constant, here $j_{i,\,\text{display}}$ is the job to display the ith frame. For playing an MPEG movie at 30 frames/s, the player has to display one frame every 33 ms, neither much shorter nor much longer. This is a periodic on-time constraint.

User inputs create "sporadic" jobs. Usually, sporadic jobs are processed concurrently with other already running programs. For example, a user may move the mouse cursor while watching an MPEG movie. The movement of the cursor has to be processed and displayed on the screen. Sporadic jobs need to be processed promptly for interactivity. We can specify two types of timing constraints for sporadic jobs: 1) the processing time of each sporadic job is shorter than a given value and 2) the average processing time is shorter than a given value.

B. Dynamic Frequency and Voltage Scaling

The dynamic power of a CMOS gate can be approximated by $p = c \cdot v_{dd}^2 \cdot sw \cdot f$, here c is the load capacitance, v_{dd} is the supply voltage, sw is the switching activity, and f is the clock frequency [38]. Power can be reduced by lowering f and/or v_{dd}. This is called dynamic frequency (voltage) scaling. The total energy consumed during the time interval $[0, T]$ is the integration of power in this duration: $e = \int_0^T p\,dt$. If we replace the load capacitance and the switching activity by their averages, the energy is given by the proportionality relation: $e \propto \int_0^T v_{dd}^2 \cdot f\,dt$. Some commercial processors, such as StrongARM, have instructions to adjust the clock frequency f. StrongARM processors have special registers to specify the current clock frequencies [21]. Modifying the values in these registers changes the frequencies. There are 11 frequency settings available, between 59 and 206 MHz. StrongARM processors do not have software-controlled voltage scaling; voltage scaling can be achieved by adding external voltage regulators [32]. Intel's Xscale processors support both frequency and voltage scaling [40].

Suppose frequencies and voltages can change only at time t, $2t$, …, nt and $nt = T$. The frequency and voltage during $((i-1)t, it]$ is f_i and $v_{dd,\,i}, i \in \{1, 2, \ldots, n\}$. Then, the energy is computed by the following formula:

$$e \propto \int_0^T v_{dd}^2 \cdot f\,dt = \sum_{i=1}^{n} (v_{dd,\,i})^2 f_i \cdot t \propto \sum_{i=1}^{n} (v_{dd,\,i})^2 f_i. \quad (1)$$

If the voltage is kept constant, the energy is determined by the frequencies. Thus, we obtain the following relationship:

$$e \propto \sum_{i=1}^{n} f_i. \quad (2)$$

Many existing scaling schemes assume that voltage and frequency can scale continuously [7], [13], [37], [25], [31]. This assumption is false for commercial processors, such as StrongARM. Some schemes consider discrete frequencies and formulate the problem as integer linear programming [22], [26], [28]; unfortunately, they are computationally expensive. This paper will present an efficient scaling method using graph traversal techniques.

C. Buffer Insertion

As explained earlier, an MPEG player can be divided into *stages*, such as decoding and displaying. These stages form a *pipeline*. Let $j_{i,\,p}$ and $j_{i,\,d}$ be the jobs to process (i.e., decode) and to display the ith frame. A frame has to be decoded before being displayed, therefore $j_{i,\,p} \rightarrow j_{i,\,d}$. Without additional storage between the two stages, no frame can be processed before the previous frame is displayed. This requires $j_{i-1,\,d} \rightarrow j_{i,\,p}$. Fig. 3 is the precedence relationship for such a pipeline. If there are data buffers between the two stages, a frame can be processed even if the previous frame has not been displayed. Hence, $j_{i-1,\,d}$ does not have to precede $j_{i,\,p}$. The precedence relationship is changed, as shown in Fig. 4. This figure assumes that the data have to be processed sequentially, consequently $j_{i,\,p} \rightarrow j_{i+1,\,p}$. Also, frames should be displayed sequentially: $j_{i,\,d} \rightarrow j_{i+1,\,d}$. After buffers are inserted between the stages, there are multiple options to arrange the execution order of these jobs. For example, $j_{2,\,p}$ can execute before $j_{1,\,d}$.

1) Energy Reduction With Buffers: An MPEG player has to maintain a constant output rate: it has to display a frame every t

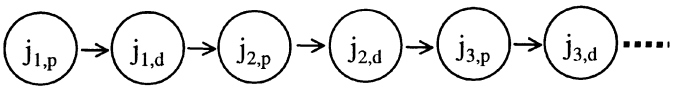


Fig. 3. Processing and displaying form a pipeline.

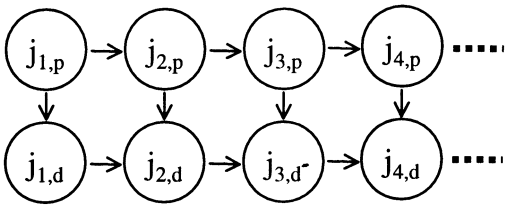


Fig. 4. Inserting buffers changes the precedence relationship.

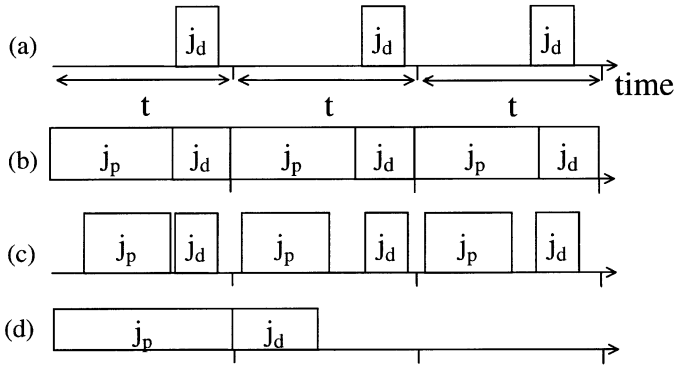


Fig. 5. (a) Constant output rate for display. (b) Scaling frequency to avoid slack time. (c) Discrete frequencies cause idleness and waste energy. (d) Scale to a lower frequency and miss the output rate.

units of time; t is called a *period*. For a movie with 30 frames/s, t is 33 ms. Fig. 5(a) shows this requirement: j_d executes once every period. If a processor's frequency can be set to any value, the processor consumes the minimum energy when it takes exactly t to process and display one frame [22]. As Fig. 5(b) showns, there is no slack time.

However, if a processor has only finite frequencies and this optimal frequency is unavailable, the processor has to run at a higher frequency. Since this frequency is higher than optimal, the processor consumes more power. The processor is idle after processing and displaying one frame, as shown in Fig. 5(c). The processor cannot enter a lower frequency because if it does, it will fail to provide the required output rate. Fig. 5(d) illustrates this situation. While it is possible to set a processor to the sleeping state to save power during the idleness, the wakeup delay can be prohibitively long. For example, it takes 160 ms to wake up a StrongARM processor from the sleeping state [4]. In contrast, it takes only a few μs to change the processor frequency. Consequently, this paper focuses on frequency scaling only and does not consider using the sleeping state.

One solution to save power is scaling down the processor frequency whenever it is idle. However, changing frequencies takes time; hence, it is preferable to avoid changing the frequencies too often. Another solution is to insert buffers between jobs so that the processor can process more frames at a higher frequency. When enough frames have been processed and stored in the buffers, the processor retrieves processed frames from the buffer to maintain the output rate. Since the processor does not have to process images, it can enter a lower frequency and still meet the output rate requirement. Fig. 6 depicts this approach. In this figure, the height means the processor frequency. Four frames are processed in the first two and half periods. Then, the processor is scaled down to a lower frequency. Before the buffers become empty, the processor enters the higher frequency and refills the buffers. Buffers are used to reduce the power in pipelines [18], [6], [9] or to smoothen run-time variations [20]. Previous studies have not considered the advantages of buffers on processors with finite frequencies.

2) Reducing Response Time: In addition to being able to reduce power, buffer insertion can also improve the performance of sporadic jobs without disrupting other jobs. Imagine that a user moves the mouse cursor and clicks one button at the end of the 12th period as shown in Fig. 7. This command can be divided into two jobs: $j_{r,1}$ and $j_{r,2}$. The first job draws the movement of the cursor; the second job processes the click command. Fig. 7 shows two scenarios: with and without a buffer. In (a), no additional frame is buffered: j_p has to execute once every period. Only $j_{r,1}$ can execute during the 13th period; $j_{r,2}$ has to wait until the 14th period. In contrast, (b) shows four additional frames being buffered ($j_{16,p}$ executes at the 12th period); both $j_{r,1}$ and $j_{r,2}$ can execute during the 13th period. As a result, the user can see the response of this command in the 13th period in (b). Buffering reduces the response time of a sporadic command.

Even though buffering images requires additional memory, a typical computer has enough memory to buffer multiple frames. For example, palm-size computers often have more than 8-MB memory. A frame of 240 $\times$ 160 pixels with 256 colors per pixel requires 240 $\times$ 160 bytes, or 38 KB. Four hundred kilobytes are enough to buffer ten frames and are only 5% of the available memory.

III. Related Work

Scaling techniques can be split into two categories according to whether or not they consider timing constraints. The first category does not guarantee that timing constraints are met. In [37], the authors propose several methods that periodically estimate process utilization and adjust the power states. Simulations of various techniques are presented in [14] and [31]. In [36], the authors model the arrival of jobs as random processes. While this approach can meet timing constraints statistically, it does not guarantee to *always* meet them. The second category considers timing constraints. In [19], the authors use off-line analysis to determine whether it is possible to meet hard deadlines and to assign the power states. Linear programming methods are proposed in [22] and [28] to find optimal voltages/frequencies for processors with discrete power states while meeting deadlines. Some techniques have been implemented on real systems. In [1], [17], and [32], the authors use StrongARM-based systems to demonstrate the effectiveness of scaling and point out some limitations in implementation. Our work differs from existing approaches in the following ways.

- Frequencies are assigned to processors that have finite frequencies.

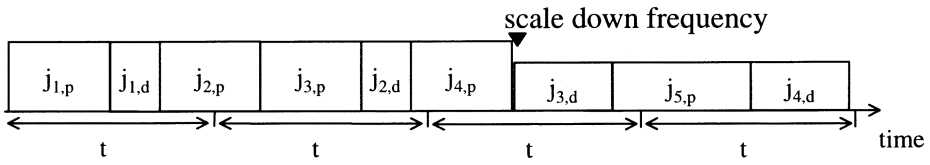


Fig. 6. Process more than one frame at the higher frequency, then scale to the lower frequency.

- Data buffers are inserted into a program that needs a constant output rate.
- An efficient graph-based method is presented to assign frequencies for reducing energy.
- Workloads with very long time horizons can be handled by this method.
- The response times of sporadic jobs are shortened without affecting the output rate.
- The minimum buffer sizes can be calculated to meet the timing constraints of sporadic jobs.

IV. Assumptions

We make the following assumptions to simplify the formulation of the problem. These assumptions may be removed as extensions of the work presented in this paper.

- The processor has only discrete and finite frequencies. The processor changes frequencies (or voltage) only at the beginning of a period of length t.
- Data processing is sequential on a single processor: there is no forward data dependence.[1]
- The total energy is determined by frequencies only: we will use formula (2) to calculate energy. We consider the average power for a given duration. Since the integration of power over time is the energy, minimizing energy is equivalent to minimizing power. We use the terms energy and power interchangeably unless it is necessary to distinguish them.
- The jobs in the multimedia program are atomic and their execution cannot cross period boundaries. Jobs are schedulable at the highest frequency. Buffers are not shared among jobs.
- The computational work of a job is measured by the number of operations. One operation takes one time unit at unit frequency. Hence, the execution time of the same job increases linearly to the reciprocal of the frequency. The number of operations for a specific job is constant.
- It takes no time to start executing a job and there is no context-switching overhead. This assumption is valid for a multimedia program because there is no real context switching. All jobs belong to the same process.

V. Analytical Model by Mathematical Programming

With the assumptions stated above, the power reduction problem can be modeled as a mathematical programming problem. For a complete comparison, we show the details of such modeling before transforming it into a graph-walking problem later. This section derives an analytical model for energy minimization under performance and resource constraints. We start with integer linear programming. The formulation becomes more general and more complex as we consider additional factors.

[1]The B (bidirectional) frames in MPEG are not considered. One example of video without forward dependence is motion JPEG. B frames cause forward data dependence and they cannot be decoded based on the information available from previous frames. However, MPEG is divided into group of frames so only finite "look-ahead" is needed. Our method can be applied to MPEG by considering a group of frames as the basic unit.

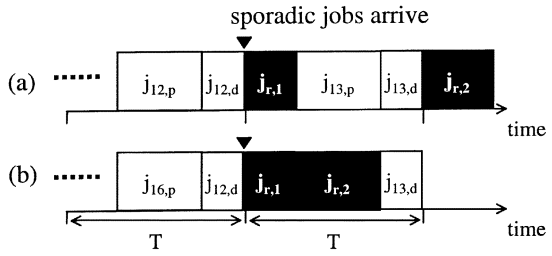


Fig. 7. (a) No buffer. (b) Buffer additional frames to reduce the response time of a sporadic job.

TABLE I
Symbols and Meanings

n	number of frames
m	number of jobs
s	number of frequencies
w_l	number of operations of job j_l
$a_{i,l}$	number of executions of j_l in the i^{th} period
ϕ_q	one of the available frequencies, $\phi_1 > \phi_2, ..., > \phi_s$
Δ	time to change frequencies and/or voltages
b_l	size of the buffer between j_l and j_{l+1}
t	length of a period
$\mathcal{N}$	$\{1, 2, ..., n\}$
$\mathcal{M}$	$\{1, 2, ..., m\}$
i	index of period, $i \in \mathcal{N}$
l	index of job, $l \in \mathcal{M}$
f_i	frequency in the i^{th} period
δ_i	frequency or voltage changed in the i^{th} period

A. Two Frequencies and Two Jobs

This section assumes: 1) there are two repetitive jobs; 2) the processor allows two integer frequencies: ϕ_1 and ϕ_2; and 3) changing frequencies takes no time. We will remove these assumptions later. An MPEG movie has n frames to display in n periods. The length of a period is t. For each frame, there are two jobs: processing (j_p) and displaying (j_d). A frame is processed before being displayed ($j_p \rightarrow j_d$). Table I summarizes the symbols used in this section.

The ith period is the time interval of $((i-1)t, it]$. Let $\mathcal{N} = \{1, 2, 3, \ldots, n\}$. We use f_i to indicate the frequency during the ith period, $f_i \in \{\phi_1, \phi_2\}$ and $i \in \mathcal{N}$. Let a_i be the number of frames processed during the ith period; it is a nonnegative integer. If $a_i = 1$, one frame is processed during this period. Suppose w_p and w_d are the number of operations for processing and displaying one frame. It takes w_p/ϕ_1 to process one frame

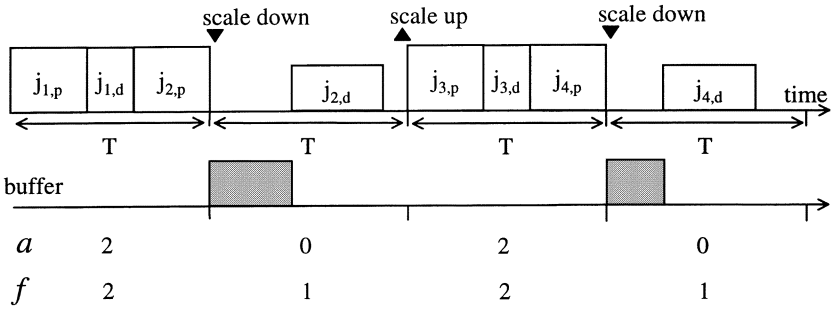


Fig. 8. The processor changes frequencies every period.

at frequency ϕ_1. During the ith period, the total number of operations is $a_i \cdot w_p$ for processing and w_d for displaying. They have to finish within a period. Therefore, $(a_i \cdot w_p + w_d)/f_i \leq t$, $\forall i \in \mathcal{N}$. This can be rewritten as

$$a_i \cdot w_p + w_d \leq t \cdot f_i \qquad \forall i \in \mathcal{N}. \tag{3}$$

The number of frames processed up to kt ($k \in \mathcal{N}$) is the sum of frames processed in each period: $\sum_{i=1}^{k} a_i$. At least k frames have to be processed before kt because k frames have been displayed at kt. This can be expressed by the constraint: $\sum_{i=1}^{k} a_i \geq k, \forall k \in \mathcal{N}$. Since $\sum_{i=1}^{k} a_i$ frames have been processed but only k frames are displayed. The additional frames are stored in a buffer. Suppose one frame takes one unit of space and the buffer size is b. The following constraint restricts the number of frames processed so that they do not overflow the buffer at kt: $(\sum_{i=1}^{k} a_i) - k \leq b, \forall k \in \mathcal{N}$. These two constraints are expressed as follows:

$$0 \leq \left(\sum_{i=1}^{k} a_i\right) - k \leq b \qquad \forall k \in \mathcal{N}. \tag{4}$$

As explained in Section II-B, when the voltage is kept constant, the total energy for n frames is proportional to the sum of frequencies of during all periods. The energy is proportional to

$$\sum_{i=1}^{n} f_i. \tag{5}$$

This is the cost function. The problem of energy minimization is to find a frequency assignment (the value of f_i for $i \in \mathcal{N}$) and an execution order (the value of a_i) to minimize the total energy, expressed in formula (5), while meeting all constraints. This is an integer linear programming problem (ILP). The parameters depend on the processor (ϕ_1 and ϕ_2), the system (b), and the workload (w_p, w_d, n, and t). While this formulation may appear excessive for minimizing the energy for a processor running two jobs at one of two possible frequencies, we use it as the foundation for handling more complex and realistic situations.

Example 1: Suppose $t = 6$, $\phi_1 = 2$, $\phi_2 = 1$, $n = 4$, $b = 1$, $w_p = 4$, and $w_d = 2$. There are four inequalities from the constraints specified in (3). There are four other inequalities from the precedence constraints and the resource constraints specified in (4)

$$\begin{aligned}
4a_1 + 2 &\leq 6f_1 \\
4a_2 + 2 &\leq 6f_2 \\
4a_3 + 2 &\leq 6f_3 \\
4a_4 + 2 &\leq 6f_4 \\
0 &\leq a_1 - 1 \leq 1 \\
0 &\leq a_1 + a_2 - 2 \leq 1 \\
0 &\leq a_1 + a_2 + a_3 - 3 \leq 1 \\
0 &\leq a_1 + a_2 + a_3 + a_4 - 4 \leq 1 \\
f_1, f_2, f_3, f_4 &\in \{1, 2\}.
\end{aligned} \tag{6}$$

By (5), the cost function is

$$\min(f_1 + f_2 + f_3 + f_4). \tag{7}$$

The minimum energy can be obtained by setting $f_1 = f_2 = f_3 = f_4 = 1$. The processor always stays in the lower frequency, ϕ_2. One frame is processed each period: $a_1 = a_2 = a_3 = a_4 = 1$. ♢

Example 2: Consider $w_p = 5$. In this case, $f_1 = f_2 = f_3 = f_4 = 1$ is no longer a valid solution because the constraints in (6) are violated. The minimum energy can be obtained by setting $f_1 = f_3 = 2$ and $f_2 = f_4 = 1$. Two frames are processed in the first and third periods and no frame is processed in the second and fourth periods: $a_1 = a_3 = 2$, $a_2 = a_4 = 0$. The processor changes frequencies every period as shown in Fig. 8. Note that it is prohibited to process four frames in the first two periods by setting $f_1 = f_2 = 2$, $f_3 = f_4 = 1$, $a_1 = a_2 = 2$, and $a_3 = a_4 = 0$, because this violates the buffer size constraints in (6). ♢

B. Multiple Jobs

We can generalize the formulation to handle multiple jobs. Suppose there are m jobs: $\mathcal{J} = (j_1, j_2, \ldots, j_m)$ and j_m has to execute once every period. Let $\mathcal{M}$ be $\{1, 2, \ldots, m\}$. We use $j_{i,l}$ for the ith execution of j_l, $l \in \mathcal{M}$. For any $l \in \{1, 2, 3, \ldots, m-1\}$, job $j_{i,l}$ has to execute before $j_{i,l+1}$, where $i \in \mathcal{N}$. Fig. 9 shows the precedence relationship between these jobs. Let w_l be the number of operations performed by j_l. Let $a_{i,l}$ be the number of executions of j_l during the ith period;

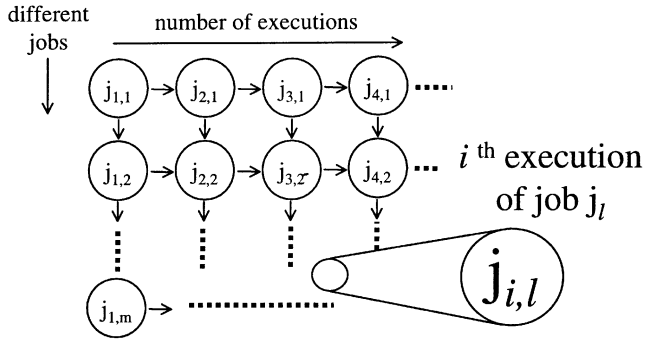


Fig. 9. Precedence of multiple jobs.

$a_{i,l} = 1$ means that j_l executes once in this period. Since j_m executes exactly once each period, $a_{i,m} = 1$ for any value of $i \in \mathcal{N}$. The total number of operations performed in the ith period is $\sum_{l=1}^{m} a_{i,l} \cdot w_l$. All operations have to finish within the period. Therefore

$$\sum_{l=1}^{m} a_{i,l} \cdot w_l \le t \cdot f_i \qquad \forall i \in \mathcal{N}. \tag{8}$$

At kt, j_l has processed $\sum_{i=1}^{k} a_{i,l}$ frames. The following constraints allow one frame to be displayed each period:

$$\sum_{i=1}^{k} a_{i,1} \ge \sum_{i=1}^{k} a_{i,2} \ge \cdots \ge \sum_{i=1}^{k} a_{i,m} = k \qquad \forall k \in \mathcal{N}. \tag{9}$$

At time kt, job j_l has executed $\sum_{i=1}^{k} a_{i,l}$ times and job j_{l+1} has executed $\sum_{i=1}^{k} a_{i,l+1}$ times. The additional frames are stored in a buffer. Let b_l be the size of the buffer between j_l and j_{l+1}. The following constraint avoids buffer overflow:

$$\sum_{i=1}^{k} a_{i,l} - \sum_{i=1}^{k} a_{i,l+1} \le b_l \qquad \forall k \in \mathcal{N} \quad \text{and} \quad \forall l \in \mathcal{M} - \{m\}. \tag{10}$$

The goal is finding f_i and $a_{i,l}$ to minimize energy (5) under the constraints expressed by (8)–(10).

C. Multiple Frequencies

Consider a processor with s integer frequencies: $\{\phi_1, \phi_2, \ldots, \phi_s\}$. Suppose f_i is the frequency during the ith period, then $f_i \in \{\phi_1, \phi_2, \ldots, \phi_s\}$. The cost function is the same, as expressed in (5). The execution time constraint in (8) is also the same, except that f_i can be one of the s available frequencies.

Example 3: Consider a processor with three frequencies: {4, 2, 1} for three jobs: j_1, j_2, and j_3. The numbers of operations for the jobs are $w_1 = 12$, $w_2 = 8$, and $w_3 = 4$. The length of a period is 11. There are two buffers, b_1 and b_2; each can accommodate one frame. There are three frames, $n = 3$. Table II shows the execution time of each job at different frequencies. Since $t = 11$ and it takes 12 time units to execute three jobs at frequency 2, the processor must run at the highest frequency in the first period. Fig. 10 shows the solution for the lowest energy. In this figure, a is the sequence of $a_{i,l}$ for the ith period. In the first period, $a = \langle 2, 2, 1\rangle$. This means $a_{1,1} = a_{1,2} = 2$ and $a_{1,3} = 1$. The frequencies in the first, second, and third periods are 4, 2, and 2, respectively. ♢

TABLE II
EXECUTION TIME AT DIFFERENT FREQUENCIES FOR EXAMPLE 3

frequency	execution time			total
	j_1	j_2	j_3	
4	3	2	1	6
2	6	4	2	12
1	12	8	4	24

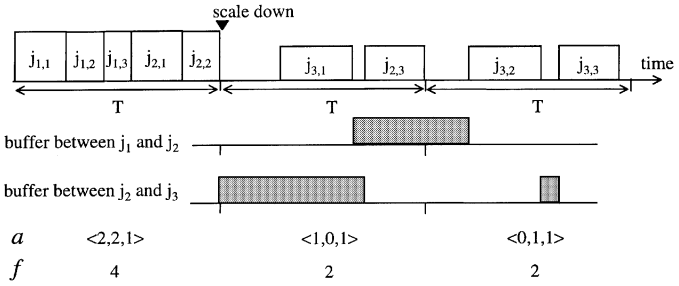


Fig. 10. Lowest energy solution for Example 3.

D. Multiple Voltages

This section generalizes the formulation to consider both frequency and voltage scaling. Suppose the processor has y discrete voltage settings: $\{v_1, v_2, \ldots, v_y\}$. Let $v_{dd,i}$ be the voltage at the ith period. The power consumption during the ith period is proportional to $(v_{dd,i})^2 f_i$. We need to choose the value of f_i and $v_{dd,i}$ to minimize the overall energy, which is proportional to

$$\sum_{i=1}^{n} (v_{dd,i})^2 \cdot f_i. \tag{11}$$

The values of f_i and $v_{dd,i}$ should be determined so that all the constraints expressed in (8)–(10) are satisfied. Note that the maximum value of f_i can be determined from $v_{dd,i}$ analytically [33]. When voltage scaling is considered, the cost function (11) contains the products of $v_{dd,i}$ and f_i. This is no longer a linear programming problem. We can further generalize the problem for noninteger values of $v_{dd,i}$ and f_i. Consequently, energy minimization with both voltage and frequency scaling can be formulated as a (cubic) mathematical programming problem.

E. Scaling Overhead

Suppose changing frequencies and/or voltages takes Δ time regardless of the original and new frequencies and voltages. Furthermore, the processor cannot execute any job during frequency and/or voltage changes. In [17], the authors report 0.2 ms for changing frequencies on a StrongARM processor. This is less than 1% of a period (33 ms). While it is possible to change frequencies multiple times within each period, this will enlarge the solution space. We, therefore, assume that the change can finish within one period. The definition of f_i and $v_{dd,i}$ are refined as follows: f_i and $v_{dd,i}$ are the frequency and the voltage at the end of the ith period. If $f_{i-1} \neq f_i$ (or $v_{dd,i-1} \neq v_{dd,i}$), the frequency (or voltage) changes at the beginning of the ith period. We use a binary

variable, $\delta_i \in \{0, 1\}$, to indicate whether the processor changes frequencies (or voltages) at beginning of the ith period. If the frequency (or voltage) is changed, δ_i is one; otherwise, δ_i is zero. The processor does not change frequencies in the first period. Thus, we set δ_1 to be zero.

During the ith period, $\delta_i \Delta$ time is used for frequency scaling, so $t - \delta_i \cdot \Delta$ is left for processing jobs. All operations have to finish in this period. The constraint expressed in (8) is modified to include the scaling overhead

$$\sum_{l=1}^{m} a_{i,l} \cdot w_l \le (t - \delta_i \cdot \Delta) f_i \qquad \forall i \in \mathcal{N}. \tag{12}$$

In summary, the problem is to minimize energy for a processor with s frequencies and y voltages for m jobs

$$\min \sum_{i=1}^{n} (v_{dd,i})^2 \cdot f_i \tag{13}$$

under the following constraints:

$$\begin{aligned}
&\sum_{l=1}^{m} a_{i,l} \cdot w_l - (t - \delta_i \cdot \Delta) f_i \le 0 && \forall i \in \mathcal{N} \\
&\left(\sum_{i=1}^{k} a_{i,l} - \sum_{i=1}^{k} a_{i,l+1} \right) - b_l \le 0 && \forall k \in \mathcal{N}, \\
& && \forall l \in \mathcal{M} - \{m\} \\
&\sum_{i=1}^{k} a_{i,1} \ge \sum_{i=1}^{k} a_{i,2} \ge \cdots \ge \sum_{i=1}^{k} a_{i,m} = k && \forall k \in \mathcal{N} \\
&\delta_i = \begin{cases} 0, & \text{if } f_{i-1} = f_i \\ & \text{and } v_{dd,i-1} = v_{dd,i} \\ 1, & \text{otherwise} \end{cases} && \forall i \in \mathcal{N} \\
&f_i \in \{\phi_1, \phi_2, \ldots, \phi_s\} && \forall i \in \mathcal{N} \\
&v_{dd,i} \in \{v_1, v_2, \ldots, v_y\} && \forall i \in \mathcal{N}.
\end{aligned} \tag{14}$$

Example 4: Consider Example 3 again with $n = 4$. If $\Delta = 0$, the minimum energy is 10 if the frequencies are set to $\langle 4, 1, 4, 1 \rangle$. When Δ is nonzero, this assignment is invalid because j_1 and j_2 cannot execute twice in the third period after the frequency change. If $\Delta = 1$, the minimum energy is 11 when the frequencies are $\langle 4, 1, 4, 2 \rangle$. ♢

F. Summary

This section formulates power reduction as mathematical programming problems. This is a general formulation to minimize the energy of a processor with finite frequencies and voltages for running a program under timing and resource constraints. The formulation can consider different variations of the same problem, including multiple frequencies (or voltages) and scaling overhead. We can also formulate the problem to find the buffer requirements when the available energy is fixed. A large amount of literature has been devoted to solving mathematical programming more efficiently [5], [23], [24], [34], [39]. One major challenge of this approach is the large number of equations because the value of n represents the number of frames and a typical MPEG movie has thousands of frames. In the next section, we present a different approach to solve the problem based on graph-walking techniques. Our method can significantly reduce the computation for finding optimal solutions. In particular, it efficiently finds frequency assignments for many periods (large n). Furthermore, our method can handle sporadic jobs more easily.

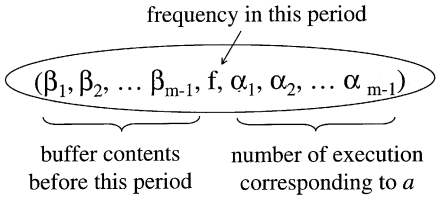


Fig. 11. Encoding of a vertex.

VI. Frequency Scaling by Graph Walking

The energy-minimization problem has additional structures that allow us to solve it more efficiently. In fact, there are only finite choices in each period, so eventually the assignments of f, v_{dd} and a will be cyclic when n is large. This section explains how to find such a cycle. This section is divided into three parts. First, we construct a finite directed graph to represent all frequency assignments. Second, we show that there is a repeating subwalk in any long walk of the graph. Finally, we demonstrate how to use the graph to find frequency assignments for energy minimization.

A. Assignment Graph

An assignment graph contains all possible choices for frequency assignments. Each vertex encodes the current state: it contains the frequency of the processor, the amount of data stored in buffers, and how the data will be consumed or refilled. In other words, the graph represents the state space of frequency assignments.

1) Vertices: Let $G = (\mathcal{V}, \mathcal{E})$ be a directed graph: $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Each vertex encodes the states of the buffers, the frequency, and the execution of each job. A vertex is identified by a vector of $2m - 1$ elements: $(\beta_1, \beta_2, \ldots, \beta_{m-1}, f, \alpha_1, \alpha_2, \ldots, \alpha_{m-1})$, here m is the number of jobs. Fig. 11 illustrates the encoding of a vertex. In this encoding, β_l ($l \in \{1, 2, \ldots, m - 1\}$) indicates the amount of data stored in buffer b_l before the period; f is the frequency in the period (or the frequency after a change). The value of α_l indicates how many times j_l executes; it corresponds to $a_{i,l}$ defined in the previous section. All βs and αs are integers. We calculate the value of i by traversing vertices, as explained later. Each vertex v has a cost $c(v)$. The cost is the frequency of the vertex, i.e., $c(v) = f$ and all costs are positive. Since a vertex represents a period, we can use "vertex" and "period" interchangeably. Table III lists the symbols and their meanings used in this section. We can also include voltage scaling by changing the vertex encoding: the value of f is changed to $(v_{dd})^2 f$. The new encoding method increases the graph size because one frequency may have multiple voltage settings. However, including voltage scaling does not affect the graph's properties. For simplicity, we consider frequency scaling only in the rest of this section.

TABLE III
SYMBOLS AND MEANINGS FOR ASSIGNMENT GRAPHS

$c(v)$	cost of vertex v, the frequency of v
$v_1 \Rightarrow v_2$	v_1 and v_2 are connected, or $(v_1, v_2) \in \mathcal{E}$
$\mathcal{W}$	a walk, general format: $< v_1, v_2, ..., v_n >$
$c(\mathcal{W})$	cost of walk $\mathcal{W}$
$\mathcal{W}_n(v)$	a minimum-cost walk from v and visits n vertices
$\mathcal{W}(v_a, v_b)$	a minimum-cost walk between vertices v_a and v_b
$\star$	concatenate two walks
$\varphi(v)$	unused operation of vertex v

Example 5: For Example 2, the processor has two frequencies, so f can be 1 or 2. The buffer can be filled or empty, so β is 0 or 1. If the processor is at frequency 1, j_1 and j_2 cannot execute within one period; consequently, α is zero when f is 1. The assignment graph includes five vertices: (1, 2, 0), (1, 1, 0) (0, 2, 1), (1, 2, 1), and (0, 2, 2). ♢

Now, we compute an upper bound of the number of vertices. First, we consider the possible values of β_1. Before one period starts, the buffer between j_1 and j_2 may have zero, one, two, …, or b_1 items; there are b_1+1 possible values for β_1. Similarly, β_2 has b_2+1 possible values and β_l has b_l+1 possible values. The processor has s frequencies, so f has s choices. At frequency ϕ_1, job j_1 can execute at most $\lfloor (t\cdot\phi_1 - w_m)/w_1 \rfloor$ times. This is an upper bound for α_1. It is an upper bound because the range for α_1 may decrease at a lower frequency. We call this upper bound $\overline{\alpha_1}$. The value of α_1 is between zero and $\overline{\alpha_1}$, so there are $\overline{\alpha_1}+1$ options. We can find upper bounds for other αs in the same way. The following formula is an upper bound of the size of an assignment graph:

$$\begin{aligned}&(b_1+1)\times(b_2+1)\times\cdots\times(b_{m-1}+1)\\&\times s\times(\overline{\alpha_1}+1)\times(\overline{\alpha_2}+1)\cdots\times(\overline{\alpha_{m-1}}+1)\\&\quad= s\times\prod_{l=1}^{m-1}(b_l+1)(\overline{\alpha_l}+1).\end{aligned}\tag{15}$$

This is a loose upper bound because we have not removed invalid vertices. There are three types of invalid vertices; they violate timing, resource, or precedence constraints.

Example 6: For Example 3, at frequency 1, j_1 and j_2 cannot execute in a single period because this violates the timing constraint specified by (8). The vertex (•, •, 1, 1, 1) is invalid regardless of the value for •. For the same example, vertex (0, 1, 4, 2, 2) starts with one frame in buffer b_2 and executes j_2 twice. Since only one frame is consumed by j_3, two frames have to be stored in buffer b_2 at the end of this period. However, b_2 can store only one frame. Therefore, (0, 1, 4, 2, 2) overflows the buffer and is an invalid vertex. Vertex (0, 0, 4, 0, 2) violates the precedence constraint because j_2 executes twice but the buffer between j_1 and j_2 is empty and j_1 does not execute. ♢

Timing constraints require the processor to operate at a frequency high enough to finish all scheduled jobs. The timing constraint of vertex $(\beta_1, \beta_2, \ldots, \beta_{m-1}, f, \alpha_1, \alpha_2, \ldots, \alpha_{m-1})$ is specified below. It is equivalent to the constraint specified in (8), namely $\sum_{l=1}^{m} \alpha_l \cdot w_l \le f \cdot t$. Resource constraints state that buffers cannot overflow. Before a new period starts, there are β_l items (or frames) in the buffer between j_l and j_{l+1}. In this period, j_l executes α_l times and j_{l+1} executes α_{l+1} times. Before this period ends, $\beta_l + \alpha_l - \alpha_{l+1}$ items must be stored in this buffer and these items cannot exceed the buffer size. This is equivalent to the constraint specified in (10)

$$\beta_l + \alpha_l - \alpha_{l+1} \le b_l \qquad l \in \mathcal{M} - \{m\}. \tag{16}$$

We define α_m as one because one frame is displayed each period. Finally, precedence constraints prevent buffer underflow. Since j_{l+1} executes α_{l+1} times, there must be enough data either from the buffer or produced by j_l. We rewrite the constraint in (9) for the vertices in the assignment graph

$$\beta_l + \alpha_l \ge \alpha_{l+1} \qquad l \in \mathcal{M} - \{m\}. \tag{17}$$

Example 7: In Example 3, either buffer can accommodate one item, so $(b_1+1) = (b_2+1) = 2$. There are three frequencies. Job j_1 can execute at most $(11\times 4-4)/12 = 3$ times and j_2 can execute at most $(11\times 4-4)/8 = 5$ times. The graph has $2\times 2\times 3\times 4\times 6 = 288$ vertices by (15). There are only 21 valid vertices after invalid vertices are removed. ♢

2) Starting Vertices: Since all buffers are empty at the beginning, the first $m-1$ elements in the encoding must be zero. Let $(0, 0, \ldots, 0, f, \alpha_1, \alpha_2, \ldots, \alpha_{m-1})$ be the first vertex. The value for f and the values of αs have to satisfy the following conditions based on (14), except that δ is always zero for the first period

$$\begin{aligned}f\cdot t &\ge w_m + \sum_{l=1}^{m-1}\alpha_l\cdot w_l\\ \alpha_l - \alpha_{l+1} &\le b_l \qquad l\in\{1, 2, \ldots, m-2\}\\ \alpha_1 &\ge \alpha_2 \ge \cdots \ge \alpha_{m-1} \ge 1.\end{aligned}\tag{18}$$

Any vertex that satisfies these conditions can be a starting vertex. After we remove invalid vertices, any vertex with the $(0, 0, \ldots, 0, f, \alpha_1, \alpha_2, \ldots, \alpha_{m-1})$ format is a valid starting vertex. We will use v^* as one starting vertex. If a vertex cannot be reached from any starting vertices, it is eliminated from the assignment graph.

3) Edges: An edge $(v_1, v_2) \in \mathcal{E}$ connects two vertices v_1 and v_2. It indicates a transition from state v_1 to state v_2 after one period. A transition from v_1 to v_2 is represented as $v_1 \Rightarrow v_2$. We call v_1 a *precedessor* of v_2 and v_2 a *successor* of v_1. There is at most one edge between two vertices. Some transitions are prohibited. There are two types of invalid transitions. The first type violates continuity conditions. Suppose $v_1 = (\beta_1, \beta_2, \ldots, \beta_{m-1}, f, \alpha_1, \alpha_2, \ldots, \alpha_{m-1})$, $v_2 = (\beta'_1, \beta'_2, \ldots, \beta'_{m-1}, f', \alpha'_1, \alpha'_2, \ldots, \alpha'_{m-1})$, and $v_1 \Rightarrow v_2$. The continuity conditions require that the data stored in the buffers remain the same at the end of the first period (leaving v_1) and at the beginning of the next period (entering v_2). Before the period represented by v_1 starts, there are β_l items in the lth buffer. During v_1, α_l more items are added to the buffer, and α_{l+1} of these items are consumed by job j_{l+1}. Consequently, there are $\beta_l + \alpha_l - \alpha_{l+1}$ items left before the period represented by v_2 starts. The following formula expresses this continuity condition:

$$\beta'_l = \beta_l + \alpha_l - \alpha_{l+1} \qquad l \in \{1, 2, \ldots, m-1\}. \tag{19}$$

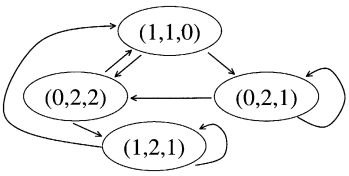


Fig. 12. Assignment graph for Example 2.

Example 8: In Example 3, (0, 0, 4, 2, 2) ⇒ (1, 0, 2, 0, 1) is an invalid transition. Before the period represented by the first vertex begins, buffer b_1 is empty. The first vertex executes both j_1 and j_2 twice, so b_1 is still empty. However, the following vertex starts with nonempty buffer b_1. This violates continuity principle. ♢

The second type of invalid transitions violates timing constraints. Consider Example 4, which includes scaling overhead. It takes one time unit to change the frequencies; therefore, j_1 and j_2 cannot execute twice at frequency 4 if the previous frequency is different from 4. In other words, (0, 1, 1, 0, 0) ⇒ (0, 0, 4, 2, 2) is an invalid transition.

4) Merging Vertices: After constructing the graph, we can further reduce its size by merging vertices. Two vertices can merge if they have identical predecessors and successors. This happens when two vertices differ only in their frequencies; the merged vertex uses the lower frequency because it suffices to use the lower frequency. For example, (0, 1, 2, 0, 0) can merge with (0, 1, 1, 0, 0) in Example 3. Since these vertices have the same inward and outward edges, they perform identical operations. Consequently, it is unnecessary to use a higher frequency if a lower frequency is sufficient.

Example 9: Fig. 12 shows the assignment graph for Example 2 after invalid vertices are removed and equivalent vertices are merged. This graph has one vertex with frequency 1 and three vertices with frequency 2. Vertex (0, 2, 1) can reach itself. This means the processor can keep running at frequency 2 and executing j_p once every period. Both successors of (1, 1, 0) have frequency 2. This means that the processor can run at frequency 1 for only one period. Then, it has to run at frequency 2 for at least one period before entering (1, 1, 0) again. ♢

5) Walks: A *walk* $\mathcal{W}$ of a graph is a sequence of vertices $\mathcal{W} = \langle v_1, v_2, \ldots, v_n\rangle$ such that $(v_i, v_{i+1}) \in \mathcal{E}$ for any $i \in \{1, 2, \ldots, n-1\}$. A walk is a sequence of assignments of frequencies (f) and executions (α) by the vertices. Walk $\mathcal{W}$ *visits* a vertex v if v appears in the sequence. Vertices v_2, v_3, …, v_{n-1} are *intermediate* vertices in the walk. The length of a walk is the number of vertices in the sequence,[2] or n. A *closed walk* of v_1 starts and ends at the same vertex: $v_1 = v_n$ [3], [12]. If $(v_i, v_i) \in \mathcal{E}$, the walk $\langle v_i, v_i\rangle$ is called a *loop*. A *subwalk* is a walk contained in a longer walk; for example, $\langle v_i, v_{i+1}, \ldots, v_j\rangle$ is a subwalk of $\langle v_1, v_2, \ldots, v_n\rangle$ if $1 \le i \le j \le n$. This subwalk starts from v_i, ends at v_j, and visits $j - i + 1$ vertices. A walk is a *path* if all vertices are distinct [3], [12]. Graph walking has been applied to a wide range of problems, such as finding the resistance in an electric network and the locations for servers [10], [41]. Two walks can be concatenated. We use $\star$ as the concatenation operator: walk $W_1 = \langle v_1, v_2, \ldots, v_n\rangle$ is concatenated with walk $W_2 = \langle u_1, u_2, \ldots, u_m\rangle$, written as $W_1 \star W_2$. The result is a longer walk, $W_1 \star W_2 = \langle v_1, v_2, \ldots, v_n, u_1, u_2, \ldots, u_m\rangle$. Walks can concatenate if $(v_n, u_1) \in \mathcal{E}$.

Example 10: Fig. 13 shows three examples of walks. The first is a walk from v_1 to v_5. The second walk, $\langle v_1, v_2, v_3, v_4, v_2, v_5\rangle$ contains a closed walk, $\langle v_2, v_3, v_4, v_2\rangle$. The third walk $\langle v_1, v_2, v_3, v_4, v_2, v_5, v_1\rangle$ contains two closed walks,[3] one starting from v_1 and the other starting from v_2. ♢

Fig. 12 has a loop of vertex (0, 2, 1). This is not incidental. We assume jobs are scheduleable at the highest frequency, so there is always a loop of vertex $(0, 0, \ldots, \phi_1, 1, 1, \ldots)$ here ϕ_1 is the highest frequency. This is equivalent to executing each job once per period and storing no additional data in the buffers.

6) Cost of a Walk: The cost of a walk, $c(\mathcal{W})$, is the sum of the cost of each vertex: $c(\mathcal{W}) = \sum_{i=1}^{n} c(v_i)$. The average cost is defined as

$$\frac{c(\mathcal{W})}{n} = \frac{1}{n}\sum_{i=1}^{n} c(v_i). \tag{20}$$

A minimum-cost walk can be defined for two different conditions: 1) A walk starts from a given vertex (v_1) and visits a given number (n) of vertices. This walk is represented as $\mathcal{W}_n(v_1) = \langle v_1, v_2, \ldots, v_n\rangle$. The ending vertex ($v_n$) is not specified. The cost is expressed as $c(\mathcal{W}_n(v_1))$. 2) A walk starts from a given vertex (v_a) and ends at another given vertex (v_b). We use $\mathcal{W}(v_a, v_b) = \langle v_a, v_1, \ldots, v_n, v_b\rangle$ to represent such a walk. The number of visited vertices is not specified. The cost is expressed as $c(\mathcal{W}(v_a, v_b))$.

Example 11: Fig. 12 has several walks, including the following.

- two loops: (0, 2, 1) ⇒ (0, 2, 1) and (1, 2, 1) ⇒ (1, 2, 1).
- (1, 1, 0) ⇒ (0, 2, 1) ⇒ (0, 2, 2).
- (1, 1, 0) ⇒ (0, 2, 2) ⇒ (1, 2, 1).
- (1, 1, 0) ⇒ (0, 2, 2) ⇒ (1, 1, 0). This walk has the minimum average cost among all walks in this example. ♢

B. Energy Minimization by Assignment Graphs

1) Minimum-Cost Subwalks: Because a walk is an assignment of frequencies, a minimum-cost walk is equivalent to an assignment that minimizes the energy consumption. This section finds a minimum-cost walk for n periods, namely, the cost for visiting n vertices.

Theorem 1: Suppose $\mathcal{W}_n(v_1) = \langle v_1, v_2, \ldots, v_n\rangle$ is a minimum-cost walk from v_1 and visits n vertices. A subwalk $\langle v_i, v_{i+1}, \ldots, v_j\rangle$ of $\mathcal{W}_n(v_1)$ is a minimum-cost walk from v_i to v_j and visits $j - i + 1$ vertices.

Proof: This can be proven by contradiction. The cost of $\mathcal{W}_n(v_1)$ is $\sum_{k=1}^{n} c(v_k)$, or $\sum_{k=1}^{i-1} c(v_k) + \sum_{k=i}^{j} c(v_k) + \sum_{k=j+1}^{n} c(v_k)$. If the walk $\langle v_i, v_{i+1}, \ldots, v_j\rangle$ is not minimum-cost, then we can find another walk $\langle v_i, v'_{i+1}, \ldots, v'_{j-1}, v_j\rangle$ that starts from v_i, ends at v_j, visits the same number of vertices, and has a lower cost: $\sum_{k=i+1}^{j-1} c(v'_k) < \sum_{k=i+1}^{j-1} c(v_k)$. Replacing

[2]Some texts use the number of edges as the length of a walk.

[3]We assume the walk stops after it visits v_1 twice.

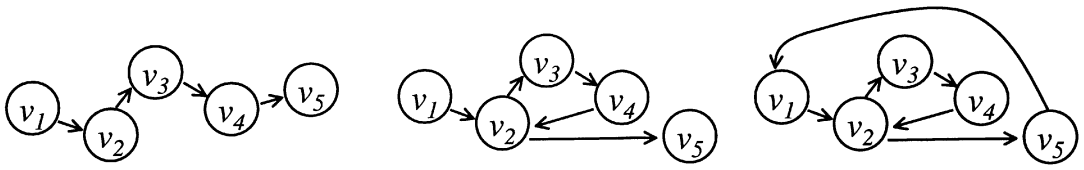


Fig. 13. Examples of walks.

$\langle v_i, v_{i+1}, \ldots, v_j \rangle$ by $\langle v_i, v'_{i+1}, \ldots, v'_{j-1}, v_j \rangle$ will reduce the cost of the original walk $\mathcal{W}_n(v_1)$. This contradicts the premise that $\mathcal{W}_n(v_1)$ is a minimum-cost walk. Therefore, $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a minimum-cost walk from v_i to v_j and visits $j - i + 1$ vertices. ♢

This theorem is similar to finding shortest subpaths while computing a shortest path between two vertices in a graph [11]. A minimum-cost walk differs from a shortest path in three ways.

1) It specifies the number of vertices, not the ending vertex.
2) Its cost is determined by the vertices, $\sum_{i=1}^{n} c(v_i)$, not by the weights on the edges.
3) It allows visiting the same vertex multiple times.

The methods presented in [11] compute the shortest paths between two given vertices. Because the power-reduction problem is different, we approach this problem by modifying the methods in [11].

2) Finding Subwalk Recursively: Suppose $\mathcal{W}_n(v_1)$ is $\langle v_1, v_2, \ldots, v_n \rangle$, then $c(\mathcal{W}_n(v_1)) = c(v_1) + c(v_2) + \cdots + c(v_n)$. By definition, $\mathcal{W}_1(v_1)$ is $\langle v_1 \rangle$ and $c(\mathcal{W}_1(v_1))$ is $c(v_1)$. We can divide $\mathcal{W}_n(v_1) = \langle v_1, v_2, \ldots, v_n \rangle$ into two walks: $\langle v_1 \rangle$ and $\langle v_2, v_3, \ldots, v_n \rangle$, here $(v_1, v_2) \in \mathcal{E}$. The cost of $\mathcal{W}_n(v_1)$ can be computed by

$$\begin{aligned} c(\mathcal{W}_n(v_1)) &= \sum_{i=1}^{n} c(v_i) \\ &= c(v_1) + \sum_{i=2}^{n} c(v_i) \\ &= c(v_1) + c(\mathcal{W}_{n-1}(v_2)) \qquad (v_1, v_2) \in \mathcal{E}. \quad (21) \end{aligned}$$

Fig. 14 illustrates this concept. This is a recursive relation: each time, we reduce the length of the walk by one. Equation (21) computes $c(\mathcal{W}_n(v_1))$ by reducing the length of the walk through the recursive relation. Since there may be multiple choices for v_2, it takes exponential time to find a minimum-cost walk by (21) [16]: $O(\chi^n)$, χ is the average number of successors of each vertex and $\chi \geq 1$. We clearly need a more efficient method.

3) Memorization of Subwalks: We can reduce the time complexity by memorizing shorter walks to construct long walks. If we already know the minimum-cost walk $\mathcal{W}_{n-1}(v_2)$, it is unnecessary to compute it again. Memorization eliminates computing the same subwalks multiple times. The algorithm in Fig. 15 computes a minimum-cost walk of $\mathcal{W}_n(v_1)$ by memorizing shorter walks. For each iteration of wlength, at most $|\mathcal{V}|^2$ vertices are visited and the execution time is $O(n|\mathcal{V}|^2)$. Memorization reduces the complexity from exponential to linear in n.

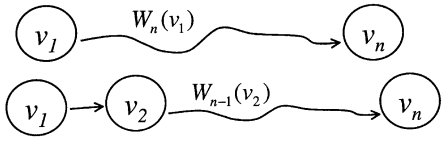


Fig. 14. Divide $\mathcal{W}_n(v_1)$ into two subwalks.

```
MinimumCostWalk(input graph: G = (V, E), integer: n)
/*  W_i(v): minimum-cost walk from v with length i
    wcost(v, i): cost of W_i(v)
    n: maximum length of walks */
begin
    /* initialization */
    for each v ∈ V
        wcost(v, 1) := c(v);
        W_1(v) := < v >;

    for (wlength := 2; wlength ≤ n; wlength ++)
        for each v_1 ∈ V
            wcost(v_1, wlength) := ∞; /* initialize */
            for each v_2 ∈ V and (v_1, v_2) ∈ E
                newcost := c(v_1) + wcost(v_2, wlength − 1);
                if (wcost(v_1, wlength) > newcost)
                    wcost(v_1, wlength) := newcost;
                    W_wlength(v_1) := < v_1 > ⋆ W_wlength−1(v_2);
                    /* ⋆: concatenation of two walks */
end
```

Fig. 15. Find minimum-cost walks by (21).

C. Efficient Assignments

Even though MinimumCostWalk has complexity $O(n|\mathcal{V}|^2)$, there are still two problems. First, the time is linear in n even though the graph size is independent of n. Second, the algorithm in Fig. 15 computes $\mathcal{W}_i(v)$ for every value of i, whereas we are interested only in $i = n$. Because assignment graphs are finite, we can compute minimum-cost walks even more efficiently for large n. This section explains how to find minimum-cost walks efficiently when $n \gg |\mathcal{V}|$.

1) Minimum-Cost Walks Between Two Vertices: Based on the Floyd–Warshall algorithm for finding the shortest paths in a graph [11], we can find a minimum-cost walk between vertex v_a and vertex v_b. The algorithm is called MinimumCostWalk2V and is shown in Fig. 16. This algorithm has complexity $O(|\mathcal{V}|^3)$ because of the nested iteration.

2) Pigeonhole Principle: Suppose there are n pigeons and m holes. We want to assign these n pigeons to the m holes. If there are more pigeons than holes ($n > m$), at least one hole must have two or more pigeons. This is called the *pigeonhole principle* [15]. Consider another example. There are m balls

```
MinimumCostWalk2V(input graph: G = (V, E))
/*  W(v_a, v_b): minimum-cost walk from v_a to v_b
    W^-(v_a, v_b): W(v_a, v_b) without visiting the last vertex, v_b
        if W(v_a, v_b) =< v_a, v_1, v_2, ..., v_n, v_b > then
        W^-(v_a, v_b) =< v_a, v_1, v_2, ..., v_n >
    mcost(v_a, v_b): cost of W(v_a, v_b)
    nW(v_a, v_b): number of vertices in W(v_a, v_b) */
begin
    for each v_a ∈ V
        for each v_b ∈ V
            if (v_a, v_b) ∈ E
                W(v_a, v_b): = < v_a, v_b >;
                mcost(v_a, v_b): = c(v_a) + c(v_b);
                nW(v_a, v_b): = 2;
            else
                W(v_a, v_b): = <>;
                mcost(v_a, v_b): = ∞;
                nW(v_a, v_b): = ∞;
    for each v_c ∈ V /* intermediate vertex */
        for each v_a ∈ V
            for each v_b ∈ V
                costpassc := mcost(v_a, v_c) + mcost(v_c, v_b) − c(v_c);
                /* subtract c(v_c) because it is counted twice */
                if (mcost(v_a, v_b) > costpassc)
                    mcost(v_a, v_b) := costpassc;
                    nW(v_a, v_b) := nW(v_a, v_c) + nW(v_c, v_b) − 1;
                    W(v_a, v_b): = W^-(v_a, v_c) ⋆ W(v_c, v_b);
end
```

Fig. 16. Find minimum-cost walks between two vertices.

labeled as $1, 2, \ldots, m$ stored in a box. Every minute, we select one ball from the box, record its number, and put it back to the box. After n minutes, we have seen n balls. If $n > m$, one number between 1 and m must occur two or more times, according to the pigeonhole principle. The pigeonhole principle can be applied to walks. If a walk visits n vertices and n is larger than the number of vertices in the graph ($n > |\mathcal{V}|$), at least one vertex must be visited twice or more often. Hence, there is at least one closed walk.

3) Redefining Closed Walk: A closed walk has the format $\langle v_1, v_2, \ldots, v_n \rangle$ where $v_n = v_1$. In the rest of this paper, we restrict closed walks so that v_1 is visited exactly twice and no other vertex is visited twice or more often: $v_i \neq v_j$ if $1 \leq i < j \leq n$ except $i = 1$ and $j = n$. We call such closed walks $\mathcal{CWALK}$s. According to the pigeonhole principle, any closed walk in $G = (\mathcal{V}, \mathcal{E})$ visits at most $|\mathcal{V}| + 1$ vertices (v_1 is counted twice). Fig. 17 is an algorithm for finding all $\mathcal{CWALK}$s that have minimum average costs. The average cost of a walk is defined by (20). If two closed walks have the same average cost, this algorithm keeps the shorter one. It first finds all minimum-cost walks of lengths up to $|\mathcal{V}| + 1$. Then, it determines whether the walks are closed and computes the average cost; finally, it keeps only closed walks with minimum average costs. Since the jobs are scheduleable, there is at least one trivial solution: a loop of vertex $(0, 0, \ldots, \phi_1, 1, 1, \ldots)$, where ϕ_1 is the highest frequency. Since $n = |\mathcal{V}| + 1$, it takes $O(|\mathcal{V}|^3)$ to call Minimum-Cost Walk. For each vertex, $\mathrm{wlength}$ changes from 2 to $|\mathcal{V}|+1$ and it takes $\mathrm{wlength}$ to compute the average cost of $\mathcal{W}_{\mathrm{wlength}}(v)$. It takes $O(|\mathcal{V}|^2)$ time for each vertex. Hence, this algorithm takes $O(|\mathcal{V}|^3)$ time.

```
FindClosedWalk(input graph: G = (V, E))
/*  CWALK(v): closed walk of v with minimum average cost
    cwcost(v): the cost of CWALK(v)
    nwalk(v): the length of CWALK(v) */
begin
    MinimumCostWalk(G, |V| + 1);
    for each v ∈ V
        /* initialization */
        CWALK(v) := empty;
        cwcost(v) := ∞;

        for (wlength := 2; wlength ≤ |V| + 1; wlength ++)
            v' := last vertex of W_wlength(v);
            if (cwcost(v)/nwalk(v) > wcost(v, wlength)/wlength) /* smaller average cost */
                if (v = v') /* a closed walk */ and
                (W_wlength(v) visits v exactly twice) and
                (no other vertex is visited more than once)
                    CWALK(v) := W_wlength(v);
                    cwcost(v) := wcost(v, wlength);
                    nwalk(v) := wlength;
                else
                    CWALK(v) := empty;
                    cwcost(v) := ∞;
end
```

Fig. 17. Find the closed walks of minimum average costs.

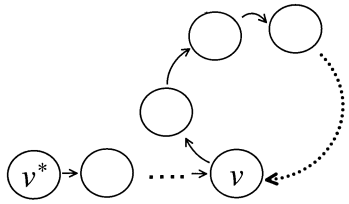


Fig. 18. A walk of infinite length must repeat a closed walk indefinitely.

4) Walks of Infinite Length: After finding the minimum-cost $\mathcal{CWALK}$s, we can easily find an infinite-length walk with the minimum average cost. When n approaches infinity, a minimum-cost walk starts from one starting vertex, defined in Section VI-A-2, reaches a closed walk, and repeats this closed walk. Fig. 18 illustrates such a walk. This closed walk is chosen because it has the minimum average cost, defined as

$$\min_{v \in \mathcal{V}} \frac{cw\mathrm{cost}(v)}{\mathrm{nwalk}(v)}. \tag{22}$$

If vertex v is not a starting vertex, we can find a minimum-cost walk that connects one v^* to v by MinimumCostWalk2V. Since FindClosedWalk takes $O(|\mathcal{V}|^3)$ time, $O(|\mathcal{V}|^3)$ time is required to find a walk of infinite length with the minimum average cost. Because the walk is infinite, the "initial cost" from v^* to v can be ignored. A natural question is whether this closed walk is reachable from a starting vertex. Since the jobs are scheduleable at the highest frequency, an infinite walk must be available. One trivial solution is the loop of the vertex $(0, 0, \ldots, \phi_1, 1, 1, \ldots)$, where ϕ_1 is the highest frequency. There may be other solutions that satisfy all constraints and require lower power consumptions. Our method finds these solutions with time complexity $O(|\mathcal{V}|^3)$ and is independent of n.

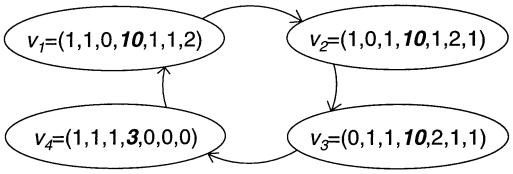


Fig. 19. The walk for four jobs with 80% processor utilization.

According to the pigeonhole principle, frequency assignments must form a closed walk for a workload with an infinite time horizon. We need to emphasize that our method does not have to know the length of the closed walk in advance. In contrast, using mathematical programming, one has to determine this length in advance and select an n large enough for the inequalities in (14). A typical MPEG movie contains thousands of frames, so we can reasonably approximate it with infinite frames. No real movie has infinite frames. Appendix explains how to find a minimum-cost walk with a finite length. A finite walk is different in three ways: 1) the initial cost needs to be considered; 2) it has a "tail" that may not be a complete closed walk; and 3) the cost of the tail needs to be considered.

Example 12: A processor has five frequencies: 10, 7, 5, 4, and 3; a program keeps the processor 80% utilized at frequency 10. There are four jobs with equal numbers of operations: each job takes 20% of the total time in a period. Without any buffer, the processor is idle 20% of the time in each period. If there is one buffer between two jobs, a low-power schedule is used, as presented in Fig. 19. In each period at frequency 10, one of the buffers is filled; then, the processor runs at frequency 3 to reduce energy. The average frequency is 8.25, or 3% above optimal. This walk also shows that some frequencies (7, 5, and 4) are not used. ♢

In summary, we demonstrate how to use graph-walking techniques for energy minimization. This method is based on the fact that only finite frequencies and buffers are available. The solution space is enumerated as a directed graph and the graph size is trimmed by removing invalid vertices and edges. A efficient algorithm (cubic time complexity) is proposed to find a minimum-energy walk for frequency assignments. While our method is developed for frequency scaling, it can be easily extended for voltage scaling. When voltage scaling is considered, the cost of each vertex is the power of a period: $(v_{dd})^2 f$. Including voltage scaling increases the graph size but it does not affect the properties of the graph. Consequently, the algorithm for finding the minimum-cost walk is still applicable.

VII. Response Time of Sporadic Jobs

The previous section considered periodic jobs and showed how frequency scaling and data buffering can reduce the energy consumption. The optimal assignments of frequencies are determined by a graph-based algorithm. This section computes the response time of a sporadic job in the presence of periodic jobs. We show how to calculate the response time of a sporadic job if it arrives at a period represented by a vertex in a walk. For simplicity, this section ignores scaling overhead. We also assume that a sporadic job completes before another sporadic job arrives.

The following scenario is an example to illustrate the mixture of periodic and sporadic jobs. When a user is watching an MPEG movie, the movie creates periodic jobs. Occasionally, the user may move the mouse to a slider and adjust the volume; this movement creates a sporadic job. A desirable outcome consists of three parts: 1) the sporadic job is processed promptly; 2) the frame rate of the MPEG movie remains constant; and 3) the power consumption is minimized. When a sporadic job arrives, the processor has to execute additional operations. These operations can be executed in two ways. First, the sporadic job is executed with only the "spare" operations in each period. Alternatively, if the buffers are nonempty, data can be retrieved from them and some jobs do not have to execute. By draining the buffers, the sporadic job can finish earlier.

A. Unused Operations

Some time periods may have "unused" operations because these operations are not used to execute any of jobs j_1 to j_{m-1}. We use $\varphi(v)$ as the number of unused operations of vertex v. For vertex $v = (\beta_1, \beta_2, \ldots, \beta_{m-1}, f, \alpha_1, \alpha_2, \ldots, \alpha_{m-1})$, $\varphi(v)$ can be found by the following formula:

$$\varphi(v) = t \cdot f - \left(w_m + \sum_{l=1}^{m-1} \alpha_l \cdot w_l \right). \tag{23}$$

In this formula, $t \cdot f$ is the total number of allowed operations and $(w_m + \sum_{l=1}^{m-1} \alpha_l \cdot w_l)$ is the number of operations needed to execute the jobs that have to be executed this period. The difference between these two quantities difference determines how many additional operations can be conducted in this period.

Example 13: In this example, we represent unused operations as the percentage of a period at the highest frequency. If the processor is completely idle while running at the highest frequency, the unused operation is 100%. If the processor is idle but runs at half of the highest frequency, the unused operation is 50%.

Let us reconsider Example 12. If there is no sporadic job, the minimum energy is achieved by repeating a closed walk with four vertices. This solution is shown in Fig. 19. In the period represented by vertex $v_2 = (1, 0, 1, 10, 1, 2, 1)$, j_1 and j_3 execute once because $\alpha_1 = \alpha_3 = 1$. In the same period, j_2 executes twice because $\alpha_2 = 2$. The fourth job always executes once each period. Since each job takes 20% of the time, the total time required to execute these four jobs is $20\% + (1 + 2 + 1) \times 20\% = 100\%$. Consequently, $\varphi(v_2) = 0$ and no additional operation can be executed.

For the period represented by v_4, only j_4 executes because $\alpha_1 = \alpha_2 = \alpha_3 = 0$. Notice that the frequency is only 30% of the highest frequency; $\varphi(v_4) = 30\% - 20\% = 10\%$. Because this is insufficient to execute any of j_1, j_2, or j_3, it is unused. However, this period can execute a sporadic job if it needs half of the operations of j_4. ♢

Consider a sporadic job that needs w_r operations. Suppose the MPEG player still maintains the same output rate. The sporadic job can execute only by using unused operations in each vertex. The sporadic job can finish in one period if the number of unused operations is larger than this job's number of operations, or $\varphi(v) \geq w_r$. Suppose the sporadic job arrives at the beginning of a walk of n vertices: $\mathcal{W} = \langle v_1, v_2, \ldots, v_n \rangle$. The

```
ResponseTime(input vertex: v = (β_1, ..., β_{m-1}, f, α_1, ..., α_{m-1}),
             job: w_r)
/*  find how many periods (t) are needed to finish the sporadic
    job that needs w_r operations */
begin
    compute φ(v) by equation (23);
    if φ(v) ≥ w_r /* enough unused operations */
        return t;
    compute φ_b(v) by equation (27);
    if φ_b(v) ≥ w_r
        find one next vertex v' by the continuity condition (19);
        return t;
    /* the job will take more than t to execute */
    /* find the response time recursively */
    find one next vertex v' by the continuity condition (19);
    return t + ResponseTime(v', w_r − φ_b(v));
end
```

Fig. 20. Find the response time of a sporadic job.

sporadic job can finish within n periods if there are enough unused operations in these vertices. This condition is expressed by the following inequality:

$$\sum_{i=1}^{n} \varphi(v_i) \geq w_r. \tag{24}$$

B. Effects of Buffers

Equation (23) does use buffers to reduce the response time of a sporadic job. To execute j_m once each period, the data required by j_m may be obtained in one of the two ways: 1) from the buffer between j_m and j_{m-1} or 2) generated by job j_{m-1} in the same period. Job j_{m-1} has to execute only if the buffer between j_m and j_{m-1} is empty. Condition 1) means $\beta_{m-1} > 0$ and condition 2) means $\alpha_{m-1} > 0$. Together, $\beta_{m-1} + \alpha_{m-1}$ must be at least one so that j_m can execute in this period. We can generalize this relationship. Suppose job j_l executes in a period. Job j_{l-1} must execute in the same period if the buffer between j_l and j_{l-1} is empty ($\beta_{l-1} = 0$). We define an indicator function γ_l to determine whether job j_l has to execute

$$\gamma_l = \begin{cases} 1, & \text{if } \gamma_{l+1} = 1 \text{ and } \beta_l = 0 \\ 0, & \text{otherwise} \end{cases} \qquad l \in \mathcal{M} - \{m\}. \tag{25}$$

We define $\gamma_m = 1$ since job j_m executes once every period. Because the γs are the minimum requirements to keep the output rate, γ_l must be smaller or equal to α_l, or $\gamma_l \leq \alpha_l$. During this period, the minimum number of operations to sustain the constant output rate is

$$w_m + \sum_{l=1}^{m-1} w_l \cdot \gamma_l. \tag{26}$$

Let $\varphi_b(v)$ be the maximum number of operations available for a sporadic job when the effects of buffers are considered

$$\varphi_b(v) = t \cdot f - \left(w_m + \sum_{l=1}^{m-1} w_l \cdot \gamma_l \right). \tag{27}$$

Since $\gamma_l \leq \alpha_l$, $\varphi_b(v)$ must be larger than or equal to $\varphi(v)$. In other words, buffering allows the processor to spend more time on the sporadic job. The response time can be computed using the procedure presented in Fig. 20. This procedure first checks whether there are enough unused operations $(\varphi(v))$ in one period. Then, it checks whether the sporadic job can finish in one period by draining the buffers $(\varphi_b(v))$. If neither is successful, it recursively computes the response time by adding one period each time.

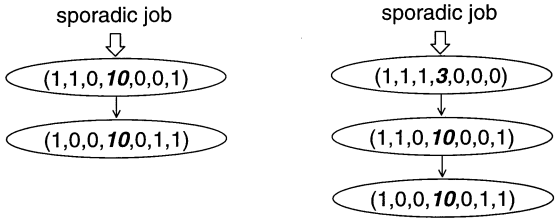


Fig. 21. A sporadic job finishes in two periods if it arrives at v_1 (left). It takes three periods if it arrives at v_4 (right).

After processing a sporadic job, the processor may reach a vertex which does not belong to a steady-state minimum-cost walk. For such a vertex, we can find a path to return to the minimum-cost walk. This can be computed in advance with $O(|\mathcal{V}|^3)$ time complexity using all-pairs shortest path algorithms presented in [11]. The vertex does not have to store the complete path returning to the minimum-cost walk. Instead, it needs to store only the next vertex of the path. The next vertex also stores only the following vertex of the path. The complete return path is available by following the chain of vertices until the steady-state minimum-cost walk is reached. Consequently, storing the return paths require $O(|\mathcal{V}|)$ memory.

Example 14: Consider Example 12 again for computing the response time of a sporadic job. Suppose a sporadic job needs one period at frequency 10 to complete. Without buffers, the job takes five periods to complete this job because the processor can spend only 20% of the time in each period on this job.

Now, let us consider how buffering reduces the response time. For vertex v_1 in Fig. 19, γ_3 equals one but γ_1 and γ_2 equal zero. Since only j_3 and j_4 have to execute, the processor can spend 60% of the time in this period for a sporadic job. Because j_1 and j_2 do not execute, the next vertex is different from v_2. Using the continuity conditions, we find one vertex to follow v_1; it is (1, 0, 0, 10, 0, 1, 1) as shown in Fig. 21. This vertex can spend 40% of the time executing the sporadic job. The sporadic job finishes in two periods, which is a 60% reduction from five periods. Similarly, we can compute the response time of the sporadic job if it arrives at v_4. The job takes three periods as shown in Fig. 21; this is 40% improvement with respect to the original five periods. Two periods are required to finish the sporadic job if it arrives at v_2 or v_3. On average, the response time of the sporadic job is $(2 \times 3 + 3)/4 = 2.25$ periods, which is a 55% improvement with respect to the original five periods. ♢

C. Timing Constraints of Sporadic Jobs

The previous section analyzed the average response time of a sporadic job. Using the same technique, we can determine whether it is possible to meet the timing constraint of a sporadic job. The timing constraint is the maximum acceptable execution time after a sporadic job arrives. In order to decide whether it is possible to finish a sporadic job j_r within n time periods, we have to find the shortest response time of j_r. The response time

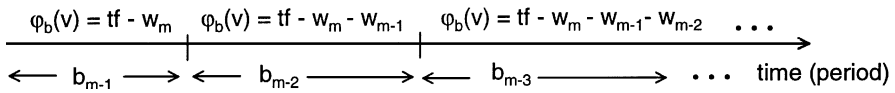


Fig. 22. $\varphi_b(v)$ decreases as more buffers become empty.

is shortest when all buffers are full and the processor is running at the highest frequency. Thus, we assume all buffers are full and the frequency is the highest when j_r arrives. We also assume that no sporadic job arrives before another sporadic job completes (because they are "sporadic"). In (27), the maximum number of operations occurs when $\gamma_1 = \gamma_2 = \cdots = \gamma_{m-1} = 0$: $\max \varphi_b(v) = t \cdot \phi_1 - w_m$, here t is the length of a period, ϕ_1 is the highest frequency, and w_m is the number of operations executed by job j_m. If $w_r/n > t \cdot \phi_1 - w_m$, then it is impossible to meet the timing constraint because there are insufficient operations in each period. Allocating more memory for buffers will not solve the problem; the only solution is to find a faster processor with higher ϕ_1.

The value of $\varphi_b(v)$ is $t \cdot \phi_1 - w_m$ only if the buffer between j_{m-1} and j_m has data. Since the buffer size is b_{m-1}, the buffer will become empty in b_{m-1} periods after j_r arrives. If $n \leq b_{m-1}$ and $w_r/n \leq t \cdot \phi_1 - w_m$, then j_r can finish in n periods. Job j_{m-1} does not have to execute during these n periods and j_m can still execute once each period. When $n > b_{m-1}$, the buffer between j_{m-1} and j_m becomes empty before j_r completes. Thus, j_{m-1} must execute $n - b_{m-1}$ times so that j_m can execute once each period. The maximum value of $\varphi_b(v)$ drops to $t \cdot \phi_1 - (w_m + w_{m-1})$ after b_{m-1} periods. Consequently, when $n \leq b_{m-1} + b_{m-2}$, j_r can finish in n periods if

$$w_r \leq b_{m-1} \times (t \cdot \phi_1 - w_m) + (n - b_{m-1}) \times (t \cdot \phi_1 - w_m - w_{m-1}). \quad (28)$$

As n becomes larger, more buffers become empty and the values of $\varphi_b(v)$ decrease. This condition is illustrated in Fig. 22. Following this analysis, we can derive the condition to finish j_r within n periods after j_r arrives:

$$\begin{aligned}
& n \leq b_{m-1} \\
& \quad \Rightarrow w_r \leq n(t \cdot \phi_1 - w_m) \\
& b_{m-1} < n \leq b_{m-1} + b_{m-2} \\
& \quad \Rightarrow w_r \leq b_m(t \cdot \phi_1 - w_m) + (n - b_{m-1}) \\
& \qquad \times \quad (t \cdot \phi_1 - w_m - w_{m-1}) \\
& \qquad \cdots \\
& \sum_{l=k}^{m-1} b_l < n \leq \sum_{l=k-1}^{m-1} b_l \\
& \quad \Rightarrow w_r < \sum_{l=k}^{m-1} b_l \left(t \cdot \phi_1 - \sum_{p=l+1}^{m} w_p \right) \\
& \qquad + \left(n - \sum_{l=k}^{m-1} b_l \right) \left(t \cdot \phi_1 - \sum_{p=k}^{m} w_p \right) \\
& n > \sum_{l=1}^{m-1} b_l \\
& \quad \Rightarrow w_r < \sum_{l=1}^{m-1} b_l \left(t \cdot \phi_1 - \sum_{p=l+1}^{m} w_p \right) \\
& \qquad + \left(n - \sum_{l=1}^{m-1} b_l \right) \left(t \cdot \phi_1 - \sum_{l=1}^{m} w_l \right) \quad (29)
\end{aligned}$$

where $1 < k < m$. The conditions in (29) indicate that enlarging the last buffer (larger b_{m-1}) is most effective because it is multiplied by the largest coefficient, $t \cdot \phi_1 - w_m$. This analysis further suggests how to calculate the minimum number of items stored in each buffer. In fact, buffers do not always have to remain full. As long as these conditions are satisfied, j_r is guaranteed to finish within n periods. This sets the lower limits of the buffer sizes. No existing scaling technique is able to analyze the relationship between buffer sizes and the response time of a sporadic job.

In the assignment graph, the β values of a vertex represent the numbers of items stored in buffers. Since (29) specifies the minimum number for each buffer, a vertex should be removed if its β values are too small. After excluding these vertices, a minimum-cost walk can meet all constraints of the mixed workloads and also achieves the minimum energy consumption. There is, however, an initial walk after the system starts and the buffers are being filled: during this period the timing constraint of a sporadic job cannot be met. The minimum time period to fill all buffers can be calculated by finding a shortest path from one starting vertex v^* to a vertex whose encoding is $(b_1, b_2, \ldots, b_{m-1}, \bullet, \bullet, \ldots, \bullet)$. Algorithms for finding shortest paths between vertices can be found in [11].

In summary, this section describes how to compute the response times of sporadic jobs based on the assignment graphs developed in Section VI. We explain how to take advantage of the buffered data to reduce the response times without affecting the on-time constraints of periodic jobs. We also analyze the response time of a sporadic job.

VIII. Experiments

In this section, we first describe our experimental environment. Then, we show the power savings of a synthesized workload. We describe an MPEG player modified to scale frequencies for power reduction. We also discuss how buffer sizes and job sizes affect power saving.

A. Experimental Setup

We set up a system to measure power saving by frequency scaling. The system was composed of a palm-size computer using Intel's StrongARM processor [21] (also called *Assabet*). The processor has eleven frequencies, between 59 and 206 MHz. It has a 320 × 240 touchscreen, 16-MB SDRAM, and a Compact Flash interface. This interface can be used for networking. This system runs Linux ported for ARM

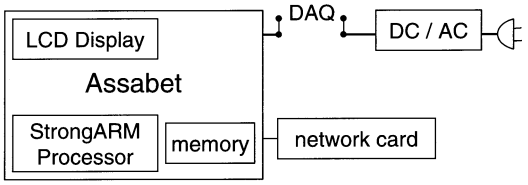


Fig. 23. Setup for our experiments.

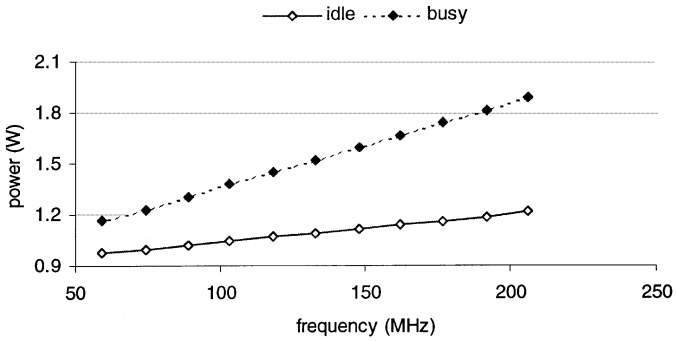


Fig. 24. Power consumption at different frequencies.

processor [2]. We used a National Instrument Data Acquisition Card (DAQ) to measure the dc current from the ac/dc adapter. This is the total power consumption of the whole system and directly affects battery lifetime. Fig. 23 illustrates the setup for our experiments. Assabet supports frequency scaling but it does not support voltage scaling. It takes approximately 2 μs to change frequencies.[4] Assabet can also connect to a companion board, called *Neponset*, that provides interfaces for PCMCIA, USB, a serial port, and audio input/output. Unfortunately, Assabet/Neponset need special additional hardware to support voltage scaling. To conduct the experiments on the target hardware, we report here frequency scaling only.

Fig. 24 shows the power consumption at different frequencies. We kept the processor busy by running an infinite loop. When the processor is busy, the system consumes 1.89 W at 206 MHz and 1.17 W at 59 MHz; the difference is 0.72 W or a 38% reduction of power consumption. The scalable range of power consumption is 0.72 W; this range excludes the power that is unaffected by frequency scaling. When the processor is idle, it consumes 1.22 W at 206 MHz and 0.97 W at 59 MHz; the power reduction is 0.25 W. There is a baseline power that cannot be reduced by frequency scaling, such as the power for the LCD display. Fig. 25 compares the performance at different frequencies. The performance scales up almost linearly with frequencies.

B. Synthesized Workload

A synthesized workload is used to compare the power consumption in the following scenarios. We use the procedures presented in Section VI to find a minimum-cost walk for each scenario.

- three to six jobs. The last job has to execute once every period.
- two to five frequencies. We start with 206 and 103 MHz. Then, we add 59, 147, and 89 MHz.
- 40% to 70% processor utilization at the highest frequency. In this paper, utilization always means the utilization at the highest frequency (206 MHz in our setup).
- Each job increments a counter until the counter's value reaches a threshold. Then, the job stops and resets the counter. The threshold value is determined by the parameters listed above.

Each period is one second, and one buffer is inserted between two jobs. Fig. 26 shows the schedules for five and six jobs with three frequencies (206, 103, and 59 MHz) when the processor utilization is 60%. Since the processor is 60% busy, it cannot stay at 103 MHz or 59 MHz indefinitely; if it did, it would violate the timing constraints. The frequency in each period is written in **boldface**. Notice that 59 MHz is not used even though it is available.

Fig. 27 depicts the measured power consumption in three cases: no frequency scaling, scaling down to 59 MHz during idleness, and scaling using our method. These data were obtained with 70% processor utilization. As can be seen at the top of the figure, the system consumes less power when the processor is idle. The boundaries of periods are clearly visible. If we scale down the frequency during idleness, period boundaries are also distinguishable. The "valleys" of the power consumption are deeper, indicating lower consumption during idleness.

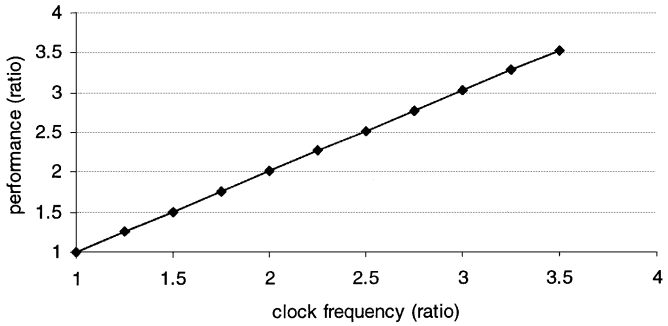


Fig. 25. Performance at different frequencies.

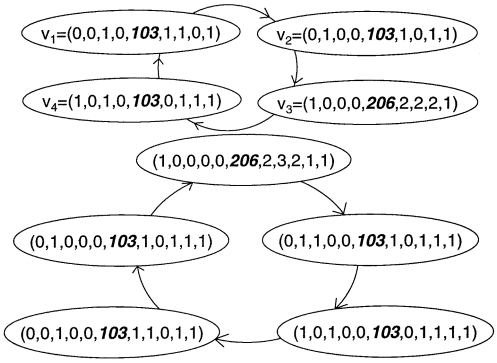


Fig. 26. Schedules for five and six jobs with 60% processor utilization.

[4]The implementation in Linux 2.4.1 recalibrates a software timer each time the clock frequency changes. This recalibration synchronizes the software timer and the hardware interrupt timer to determine their correct ratios in different clock frequencies. Linux kernel executes a while block until the ratio converges; our measurements show that it can take up to 150 ms to converge. However, such recalibration is not always necessary. A lookup table of the ratios can be calculated in advance; this reduces the scaling delay to 2 μs. The shorter delay comes from the hardware internal synchronization and cannot be further reduced by software.

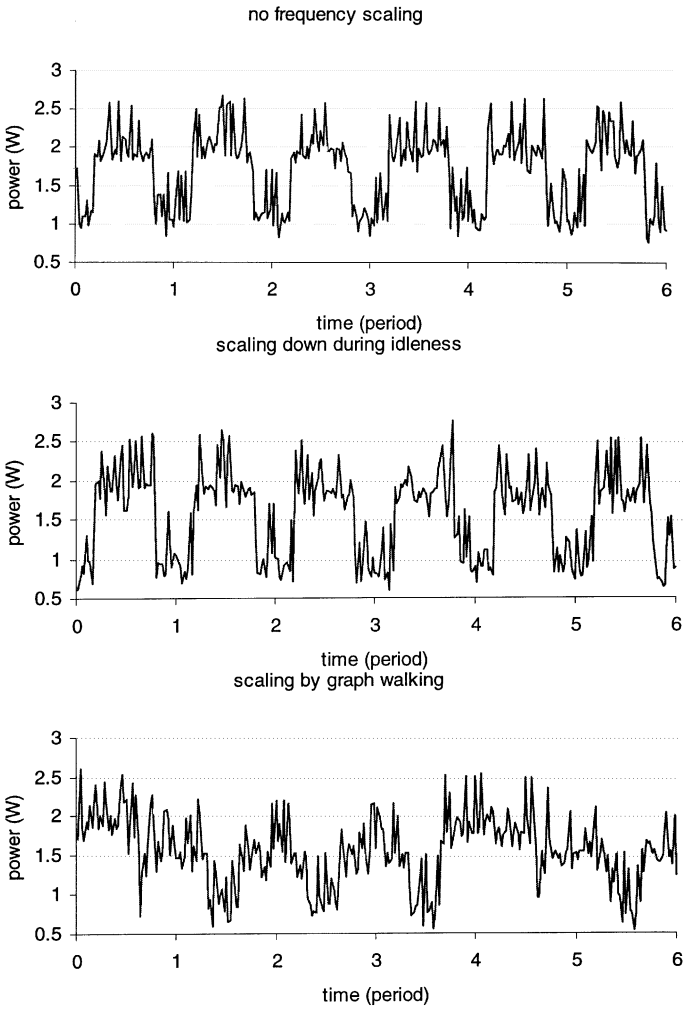


Fig. 27. No frequency scaling (top), scale down to 59 MHz during idleness (middle), and using graph-walking technique with buffer insertion (bottom). Our method saves 40% power in the scalable range.

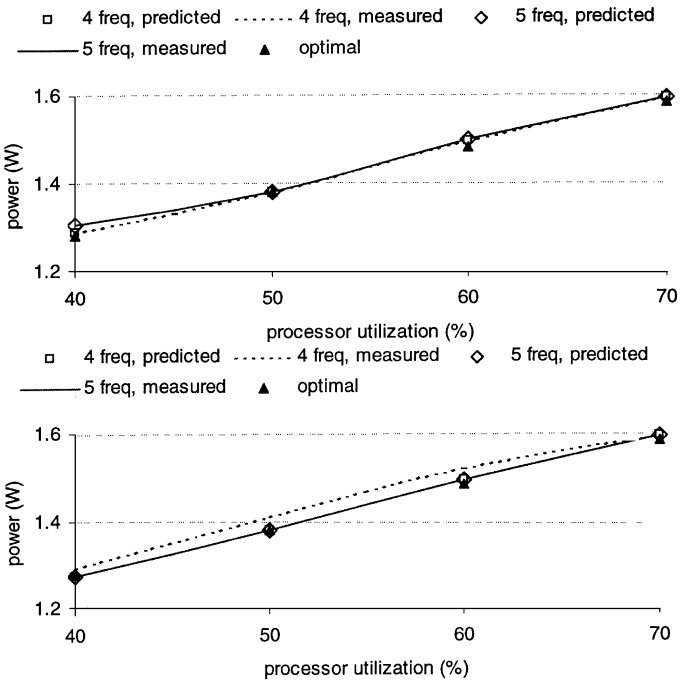


Fig. 28. Estimated, measured, and optimal power for three (top) and four jobs (bottom).

However, the power remains virtually unchanged when the processor is busy. Finally, the bottom of this figure is the power consumption of our method. The period boundaries are now blurred. This is because our method rearranges the execution order of jobs. Some jobs execute at a higher frequency whereas some other jobs execute at a lower frequency. This figure shows clearly that our method has lower average power, namely, approximately 1.6 W. This is nearly a 40% reduction in the scalable range $((1.89-1.6)/(1.89-1.17)=40\%)$.

Because Figs. 24 and 25 show almost linear scaling in power and performance, we can predict the power consumption accurately for different scenarios. The average error is 2.5% and the maximum error is 8%. Fig. 28 depicts the predicted and measured power with four and five frequencies. The horizontal axes are the processor utilization at 206 MHz, and the vertical axes are the power consumption. The squares and diamonds represent the predicted power consumption. The lines connect the measurement results. The triangles are the optimal solutions (minimum power) if the processor's frequency can be continuously scaled. Squares, diamonds, and triangles almost overlap in the figure because their values are very close. Fig. 28 shows that four frequencies (206, 147, 103, and 59 MHz) can achieve almost the minimum power consumption. The minimum power is computed as follows. Let p_{59} (1.167 W) and p_{206} (1.886 W) be the power at 59 and 206 MHz, respectively. The power is a linear interpolation obtained from the following formula:

$$p_{59}+(p_{206}-p_{59})\frac{\text{utilization}-\frac{59}{206}}{1-\frac{59}{206}}. \tag{30}$$

C. Reducing Power for Playing MPEG Video

An MPEG player differs from the previously synthesized workload in two major ways: the execution time varies from frame to frame and it has many IO operations. We characterize an MPEG player ported to Assabet. At 206 MHz, the program can display approximately 20 frames/s. We divided the program into three stages: reading data, decoding data, and displaying images. To conquer the variations in execution time, we assigned fixed duration to each stage: 1) 10 ms to read one frame, 2) 25 ms to decode one frame, and 3) 10 ms to display one frame. The allocated time durations cover all cases and can be considered as the worst case requirements. Together, the execution takes 45 ms. Our target frame rate was 15 frames/s so the processor utilization was 68% (67 ms for one frame, $45/67 = 68\%$). We compared the power consumption in several scenarios listed below. We obtained the measurements using the same setup illustrated in Fig. 23 and all values included the power of a network card.

- No frequency scaling, frame rate controlled by busy waiting. The system consumes 2.45 W.
- No frequency scaling, frame rate controlled by calling usleep. This function suspends the execution of the calling process; the units are microseconds. Using usleep reduces the power consumption to 2.30 W. Calling usleep reduces the power when a job finishes earlier than the time allocated. For

example, if reading one frame takes 9 ms, the processor is idle for 1 ms. The usleep function will cause the operating system kernel to schedule the idle process (pid = 0) which executes a low-power instruction.

- Frequency scaling with two frequencies: 206 and 103 MHz. A buffer with three slots is inserted between stages. The power consumption is 2.16 W.
- Frequency scaling with three frequencies: 206, 103, and 59 MHz. The power consumption is 2.16 W. The frequency of 59 MHz is not used because it does not help reduce the power consumption.
- Frequency scaling with four frequencies: 206, 147, 103, and 59 MHz. The processor stays at 147 MHz and the power consumption is 2.12 W. This means a 46% reduction in the scalable range $((2.45 - 2.12)/(1.89 - 1.17) = 46\%)$.

In the first case, the processor is kept busy while it waits for the beginning of the next period. The following code illustrates this busy waiting, which is implemented to control the frame rate

```
···after finish one frame
while (target finish time
        > current time)
    {update current time;}
target finish time+ = one period.;
```

In contrast, the second scenario calls usleep if there is slack time

```
···after finish one frame
slack time = target finish time
           −current time;
if (slack time > 0)
    {usleep(slack time);}
target finish time+ = one period.;
```

When a process calls usleep, this process voluntarily suspends its own execution. If no process needs the processor, the processor becomes idle and consumes less power. In our measurement, approximately 0.15 W power is saved by calling usleep. We call this *intraperiod* power saving because it saves power inside each period. In contrast, our method uses buffers across the boundaries of periods to save power, and we therefore call it an *interperiod* power-saving technique. Combining our method with the intraperiod technique saves 0.33 W, 46% in the scalable range or 14% $(0.33/2.45 = 14\%)$ of the original power. This is very close to the 0.33 W predicted by (30). Notice that after buffers are inserted, the power is reduced by 0.29 W, even when there are only two frequencies. This example shows that adding buffers is more effective than adding available frequencies to the processor.

D. Buffer Size

Fig. 28 indicates that our predicted power consumption is very close to the measured values. In the rest of this section, we use the same method to estimate the power consumption for different buffer sizes, job sizes, and arrival rates of sporadic jobs. The experimental results in Section VIII-B show that after buffers are inserted, a few frequencies are sufficient to save a significant amount of power. We use the same workload to study the effect of buffer sizes. We consider five frequencies (206, 147, 103, 89, 59 MHz), four jobs with an equal number of operations, and 45% to 75% processor utilization at 206 MHz. Fig. 29 compares the power consumption with different buffer sizes. In all cases, inserting one or two buffers between two jobs has identical effects. For 55%, 65%, and 75% utilization, adding one buffer between two jobs reduces the power because the processor can switch to low frequencies. When the utilization is 45%, adding buffers has no effect because the system consumes the minimum power if the processor stays at 103 MHz. This figure shows that whereas adding one buffer between two jobs is very effective, adding two buffers has no additional advantages. This example suggests that buffer insertion does not need a substantial amount of memory. The actual size of one buffer depends on the application programs. For an image of 240 × 160 pixels with 256 colors per pixel requires 240 × 160 bytes, or 38 KB, for one frame. It is a small portion of the memory on most systems.

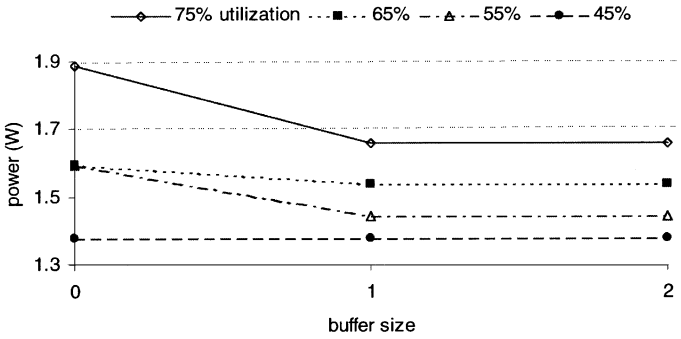


Fig. 29. Power consumption with different buffer sizes.

E. Job Size

In this section, we examine how job partition affects power consumption. The synthesized workload we discussed in Section VIII-B assumes all jobs need the same number of operations. Dividing a program into equal-size stages can be difficult: for example, the MPEG player described in Section VIII-C is naturally divided into three stages, and these stages take different amounts of time. Suppose the total number of operations of all jobs ($\sum_{l=1}^{m} w_l$) is a constant. Dividing the operations in different ways may affect power consumption. Consider the following possible ways to divide the operations. We are interested in finding the best one for power saving.

1) $w_1 = 7w_2, w_2 = w_3 = w_4$.
2) $w_1 = 4w_4, w_2 = 3w_4, w_3 = 2w_4$.
3) $4w_1 = w_4, 3w_1 = w_2, 2w_1 = w_3$.
4) $w_1 = w_2 = w_3 = w_4$
5) $w_1 = w_4 = 4w_2, w_2 = w_3$.
6) $w_2 = w_3 = 4w_1, w_1 = w_4$.

Fig. 30 depicts the relationships of these cases. The widths indicate the relative numbers of operations in each case. We consider five frequencies of the processor; one buffer is inserted between two jobs. Fig. 31 is the ratio of power consumption related to the first case. For 60% utilization (white bars), the first case consumes more power than the other cases. This is because w_1 is so large that the processor cannot execute j_1 twice in one

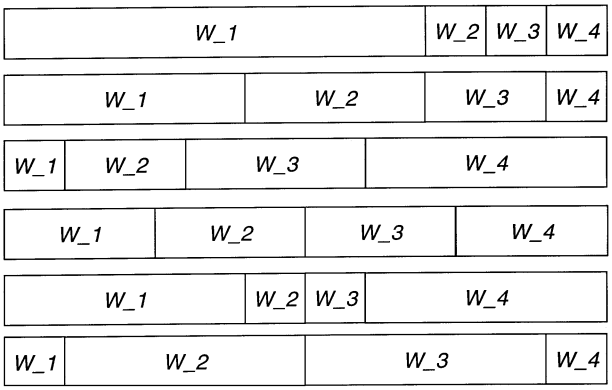


Fig. 30. Different ways to divide operations into four jobs.

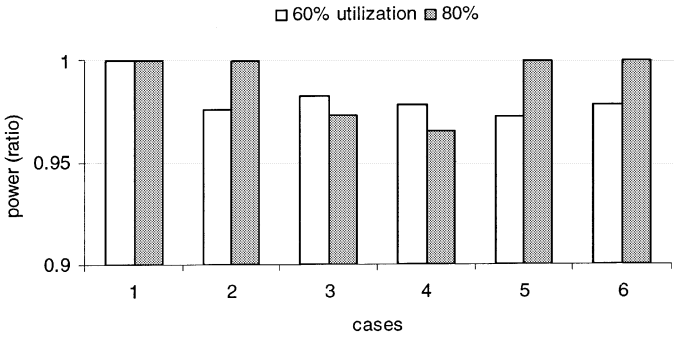


Fig. 31. Power consumption of job sizes in Fig. 30.

period to fill the first buffer. Because the first buffer is not filled, the processor cannot scale down the frequency. This case result in the most power consumption. In the other cases, the size of w_1 is smaller, so data can fill the buffers to reduce power. For 80% processor utilization, however, only case three and case four can use the buffers to save power. This example suggests the following rule: earlier jobs, such as j_1 and j_2, should need fewer operations so that the buffers can be filled. If j_1 needs too many operations, then the first buffer cannot be filled and all the other buffers are unused. If the buffers are unused, they cannot facilitate power saving.

F. Arrival Rate of Sporadic Jobs

If sporadic jobs arrive frequently, the average power consumption is higher. This section discusses how the arrival rate of sporadic jobs affects the average power. Fig. 32 is a redraw of the five-job case in the top of Fig. 26. Consider a sporadic job that takes one period at the peak frequency (206 MHz) to complete. Suppose this sporadic job arrives at the period represented by v_2. We can follow the procedure explained in Example 14 to compute the response time of the sporadic job. The result is shown in Fig. 33. After running at 206 MHz for three periods, the sporadic job completes and the processor returns to the closed walk shown in Fig. 32, starting from v_1. Even though the closed walk was "interrupted" by the sporadic job at v_2, the frequency assignment does not necessarily continue from v_3 after the job has been processed. If there is no sporadic job, the average power in the next four periods from v_2 is 1.51 W. In contrast, the processing of a sporadic job causes the average power in the same time interval to rise to 1.76 W.

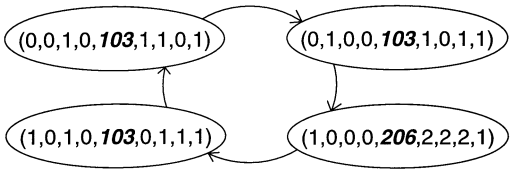


Fig. 32. Closed walk for five jobs with 60% processor utilization, a redraw of Fig. 26.

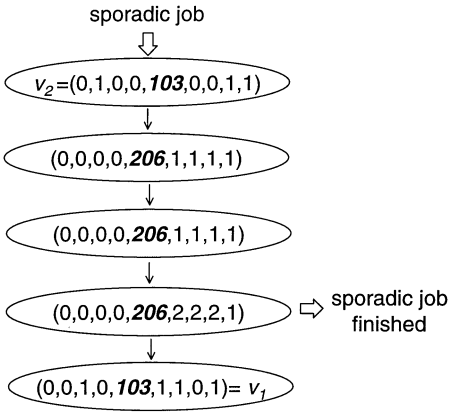


Fig. 33. Processing a sporadic job that arrives at the period represented by v_2.

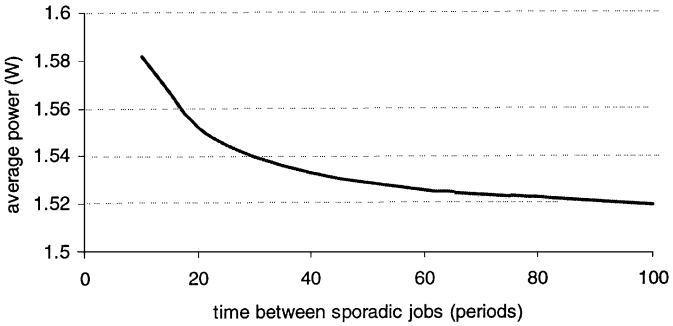


Fig. 34. Average power consumption for different intervals between two sporadic jobs.

The same method can be applied to compute the response time of the sporadic job if it arrives at the period represented by v_1, v_3, or v_4. We can also find the additional energy required to process this sporadic job. After processing the sporadic job, the frequency assignment will eventually return to the closed walk in Fig. 32, but it does not always start from v_1.

Suppose that the time interval between two sporadic jobs is long enough so that the processor can return to the original closed walk in Fig. 32. Fig. 34 shows the power consumption for different arrival rates of the sporadic job. The horizontal axis is the average number of periods between two sporadic jobs; the vertical axis is the average power consumption. This figure shows that the average power decreases rapidly as the time between two sporadic jobs increases. Since sporadic jobs are "sporadic" and the time interval between them should generally be large, their effect on power is insignificant.

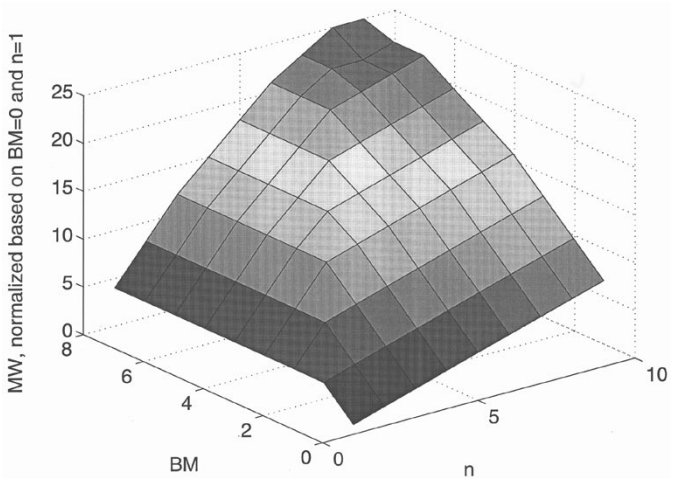


Fig. 35. Maximum number of operations of a sporadic job.

G. Timing Constraints and Maximum Operations of Sporadic Jobs

Section VII-C derives the relationship between buffer sizes and the response time of a sporadic job. We use a synthesized workload described in Section VIII-B to pictorially illustrate the conditions. There are four jobs and each takes 20% time in a period when the processor runs at the highest frequency. Without any buffers, 20% of the processor time is unused in each period. Let BM be the total memory allocated for buffers: $BM = \sum_{l=1}^{m-1} b_l$. Let $MW(n)$ be the maximum number of operations a sporadic job can complete within n periods. If BM is zero, there is no buffer, so $MW(1)$ is 20% of a period. We use 20% of a period as the normalization base. Fig. 35 shows MW for different ns and BMs. In this figure, we can observe that MW is largest when BM and n have compatible values. This can be understood by examining the conditions of (29) in Section VII-C. When n is small and BM is sufficiently large, adding more buffers will not increase MW; this corresponds to the first condition in (29) and the lower left side of the surface in Fig. 35. In contrast, when BM is too small, MW increases linearly with n because buffers become empty before the sporadic job completes. This situation corresponds to the last condition in (29) and the lower right side of the surface.

H. Summary

We set up an experimental environment using a StrongARM-based system. Our measurements indicate that power and performance scale almost linearly with the processor frequency. We used the graph-based algorithm presented in Section VI to find frequency assignments for different scenarios. Our experiments show that inserting buffers effectively reduces power consumption even for a processor with only a few frequencies. Inserting buffers can achieve nearly the minimum power with four frequencies. We also demonstrate that adding one buffer between two jobs is sufficient in many cases. We also provide a guideline for dividing operations into multiple jobs. Finally, we show that sporadic jobs have negligible effects on power increases.

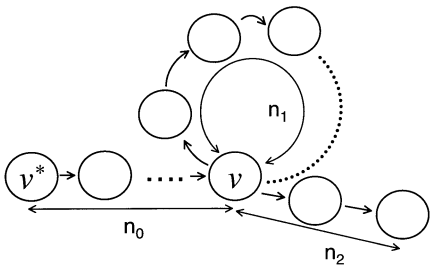


Fig. 36. A walk with a closed walk.

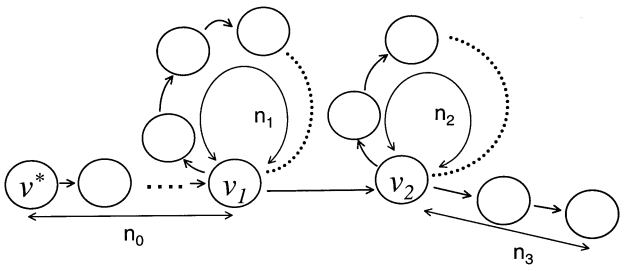


Fig. 37. A walk with multiple closed walks.

IX. Conclusion

This paper addresses power reduction by frequency scaling for mixed workloads. We explain existing methods based on mathematical programming and point out the need for efficient solutions. Our method inserts buffers between jobs and builds an assignment graph: each vertex encodes the current states of the buffers and the frequency of the processor. We develop a graph-based method that has complexity $O(|\mathcal{V}|^3)$ where $|\mathcal{V}|$ is the number of vertices of the state-space graph. We use a frequency-scalable system to demonstrate the effectiveness of our method. By inserting buffers, we can achieve nearly optimal power saving using only four frequencies. Furthermore, this method is able to dramatically reduce the response time of sporadic jobs. This method saves approximately 46% of the power required to run an MPEG player, after the nonscaleable base power is excluded. We also analyze the effects of buffer sizes and how to divide programs into multiple jobs.

Appendix

Section VI explained how to find a minimum-cost walk for a workload that has an infinite time horizon. Because an MPEG movie usually has thousands of frames, it is valid to approximate the movie as infinite frames. When a workload is infinite, we can ignore the initial cost in the walk that determines the frequency assignment. This initial cost is the cost of the walk from a starting vertex to a minimum-cost closed walk. In reality, no workload can have an infinite time horizon. For a finite-length workload the initial cost cannot be ignored. Also, there may be a "tail" that does not form a complete closed walk. Recall that $G = (\mathcal{V}, \mathcal{E})$ is an assignment graph for frequency scaling. When the number of vertices in a walk exceeds the number of vertices in G, the walk must contain at least one closed walk, according to the pigeonhole principle. Fig. 36 illustrates such a long walk.

Even though a long walk $\mathcal{W}$ must contain one closed walk, the pigeonhole does not explain whether $\mathcal{W}$ may contain multiple

```
LongFiniteWalk(input graph: G = (V, E), integer: n)
/*   mincost: minimum cost of a walk visiting n vertices */
begin
    mcost := ∞;
    /* if n is small, find a minimum-cost walk directly */
    if (n < |V| + 1)
        MinimumCostWalk(G, n);
        for each v* /* starting vertex */
            if mincost > wcost(v*, n)
                mincost := wcost(v*, n);
        return mincost;

    /* n is large */
    MinimumCostWalk(G, |V| + 1);
    MinimumCostWalk2V(G);
    FindClosedWalk(G);
    for each v*
        for each v ∈ V
            if (Cwalk(v) not empty)
                lr := ⌊(n − nW(v*, v)) / (nwalk(v) − 1)⌋;
                n3 := (n − nW(v*, v)) mod (nwalk(v) − 1);
                newcost := mcost(v*, v) + lr × (cwcost(v) − c(v)) + wcost(v, n3 − 1);
                if (mincost > newcost)
                  mincost := newcost;
    return mincost;
end
```

Fig. 38. Find minimum-cost walks.

nonoverlapping closed walks. In fact, it is possible that a minimum-cost long walk contains multiple closed walks. However, we can always find another walk whose cost is also minimum but has a special format. This special format divides $\mathcal{W}$ into three parts as shown in Fig. 36. The first part starts at one starting vertex v^* and ends at the beginning of the first closed walk; the second part repeats this closed walk; the third part leaves this closed walk and finishes $\mathcal{W}$. We can prove that the length of the third part is always less than the length of the closed walk.

Theorem 2: If a finite walk contains a closed walk constructed by FindClosedWalk, there is a walk of equal or smaller cost such that the subwalk after leaving the closed walk is shorter than the length of the closed walk. Suppose the walk in Fig. 36 is a minimum-cost walk of a finite length and it contains a closed walk of v with length n_1. There is a minimum-cost walk such that the subwalk after leaving the closed walk is shorter, or $n_2 < n_1$.

Proof: We prove the theorem by contradiction. If $n_2 > n_1$, there is a walk from v of length n_2 with a lower average cost than the closed walk, or $(\mathcal{W}_{n_2}(v))/n_2 < (\text{cwcost}(v))/n_1$. Otherwise, this minimum-cost walk should repeat the closed walk more times until n_2 is less than n_1. However, this is impossible because FindClosedWalk finds only closed walks that have the minimum average cost among all walks from v. Since the original walk contains a closed walk of v, the walk must have a lower average cost than $\mathcal{W}_{n_2}(v)$ (or equal average cost). If $n_2 > n_1$, the original walk is not a minimum cost walk and this violates the premise. Hence, n_2 cannot be larger than n_1. ♢

Notice that this theorem does not claim that all minimum-cost walks have such a format; instead, it guarantees that among all same-length walks of the minimum cost, there is one walk with this format. It is possible that a walk may have a different format and have the same cost.

Corollary 3: If a minimum-cost walk has multiple closed walks, there is a walk of the same cost such that the length of the subwalk after it leaves the first closed walk is shorter than the length of the first closed walk.

For example, a minimum-cost walk has two closed walks as shown in Fig. 37. Let n_1 and n_2 be the lengths of the first and second closed walks. If this minimum-cost walk repeats the second walk l_2 times, then $l_2 \times (n_2 - 1) + n_3 - 1 < n_1$. Here, we need to subtract one from n_2 because the length of a closed walk counts the starting and ending vertices (v_2) twice. When this closed walk repeats multiple times, the same vertex should be counted only once.

The above theorem states that the "tail" after a subwalk leaves the closed walk is shorter than the length of the closed walk. We can find a minimum-cost walk by dividing it into three subwalks: before entering a closed walk, the closed walk, and after leaving the closed walk, as illustrated in Fig. 36. The lengths of the first and the third subwalks (n_0 and n_2) must be less than $|\mathcal{V}|$ by the pigeonhole principle.

In order to find a minimum-cost, finite-length walk, our algorithm first checks whether n is small. For a small n, a minimum-cost walk does not necessarily contain a closed walk. Such a walk can be found directly by MinimumCostWalk. For a

larger n, the algorithm finds a closed walk that has a minimum cost. Since the first and the third subwalks are shorter than $|\mathcal{V}|$, they can be found by MinimumCostWalk. Fig. 38 shows the algorithm; it compares which closed walks produce the minimum cost. For each vertex that has a closed walk, the algorithm finds a minimum-cost walk from a starting vertex. The length of this walk is $n\mathcal{W}(v^*, v)$; this is n_0 in Fig. 36. Then, it computes lr; this is the number of times the closed walk repeats: $lr = \lfloor (n-n_0)/(n_1-1) \rfloor$. Finally, it computes the length of the walk after leaving the closed walk: $n_2 = (n-n_0) \bmod (n_1-1)$. The cost of this walk is

$$\mathrm{mcost}(v^*, v) + lr \times (\mathrm{cwcost}(v) - c(v)) + \mathrm{wcost}(v, n_3 - 1). \tag{31}$$

The complexity of this algorithm is $O(|\mathcal{V}|^3)$. When n is small, LongFiniteWalk takes $O(|\mathcal{V}|^2 n)$, the same as MinimumCostWalk. When n is larger than $|\mathcal{V}|$, LongFiniteWalk calls MinimumCostWalk, MinimumCostWalk2V, and FindClosedWalk. Their complexity is $O(|\mathcal{V}|^3)$. Then, LongFiniteWalk considers every closed walk reachable from a starting vertex. This takes $O(|\mathcal{V}|^2)$ iterations. Consequently, LongFiniteWalk takes $O(|\mathcal{V}|^3)$. This is independent of n. The minimum-cost walk can be constructed by applying *repeated squaring* of the closed walk [11]; this takes $O(\log lr)$ iterations.

REFERENCES

[1] A. Acquaviva, L. Benini, and B. Riccó, "An adaptive algorithm for low-power streaming multimedia processing," in *Design Automation Test Europe*, Mar. 2001, pp. 273–279.

[2] ARM Linux.. [Online]. Available: http://www.arm.linux.org.uk.

[3] R. Balakrishnan and K. Ranganathan, *A Textbook of Graph Theory*. New York: Springer, 2000.

[4] L. Benini, A. Bogliolo, and G. D. Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. VLSI Syst.*, vol. 8, pp. 299–316, June 2000.

[5] J. R. Birge and F. Louveaux, *Introduction to Stochastic Programming*. New York: Springer, 1997.

[6] J. J. Brown, D. Z. Chen, G. W. Greenwood, X. Hu, and R. W. Taylor, "Scheduling for power reduction in a real-time system," in *Int. Symp. Low Power Electronics and Design*, Monterey, CA, Aug. 1997, pp. 84–87.

[7] T. D. Burd and R. W. Brodersen, "Design issues for dynamic voltage scaling," in *Int. Symp. Low Power Electronics and Design*, July 2000, pp. 9–14.

[8] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. New York: Kluwer, 1997.

[9] L. H. Chandrasena and M. J. Liebelt, "A rate selection algorithm for quantized undithered dynamic supply voltage scaling," in *Int. Symp. Low Power Electronics and Design*, July 2000, pp. 213–215.

[10] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir, "Random walks on weighted graphs and applications to on-line algorithms," *J. Assoc. Computing Machinery*, vol. 40, no. 3, pp. 421–453, July 1993.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.

[12] R. Diestel, *Graph Theory*. New York: Springer, 1997.

[13] A. Forestier and M. R. Stan, "Limits to voltage scaling from the low power perspectiv," in *Symp. Integrated Circuits Systems Design*, Sept. 2000, pp. 365–370.

[14] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithms for dynamic speed-setting of a low-power CPU," in *ACM Int. Conf. Mobile Computing Networking*, Nov. 1995, pp. 13–25.

[15] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*. New York: Addison-Wesley, 1989.

[16] R. P. Grimaldi, *Discrete and Combinatorial Mathematics*, 2 ed. New York: Addison-Wesley, 1989.

[17] D. Grunwald, P. Levis, K. I. Farkas, C. B. M. III, and M. Neufeld, "Policies for dynamic clock scheduling," in *Symp. Operating System Design Implementation*, Oct. 2000, pp. 73–86.

[18] V. Gutnik and A. P. Chandrakasan, "Embedded power supply for low-power DSP," *IEEE Trans. VLSI Syst.*, vol. 5, pp. 425–435, Dec. 1997.

[19] I. Hong, M. Potkonjak, and M. B. Srivastava, "On-line scheduling of hard real-time tasks on variable voltage processor," in *Int. Conf. Computer-Aided Design*, Nov. 1998, pp. 653–656.

[20] C. Im, H. Kim, and S. Ha, "Dynamic voltage scheduling techniques for low power multimedia applications using buffers," in *Int. Symp. Low Power Electronics and Design*, Aug. 2001, pp. 34–39.

[21] Intel. StrongARM development kit. [Online]. Available: developer.intel.com/design/strong.

[22] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Int. Symp. Low Power Electronics Design*, Aug. 1998, pp. 197–202.

[23] P. Kall and S. W. Wallace, *Stochastic Programming*. New York: Wiley, 1997.

[24] A. I. Kibzun and Y. S. Kan, *Stochastic Programming Problems*. New York: Wiley, 1996.

[25] C. Krishna and Y.-H. Lee, "Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems," in *Real-Time Technology Applications Symp.*, May 2000, pp. 156–165.

[26] Y.-R. Lin, C.-T. Hwang, and A. C. Wu, "Scheduling techniques for variable voltage low power designs," *ACM Trans. Design Automation Electron. Syst.*, vol. 2, no. 2, pp. 81–97, Apr. 1997.

[27] J. Luo and N. K. Jha, "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems," in *Int. Conf. Computer-Aided Design*, Nov. 2000, pp. 357–364.

[28] A. Manzak and C. Chakrabarti, "Variable voltage task scheduling for minimizing energy or minimizing power," in *Int. Conf. Acoustics, Speech, Signal Processing*, June 2000, pp. 3239–3242.

[29] T. L. Martin and D. P. Siewiorek, "The impact of battery capacity and memory bandwidth on CPU speed-setting: A case study," in *Int. Symp. Low Power Electronics Design*, Aug. 1999, pp. 200–205.

[30] T. Okuma, T. Ishihara, and H. Yasuura, "Real-time task scheduling for a variable voltage processor," in *Int. Symp. System Synthesis*, Nov. 1999, pp. 24–29.

[31] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Int. Symp. Low Power Electronics and Design*, Aug. 1998, pp. 76–81.

[32] J. Pouwelse, K. Langendoen, and H. Sips, "Energy priority scheduling for variable voltage processors," in *Int. Symp. Low Power Electronics and Design*, Aug. 2001, pp. 28–33.

[33] J. M. Rabaey and M. Pedram, Eds., *Low Power Design Methodologies*. New York: Kluwer, 1996.

[34] S. Ross, *Introduction to Stochastic Dynamic Programming*. New York: Academic, 1983.

[35] Y. Shin, K. Choi, and T. Sakurai, "Power optimization of real-time embedded systems on variable speed processors," in *Int. Conf. Computer-Aided Design*, Nov. 2000, pp. 365–368.

[36] T. Šimunić, L. Benini, A. Acquaviva, P. Glynn, and G. D. Michel, "Dynamic voltage scaling for portable systems," in *Design Automation Conf.*, June 2001, pp. 524–529.

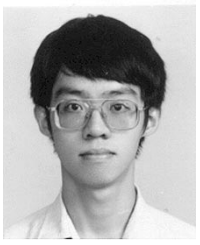
[37] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Symp. Operating Systems Design Implementation*, Monterey, CA, Nov. 1994, pp. 13–23.

[38] N. H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. New York: Addison Wesley, 1993.

[39] L. A. Wolsey, *Integer Programming*. New York: Wiley, 1998.

[40] XScale.. [Online]. Available: http://www.intel.com/design/intelxscale/.

[41] A. Zemanian, "Wandering through infinity," in *IEEE Int. Symp. Circuits Systems*, May 1992, pp. 1749–1750.



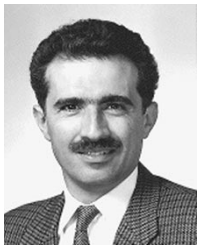
Yung-Hsiang Lu received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 2002.

He is an Assistant Professor in the School of Electrical and Computer Engineering at Purdue University, W. Lafayette, IN. His research interests include computer system design, embedded system design, and energy-efficient high-performance computing.

Luca Benini received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1997.

He is an Associate Professor in the Department of Electronics and Computer Science at the University of Bologna, Italy. He also holds visiting researcher positions at Stanford University and the Hewlett-Packard Laboratories, Palo Alto, CA. His research interests include all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications, and in the design of portable systems. On these topics he has published more than 120 papers in international journals and conferences, a book, and several book chapters.

Dr. Benini is a member of the organizing committee of the International Symposium on Low Power Design. He is a member of the technical program committee for several technical conferences, including the Design and Test in Europe Conference, International Symposium on Low Power Design, and the Symposium on Hardware–Software Codesign.



Giovanni De Micheli (S'79–M'79–SM'80–F'94) received the nuclear engineer degree from Politecnico di Milano, in 1979, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is a Professor of Electrical Engineering, and by courtesy, of Computer Science at Stanford University, Stanford, CA. Previously, he held positions at the IBM T.J. Watson Research Center, Yorktown Heights, NY, at the Department of Electronics of the Politecnico di Milano, Italy, and at Harris Semiconductor, Melbourne, FL. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software co-design and low-power design. He is the author of *Synthesis and Optimization of Digital Circuits*, (New York: McGraw-Hill, 1994) and co-author and/or co-editor of five other books and of over 250 technical articles. He is a member of the technical advisory board of several EDA companies, including Magma Design Automation, Coware, and Aplus Design Technologies. He was a member of the technical advisory board of Ambit Design Systems.

Dr. De Micheli is a Fellow of ACM. He received the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and 1993. He is President Elect of the IEEE CAS Society in 2002 and he was its Vice President (for publications) in 1999 through 2000. He was Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS from 1987 to 2001. He was the Program Chair and General Chair of the Design Automation Conference (DAC) from 1996 to 1997 and 2000, respectively. He was the Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively. He was also codirector of the NATO Advanced Study Institutes on Hardware/Software Co-design, held in Tremezzo, Italy, 1995, and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, 1986. He was a founding member of the ALaRI institute at Universita' della Svizzera Italiana (USI), in Lugano, Switzerland, where he is currently scientific counselor.