Intraprogram Dynamic Voltage Scaling: Bounding Opportunities with Analytic Modeling

FEN XIE, MARGARET MARTONOSI, and SHARAD MALIK
Princeton University

Dynamic voltage scaling (DVS) has become an important dynamic power-management technique to save energy. DVS tunes the power-performance tradeoff to the needs of the application. The goal is to minimize energy consumption while meeting performance needs. Since CPU power consumption is strongly dependent on the supply voltage, DVS exploits the ability to control the power consumption by varying a processor's supply voltage and clock frequency. However, because of the energy and time overhead associated with switching DVS modes, DVS control has been used mainly at the interprogram level.

In this paper, we explore the opportunities and limits of intraprogram DVS scheduling. An analytical model is derived to predict the maximum energy savings that can be obtained using intraprogram DVS given a few known program and processor parameters. This model gives insights into scenarios where energy consumption benefits from intraprogram DVS and those where there is no benefit. The model helps us extrapolate the benefits of intraprogram DVS into the future as processor parameters change. We then examine how much of these predicted benefits can actually be achieved through compile-time optimal settings of DVS modes. We extend an existing mixed-integer linear program formulation for this scheduling problem by accurately accounting for DVS energy switching overhead, by providing finer-grained control on settings and by considering multiple data categories in the optimization. Overall, this research provides a comprehensive view of intraprogram compile-time DVS management, providing both practical techniques for its immediate deployment as well theoretical bounds for use into the future.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.4.7 [**Operating System**]: Organization and Design—*Real-time systems and embedded systems*; I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model validation and analysis*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Analytical model, compiler, dynamic voltage scaling, low power, mixed-integer linear programming

1. INTRODUCTION

The International Technology Roadmap for Semiconductors highlights system power consumption as a limiting factor in our ability to develop designs below

Authors' address: Fen Xie, Margaret Martonosi, and Sharad Malik, Equad C305, Department of Electrical Engineering, Princeton University, Princeton, NJ 08544; email: fxie@princeton.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2004 ACM 1544-3566/04/0900-0001 $5.00

the 50-nm technology point [Semiconductor Industry Association 2001]. Indeed, power/energy consumption has already started to dominate execution time as the critical metric in system design. This holds not just for mobile systems due to battery life considerations, but also for server and desktop systems due to exorbitant cooling, packaging, and power costs.

Various power-management techniques have been implemented to save energy. Dynamic voltage and frequency scaling (DVS) is a recent dynamic power-management technique that allows the system to explicitly tradeoff performance for energy savings, by providing a range of voltage and frequency operating points. Since power consumption is strongly dependent on the supply voltage, DVS makes it possible to reduce power/energy consumption by reducing the supply voltage at run-time. However, in CMOS technology the circuit delay increases as the supply voltage decreases [Sakurai and Newton 1990]: $delay \propto V_{dd}/(V_{dd} - V_t)^{\alpha}$, where V_{dd} is the supply voltage, V_t is the threshold voltage, and α is a technology-dependent factor (between 1 and 2). Thus, reducing voltage will result in performance degradation as the clock frequency f needs to be decreased to account for the increased circuit delay. Consequently, DVS techniques need to make sure that the performance needs continue to be met as voltage and frequency are scaled.

Proposals have been made for purely-hardware DVS [Marculescu 2000] as well as for schemes that allow DVS with software control [Flautner et al. 2001; Intel Corp. 2003b; Hughes et al. 2001]. DVS accomplishes energy reduction through scheduling different parts of the computation to different (V, f) pairs so as to minimize energy while still meeting execution time deadlines. Over the past few years DVS has been shown to be a powerful technique that can potentially reduce overall energy consumption by several factors. More recently, DVS control has been exposed at the software level through instructions that can set particular values of (V, f). These mode-set instructions are provided in several contemporary microprocessors, such as Intel XScale [Intel Corp. 2003b], StrongARM [Intel Corp. 2003a], and AMD mobile K6 Plus [Advanced Micro Devices Corporation 2002]. However, the use of these instructions has been largely at the process/task level under operating system control. The coarser grain control at this level allows for easy amortization of the energy and run-time overhead incurred in switching between modes for both the power supply (V) as well as the clock circuitry (f). It also makes the control algorithm easier because it is relatively easy to assign priorities to tasks, and schedule higher priority tasks at higher voltages and lower priority tasks at lower voltages. Providing this control at a finer-grain level within a program would require careful analysis to determine when the mode-switch advantages outweigh the overhead. Hsu and Kremer provide a heuristic technique that lowers the voltage for memory-bound sections [Hsu and Kremer 2002]. The intuition is that the execution time here is bound by memory access time, and thus the compute time can be slowed down with little impact on the total execution time, but with potentially significant savings in energy consumption. Using this technique, they have been able to demonstrate modest energy savings. Subsequent work on using mathematical optimization by Saputra et al. [2002] provides an exact mixed-integer linear programming (MILP) technique that can determine

the appropriate (V, f) setting for each loop nest. This optimization seems to result in better energy savings. However, this formulation does not account for any energy penalties incurred by mode switching. Thus, it is unclear how much of these savings will hold up once those are accounted for.

In this paper, we are interested in studying the opportunities and limits of intraprogram DVS using compile-time mode settings. We seek to answer the following questions: Under what scenarios can we get significant energy savings? What are the limits to these savings? The answers to these questions determine when and where (if ever) is compile-time intraprogram DVS worth pursuing. We answer these questions by building a detailed analytical model for energy savings for a program and examining its upper limits. In the process, we determine the factors that determine energy savings and their relative contributions. These factors include some simple program-dependent parameters, memory speed, and the number of available voltage levels.

We also examine how these opportunities can be exploited in practice. Toward this end we develop an MILP-optimization formulation that extends the formulation by Saputra et al. by including energy penalties for mode switches, providing a much finer grain of program control, and enabling the use of multiple input data categories to determine optimal settings. We show how the solution times for this optimization can be made-acceptable in practice through a judicious restriction of the mode-setting variables. Finally, we show how the results of this optimization relate to the limits predicted by our analytical model.

The rest of this paper is organized as follows. Section 2 reviews related work in this area. This is followed by a description of our basic analytical model and analysis in Section 3. The detailed analysis is presented in Sections 4 and 5. Section 6 derives the MILP formulation used to determine the values of the optimal mode-setting instructions. Section 7 discusses some implementation details for our MILP-based approach, and Section 8 provides the results of various experiments. Section 9 considers the applicability of intraprogram DVS on a real-time multitask environment. Critical and unresolved issues are the focus of Section 10. Finally, we present some concluding remarks in Section 11.

2. RELATED WORK

DVS scheduling policies have been studied exhaustively at the operating system (OS), microarchitecture, and compiler levels. Algorithms at the OS level using heuristic scheduling include interval-based algorithms and task-based algorithms. Interval-based algorithms divide the execution time into fixed-length intervals. Information from previous intervals is used to predict the workload of the current interval and an appropriate DVS setting is picked based on the prediction. Algorithms in the category include Lorch and Smith [2001], Weiser et al. [1994], and Sinha and Chandrakasan [2001]. The task-based algorithms [Luo and Jha 2003; Jejurikar and Gupta 2002; Zhang et al. 2003; Swaminathan et al. 2002; Pillai and Shin 2001] schedule at the granularity of a task and use heuristics to pick a DVS setting. Integer linear programming (ILP)-based scheduling has also been used in algorithms at the OS level. For

example, Ishihara and Yasuura [1998] give an ILP formulation that does not take into account the transition costs. Swaminathan and Chakrabarty [2001] incorporate the transition costs into the ILP formulation but make some simplifications and approximations in order to make the formulation linear. At the microarchitecture level, Ghiasi et al. [2000] suggest the use of IPC (instructions per cycle) to direct DVS, and Marculescu [2000] proposes the use of cache misses to direct DVS. Both are done through hardware support at run-time and work for general applications. In Im et al. [2001], buffers are used to facilitate the utilization of DVS techniques for low-power multimedia applications. The multiple clock domain microarchitecture proposed by Magklis et al. [2003] allows the frequency/voltage of microprocessor regions to be adjusted independently and dynamically. The utilization of interregion communication queues is used to determine the appropriate DVS settings for each domain. The DVS techniques for similar multiple clock domain architectures have also been proposed in Iyer and Marculescu [2002].

Some research efforts have targeted the use of mode-set instructions. Mode-set instructions are inserted either evenly at regular intervals in the program [Lee and Sakurai 2000], or on a limited number of control-flow edges as proposed by Shin [Shin et al. 2001]. In Shin's work [Shin et al. 2001], the mode value is set statically using the worst-case execution time analysis for each basic block. Branching edges in control-flow graph (CFG) that drop the remaining worst-case execution time faster than the current execution rates are selected as voltage/frequency scaling points. In AbouGhazaleh et al. [2003], a power-management point routine is invoked periodically to adjust the processor voltage/frequency based on the worst-case remaining cycles, which are collected at run-time by instrumenting code at compile time. Thus, there exists run-time overhead related to the mode decision in addition to the switching overhead. Hsu and Kremer [2002] suggest lowering voltage/frequency in memory-bound regions using power-down instructions and provide a nonlinear optimization formulation for optimal scheduling. Saputra et al. [2002] derive an ILP formulation for determining optimal modes for each loop nest, but do not consider the energy overhead of switching modes.

The efficiency of scheduling policies has also been discussed in the literature. Hsu and Kremer [2003] have introduced a simple model to estimate theoretical bounds of energy reduction, which any DVS algorithm can produce. In Ishihara and Yasuura [1998], a simple ideal model that is solely based on the dynamic power dissipation of CMOS circuits has been studied and an OS-level scheduling policy is discussed based on that model and ILP. Some other work focuses only on the limits of energy savings for DVS systems without taking into consideration actual policies. Qu provides models for feasible DVS systems in his work [Qu 2001]. However, evaluating the potential energy savings of compile-time DVS policies for real programs has not received much attention thus far. We feel it is important as it gives us deep insight into opportunities and limits of compile-time intraprogram DVS. Since the interprogram DVS scheduling at the OS level is fully developed, we assume benefits from interprogram DVS scheduling have been exploited and focus our efforts on extra energy savings provided by intraprogram DVS.

In our previous work [Xie et al. 2003], we touched on this aspect by providing an analytical model. This work extends on that by incorporating features of both real-program behavior and intraprogram DVS scheduling. In this paper, we present an accurate and simplified analytical model, which is a refinement of our previous analytical model [Xie et al. 2003]. We also extend existing ILP formulations to apply DVS to any granularity of program code with practical transition costs and multiple data categories.

3. ANALYTICAL MODELING FOR ENERGY SAVINGS USING INTRAPROGRAM DVS

3.1 Overview

We are interested in answering the following questions: What are the factors that influence the amount of power savings obtained by using compile-time DVS? Can we determine the power savings and upper bounds for them in terms of these factors? The answers to these questions will help provide insight into what kinds of programs are likely to benefit from compile-time DVS, under what scenarios and by how much. Among other outcomes, accurate analysis can help lay to rest the debate on the value of intraprogram DVS scheduling.

There has been some research on potential energy savings for DVS scheduling. Analytical models have been studied in Ishihara and Yasuura [1998] and Qu [2001]. However, that research only models computation operations and not memory operations, thus not capturing a critical aspect of program execution. Further, their models are not suitable for bound analysis for compile-time DVS because they ignore critical aspects of the compile-time restriction. In this section, we describe a more realistic and accurate analytical model to determine achievable energy savings that overcomes the restrictions of prior modeling efforts.

In deriving this model, we make the following assumptions about the program, microarchitecture, and circuit implementation:

(1) The program's logical behavior does not change with frequency.
(2) Memory is asynchronous with the CPU.
(3) The clock is gated when the processor is idle.
(4) The relationship between frequency and voltage is: $f = k(v - v_t)^\alpha / v$ where v_t is the threshold voltage, and α is a technology-dependent factor (currently around 1.5) [Sakurai and Newton 1990].
(5) Computation can be assigned to different frequencies at an arbitrarily fine grain, that is, a continuous partitioning of the computation and its assignment to different voltages is possible. Also each invocation of an instruction can run at different frequency.
(6) There is no energy or delay penalty associated with switching between different (V, f) pairs.

While the first four assumptions are very realistic, the last two are optimistic in the sense that they allow for higher energy savings than may be

achievable in practice. As we are interested in determining upper bounds on the achievable savings, this is acceptable. When we move from the analytical models to practical implementation, we do not use the last two assumptions. In the implementation, the granularity of computation is determined by the potential positions of voltage and frequency switches. In addition, energy and delay penalty for switches are also included. Thus, the optimism will result in higher energy savings in theory as compared to practice. We will review the tightness of the bounds in Section 8.

3.2 Basic Definitions of Model Parameters

The existence of memory operations complicates the analysis. Cache misses provide an opportunity for intraprogram DVS, since memory access time and thus possible execution time will not change with voltage/frequency if the memory is asynchronous with the processor. Slowing down the frequency in that region may save energy without impacting performance.

Instruction cache misses, on the other hand, do not provide a good opportunity for intraprogram DVS. First, most instruction cache misses are cold-start misses. Second, when instruction cache misses happen, the CPU usually stalls and clock gating will reduce the energy significantly. This is not the case with data cache misses since computation may be able to continue while waiting for data from the memory.

For each instruction execution, we classify it as either an overlap operation or a nonoverlap operation. Some operations can run concurrently with cache miss memory operations. They are referred to as the overlap part. Some operations cannot run in parallel with cache miss memory operations and are dependent on the data being fetched from the memory. They are referred to as the nonoverlap part.

For a piece of code consisting of three instructions,

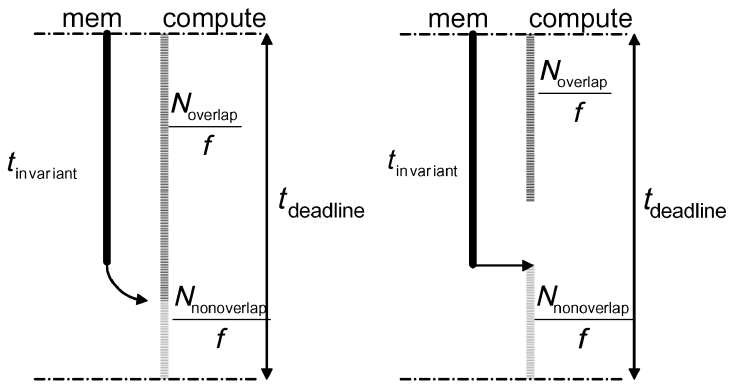
(1) LOAD r1, s1(4)
(2) ADD r2, r3, r5
(3) SUB r4, r1, r4

suppose the first LOAD instruction encounters a cache miss and goes to memory to fetch data. The ADD instruction can run concurrently with the LOAD instruction, so it is an overlapping operation. Since the SUB instruction needs to use data from the LOAD instruction and it cannot run in parallel with the LOAD instruction, it is a nonoverlapping operation.

We define the following program parameters of the model based on this observation.

N_{overlap} The number of execution cycles of operations that can run in parallel with cache miss memory operations at any frequency.

$N_{\text{nonoverlap}}$ The number of execution cycles of operations that cannot run in parallel with any cache miss memory operations at all possible frequencies.



(a) Computation-dominated case (b) Memory-dominated case

Fig. 1. Possible overlaps of memory and computation. For the computation-dominated case, overlapped computation operations take longer than cache misses. For the memory-dominated case, cache misses take longer than overlapped operations.

$t_{\text{invariant}}$ The execution time of cache miss memory operations. As memory operates asynchronously relative to the processor, this time is independent of processor frequency, and thus is measured absolutely rather than in terms of processor cycles.

Some execution cycles may run in parallel with cache miss memory operations at high frequencies but do not overlap with any cache miss memory operations at lower frequencies because cache miss memory operations have smaller latencies in terms of execution cycles at lower frequencies. However, according to the definition, if they can run in parallel with cache miss memory operations at high frequency, they are still classified as N_{overlap}. So N_{overlap} and $N_{\text{nonoverlap}}$ do not change with processor frequency.

For the simple example above, $N_{\text{overlap}} = 1$, $N_{\text{nonoverlap}} = 1$, and $t_{\text{invariant}} = 0.1\ \mu$s, if we assume ADD and SUB instructions take 1 cycle and the memory latency is $0.1\ \mu$s.

Thus, we can represent the execution time of this simple piece of code as illustrated in Figure 1. If the ADD instruction hypothetically takes longer than the cache miss LOAD instruction, the execution time will be the summation of ADD and SUB instruction as shown in Figure 1(a). If the ADD instruction takes time shorter than the LOAD instruction, then the execution time is illustrated in Figure 1(b), which is the summation of execution time of the LOAD and SUB instructions.

The actual program execution consists of many pieces of code, so the execution time will be represented by the accumulation of many similar figures as shown in Figure 1. In other words, the cache miss memory operations, the overlapped and nonoverlapped operations will be interleaved. However, we can lump all the occurrences of each category together and get an overall number for each category, since for the purpose of the analysis it is sufficient to represent

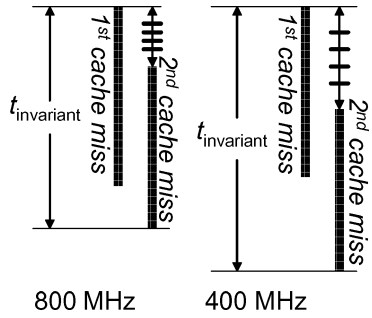


Fig. 2. Variation of $t_{\text{invariant}}$ due to different processor frequencies when multiple outstanding cache misses are allowed.

the total execution time. Consequently, the overall execution time of the whole program still can be represented as illustrated in Figure 1.

3.3 Validation of Model Parameters

The program parameter $t_{\text{invariant}}$ we defined may be dependent on the processor frequency. If the front-side bus has different frequency levels as the processor frequency changes, memory access time varies. However, we can divide the analysis into separate processor frequency domains so that the front-side bus speed is fixed in that processor frequency domain. Thus in each domain, memory access time is constant and analysis can be performed exactly the same as the fixed front-side bus speed case. Thus, we assume the fixed front-side bus speed thereafter without losing any generality.

Since memory access time is fixed, for processors allowing only one cache miss at a time, $t_{\text{invariant}}$ is constant. However, for processors that allow multiple outstanding cache misses, $t_{\text{invariant}}$ might not be constant. This is because multiple cache misses can overlap and the overlapped ratio depends on the processor frequency as shown in Figure 2. In Figure 2, there are two consecutive cache misses. The second cache miss is 5 cycles behind the first one. For 800 MHz, 5 cycles mean 6.25 ns but for 400 MHz, 5 cycles mean 12.5 ns. So $t_{\text{invariant}}$ increases as the processor frequency decreases.

In order to illustrate the trend, we use SimpleScalar to profile $t_{\text{invariant}}$ for five benchmarks from the MediaBench suite [Lee et al. 1997] (mpg123 is from MediaBench II [MediaBench II 2003]) running at a single frequency f. If for an execution cycle, a cache miss is being handled, then $t_{\text{invariant}}$ increases by $1/f$.

Figure 3 shows the profiled $t_{\text{invariant}}$ of the five benchmarks we consider using different frequencies for a processor allowing at most eight outstanding cache misses at one time. It is easy to notice that $t_{\text{invariant}}$ has variations across different frequencies. Higher frequency gives a higher overlap ratio. For programs with high miss rates such as mpg123 and epic, the variation is more obvious. For epic, 0.85 million cycles out of 2.7 million cache miss cycles have more than two simultaneous cache misses and the variation is 7% at 200 MHz. Since the variation is less than 8%, we will treat $t_{\text{invariant}}$ as constants to simplify the analysis.

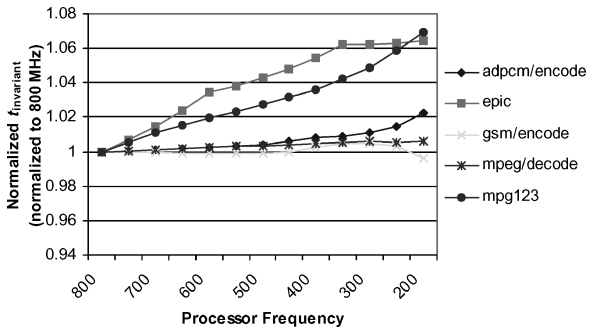


Fig. 3. $t_{\text{invariant}}$ for different processor frequencies: Eight outstanding cache misses allowed.

Note that N_{overlap} and $N_{\text{nonoverlap}}$ are independent of the processor frequency. We pick the values sampled at the highest frequency (800 MHz) as the values of the three parameters. Any cycles that do not overlap with memory accesses at the highest frequency will not overlap with any memory accesses at lower frequency, so they belong to $N_{\text{nonoverlap}}$. Each execution cycle is classified into one of the three categories. If for an execution cycle, a cache miss is being handled, then $t_{\text{invariant}}$ increases by $1/f$. If at the same time, there are some computation operations, then N_{overlap} increases by 1. For an execution cycle, if there is no cache miss and there are computation operations, then $N_{\text{nonoverlap}}$ increases by 1. Note that the classification is based on every execution cycle and not every instruction. This is because for superscalar processors multiple instructions can be executed at the same time.

Consequently, the total execution time for any piece of code or program as it just meets its deadline can be represented as illustrated in Figure 1, with the different cases corresponding to different relative values of these parameters. In these figures, the frequency f is not fixed through the program execution and may vary.

In the figures, t_{deadline} is the time deadline that the computation needs to meet. In Figure 1(a), parallel operations determine the execution time of the overlap part and total execution time is $N_{\text{overlap}}/f + N_{\text{nonoverlap}}/f$. In Figure 1(b), cache miss memory operations dominate the overlap part and total execution time is $t_{\text{invariant}} + N_{\text{nonoverlap}}/f$.

The total execution time expression for any of these cases is

$$\max(t_{\text{invariant}}, N_{\text{overlap}}/f) + N_{\text{nonoverlap}}/f$$

3.4 Basic Model

The goal of the analytical model is to find minimum energy points (i.e., maximum energy savings) using different voltages for different parts of the execution, subject to the following two time constraints:

(1) The total execution time is less than t_{deadline}.
(2) The time for the nonoverlap operations cannot overlap with the time for the cache miss memory operations $t_{\text{invariant}}$. This respects the fact that nonoverlap operations must wait for the memory operations to complete.

We now state our overall optimization problem. In Figure 1, assume a time ordering from top to bottom. Let t_1 be the time the overlapping part finishes, t_2 be the time the nonoverlap operations start execution, and t_3 be the time all computation finishes. The goal, then, is to minimize:

$$E = \int_0^{t_1} v_1^2(t) f_1(t)\, dt + \int_{t_2}^{t_3} v_2^2(t) f_2(t)\, dt$$

subject to the constraints

(1) $t_3 \leq t_{\text{deadline}}$
(2) $t_2 \geq t_{\text{invariant}}$ and $t_1 \leq t_2$

The first integral represents energy consumption during the overlapping computation. The second integral represents energy during the nonoverlap period. As mentioned earlier, we assume perfect clock gating when the processor is waiting for memory, and thus there is no energy consumption in the processor during idle memory waits. Also, we account here for only the processor energy; the memory energy is a treated as constant independent of processor frequency. Unlike models proposed in Ishihara and Yasuura [1998] or Qu [2001], our inclusion of memory operations adds significant complexity to the optimization problem. We now consider specific cases of this optimization, corresponding to different options for the set of voltages available.

4. DISCUSSION OF MODEL DETAILS: CONTINUOUSLY SCALABLE VOLTAGE CASE

We first consider the case where the supply voltage can be scaled continuously, that is, v_1, v_2 can vary continuously over some range $[V_L, V_H]$. While continuous scaling may not be available in practice, this analysis serves two purposes. First, it helps us build up to the discrete case presented in the next section. Second, it approximates the case where the number of discrete voltages available is sufficiently large. Previous work considering only computation operations and not memory operations showed that for a fixed deadline, the optimal solution is to use a single supply voltage that adjusts the total execution time to just meet the deadline [Ishihara and Yasuura 1998]. So, in this case v_1 is a constant over $[0, t_1]$ and 0 at other times. v_2 is a constant over $[t_2, t_3]$ and 0 otherwise. We now need to find appropriate v_1, v_2, $[0, t_1]$ and $[t_2, t_3]$ such that energy is minimized.

A key dividing line between the computation-dominated and memory-dominated case concerns the inflection point when memory time begins to matter. We use the term $f_{\text{invariant}}$ to refer to a clock frequency at which the execution time of N_{overlap} cycles of computation operations just balances the cache miss service time $t_{\text{invariant}}$, thus $f_{\text{invariant}} = N_{\text{overlap}}/t_{\text{invariant}}$. (We use $v_{\text{invariant}}$ to refer to the voltage setting paired with that frequency). If the speed for the overlapped part is slower than $f_{\text{invariant}}$, then the N_{overlap} cycles of computation operations take up the entire cache miss period $t_{\text{invariant}}$ and go beyond as shown in Figure 1(a). Similarly, if the speed is faster than $f_{\text{invariant}}$, then $t_{\text{invariant}}$ is not filled up with overlap operations as shown in Figure 1(b). If the optimal energy

point can be reached with a frequency slower than $f_{\text{invariant}}$ we say that the program is computation dominated.

The problem becomes one of minimizing:

$$E(v_1, v_2) = N_{\text{overlap}} \times v_1^2 + N_{\text{nonoverlap}} \times v_2^2$$

subject to different constraints depending on

if $f_1 > f_{\text{invariant}}$

$$\frac{N_{\text{nonoverlap}}}{f_2} + t_{\text{invariant}} = t_{\text{deadline}} \tag{1}$$

if $f_1 \leq f_{\text{invariant}}$

$$\frac{N_{\text{overlap}}}{f_1} + \frac{N_{\text{nonoverlap}}}{f_2} = t_{\text{deadline}} \tag{2}$$

When $f_1 > f_{\text{invariant}}$, f_2 (hence v_2) is completely determined by the constraint (1). f_1 and f_2 (thus v_1 and v_2) are independent. As a result, to get the most energy savings, f_1 should be as low as possible, which means f_1 should not be greater than $f_{\text{invariant}}$. Hence, we will only focus on the second case where $f_1 \leq f_{\text{invariant}}$ from now on.

Representing f as a function of v to get all equations in terms of v:

if $k(v_1 - v_T)^\alpha / v_1 \leq f_{\text{invariant}}$

$$\frac{N_{\text{overlap}} v_1}{k(v_1 - v_T)^\alpha} + \frac{N_{\text{nonoverlap}} v_2}{k(v_2 - v_T)^\alpha} = t_{\text{deadline}} \tag{3}$$

Due to the deadline constraint, v_1 and v_2 are not independent. We use this in deriving the value of v_1 (and thus v_2) that results in the least energy as follows:

$$\frac{dE}{dv_1} = 2N_{\text{overlap}} v_1 + 2N_{\text{nonoverlap}} v_2 \frac{dv_2}{dv_1}$$

$\frac{dv_2}{dv_1}$ is obtained from the constraint equation (3).

$$\frac{N_{\text{overlap}}(v_1 - \alpha v_1 - v_T)}{k(v_1 - v_T)^{\alpha+1}} + \frac{N_{\text{nonoverlap}}(v_2 - \alpha v_2 - v_T)}{k(v_2 - v_T)^{\alpha+1}} \frac{dv_2}{dv_1} = 0$$

and

$$\frac{dv_2}{dv_1} = -\frac{N_{\text{overlap}} v_1}{N_{\text{nonoverlap}} v_2} \frac{(1 - \alpha - v_T/v_1)}{(1 - \alpha - v_T/v_2)} \frac{(v_2 - v_T)^{\alpha+1}}{(v_1 - v_T)^{\alpha+1}}$$

Thus, we get

$$\frac{dE}{dv_1} = 2N_{\text{overlap}} v_1 \left(1 - \frac{(1 - \alpha - v_T/v_1)}{(1 - \alpha - v_T/v_2)} \frac{(v_2 - v_T)^{\alpha+1}}{(v_1 - v_T)^{\alpha+1}}\right) \tag{4}$$

The sign of $\frac{dE}{dv_1}$ depends on the relationship between v_1 and v_2.

Let $g(v_1, v_2) = \frac{(1-\alpha-v_T/v_1)}{(1-\alpha-v_T/v_2)} \frac{(v_2-v_T)^{\alpha+1}}{(v_1-v_T)^{\alpha+1}}$, then $g(v_1, v_2)$ is a decreasing function of v_1. Hence $g(v_1, v_2)$ is an increasing function of v_2, since v_1 and v_2 must satisfy constraint (2). It is easy to see that when $v_1 = v_2$, $g(v_1, v_2) = 1$.

When $v_1 = v_2$, we get $f_1 = f_2 = (N_{\text{overlap}}+N_{\text{nonoverlap}})/t_{\text{deadline}}$ from constraint (2). We define $f_{\text{ideal}} = (N_{\text{overlap}} + N_{\text{nonoverlap}})/t_{\text{deadline}}$ and v_{ideal} is the paired voltage setting with f_{ideal}. So f_{ideal} is the single frequency at which the whole program just finishes at the deadline.

It is easy to deduce that

$$g(v_1, v_2) \begin{cases} < 1 & \text{if } v_1 > v_2 \\ = 1 & \text{if } v_1 = v_2 = v_{\text{ideal}} \\ > 1 & \text{if } v_1 < v_2 \end{cases}$$

By applying the results of $g(v_1, v_2)$ to Equation (4), we get the sign of $\frac{dE}{dv_1}$:

$$\frac{dE}{dv_1} \begin{cases} > 0 & \text{if } v_1 > v_2 \\ = 0 & \text{if } v_1 = v_2 = v_{\text{ideal}} \\ < 0 & \text{if } v_1 < v_2 \end{cases} \tag{5}$$

Depending on the parameters, however, not all three relationships in Equation (5) are valid. We define $f_{\text{nonoverlap}} = N_{\text{nonoverlap}}/(t_{\text{deadline}} - t_{\text{invariant}})$ and $v_{\text{nonoverlap}}$ is the paired voltage with $f_{\text{nonoverlap}}$. $f_{\text{nonoverlap}}$ is the lowest frequency f_2 can achieve, since nonoverlap operations cannot be executed within $t_{\text{invariant}}$. So $(t_{\text{deadline}} - t_{\text{invariant}})$ is the maximum time for nonoverlap operations. From the previous discussion, $f_{\text{invariant}}$ is the upper bound for f_1. So the relationship between $f_{\text{invariant}}$ and $f_{\text{nonoverlap}}$ determines the possible relationship for v_1 and v_2. As a result, we will discuss two cases separately.

4.1 Computation-Dominated Case

Since $f_{\text{nonoverlap}}$ is the lowest frequency for f_2 and $f_{\text{invariant}}$ is the highest frequency for f_1, if $f_{\text{nonoverlap}} < f_{\text{invariant}}$, all three possible relationships (namely, $v_1 > v_2$, $v_1 = v_2$, and $v_1 < v_2$) are valid because they do not violate the boundary requirements. Thus, all conditions in Equation (5) can be satisfied. Then, from Equation (5), it is easy to see that when $v_1 = v_2 = v_{\text{ideal}}$, energy consumption is minimized, since $\frac{dE}{dv_1} = 0$.

Figure 4 shows the relationship between energy consumption and supply voltage v_1 for a set of parameters that satisfy these conditions (i.e., $f_{\text{nonoverlap}} \leq f_{\text{invariant}}$). When $v_1 < v_{\text{ideal}}$ or $v_1 > v_{\text{ideal}}$, energy consumption increases as you move away from v_{ideal}. When $v_1 = v_2 = v_{\text{ideal}}$, energy is minimized. This is the computation-dominated case as shown in Figure 5(a), since the overlapped operations dominate the execution time of the overlapped part. Using the single frequency f_{ideal}, overlap operations take up the entire cache miss period $t_{\text{invariant}}$ and go beyond. So the optimal energy point is reached with a single frequency f_{ideal}, which is slower than $f_{\text{invariant}}$. The minimum energy is

$$E = (N_{\text{overlap}} + N_{\text{nonoverlap}})v_{\text{ideal}}^2$$

In this case, the memory time is largely irrelevant because meeting the deadline mainly revolves around getting the computation done in time. This case is similar to the pure computation case in Ishihara and Yasuura [1998] and thus a *single frequency* leads to optimal energy performance. Since a single (V, f)

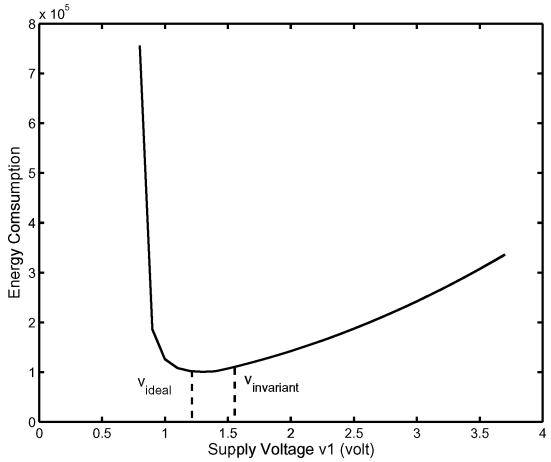
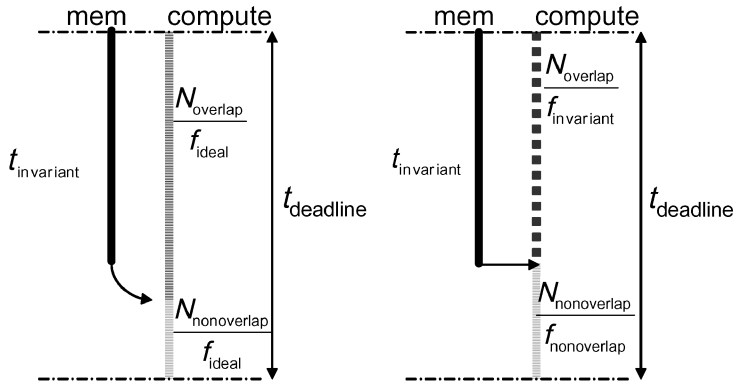


Fig. 4. Computation dominated: Energy consumption versus supply voltage (v_1) of overlapped compute/memory region ($t_{\text{deadline}} = 10$ ms, $t_{\text{invariant}} = 3$ ms, $N_{\text{overlap}} = 2 \times 10^6$ cycles, $N_{\text{nonverlap}} = 4 \times 10^6$ cycles, $\alpha = 1.5$, $v_T = 0.45$ v.)



(a) Optimized frequency for computation-dominated case

(b) Optimized frequencies for memory-dominated case

Fig. 5. Optimized frequencies to achieve minimum energy for computation-dominated case and memory-dominated case.

setting is optimal for this case, intraprogram DVS will not provide energy savings here.

4.2 Memory-Dominated Case

If $f_{\text{nonoverlap}} > f_{\text{invariant}}$, then f_2 is always greater than f_1 because $f_{\text{nonoverlap}}$ is the lower bound for f_2 and $f_{\text{invariant}}$ is the upper bound for f_1. So $v_1 > v_2$ and $v_1 = v_2$ are not valid. There is only one valid condition here: $v_1 < v_{\text{invariant}} < v_{\text{nonoverlap}} < v_2$.

$$\frac{dE}{dv_1} < 0 \quad \text{if } v_1 < v_{\text{invariant}} < v_2$$

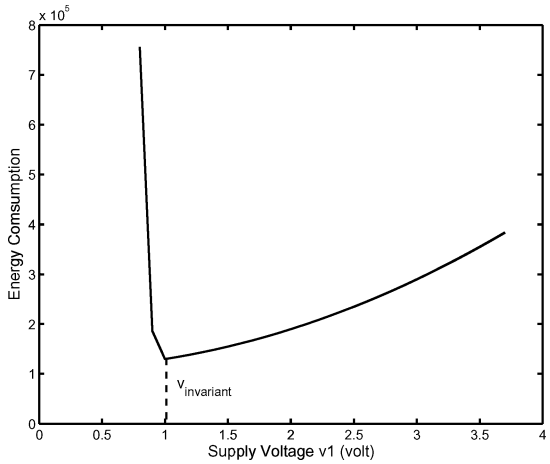


Fig. 6. Memory dominated: Energy consumption versus supply voltage (v_1) of mixed compute/memory region ($t_{\text{deadline}} = 10$ ms, $t_{\text{invariant}} = 5$ ms, $N_{\text{overlap}} = 2 \times 10^6$ cycles, $N_{\text{nonverlap}} = 4 \times 10^6$ cycles, $\alpha = 1.5$, $v_T = 0.45$ v).

Since the energy function E is a decreasing function of v_1 and $v_{\text{invariant}}$ is the upper bound for v_1, $v_{\text{invariant}}$ is the frequency for the overlapped part to minimize energy. At this point, $v_{\text{nonoverlap}}$ is the frequency for the nonoverlapped part.

Figure 6 shows the energy consumption with respect to different v_1 for this situation. When $v_1 > v_{\text{invariant}}$ or $v_1 < v_{\text{invariant}}$, energy consumption increases as you move away from $v_{\text{invariant}}$. Unlike the computation-dominated case, there is a sharp turning point at $v_{\text{invariant}}$. At this point, energy consumption is minimized, which requires two different (V, f) settings as shown in Figure 5(b). This is the memory-dominated case because $t_{\text{invariant}}$ is just filled up with overlap operations for minimum energy and overlap operations cannot go beyond.

The minimum energy is achieved when the overlapped part uses $v_{\text{invariant}}$ and the nonoverlap part uses $v_{\text{nonoverlap}}$ ($v_{\text{nonoverlap}} \geq v_{\text{overlap}}$).

$$E_{\text{min}} = N_{\text{overlap}} v_{\text{invariant}}^2 + N_{\text{nonoverlap}} v_{\text{nonoverlap}}^2$$

Note that in this case, the computation is broken into two parts, the overlapped and the nonoverlapped part, each with its own time constraint. The optimization problem has been solved by treating it as two optimization problems for the overlapped part and the nonoverlapped part, respectively. Thus, the optimal operating point arises when *two frequencies* are chosen: one for the overlapped computation, and a different one for the nonoverlapped computation.

4.3 Continuous Voltage Settings: Summary and Results

The primary result here is that a special relationship between various parameters is required to achieve energy savings using compile-time mode settings. Specifically, we need $f_{\text{nonoverlap}} > f_{\text{invariant}}$. This condition translates to: $N_{\text{nonoverlap}}/(t_{\text{deadline}} - t_{\text{invariant}}) > N_{\text{overlap}}/t_{\text{invariant}}$. When all the parameters are consistent with the above condition, it is possible to use multiple voltages for

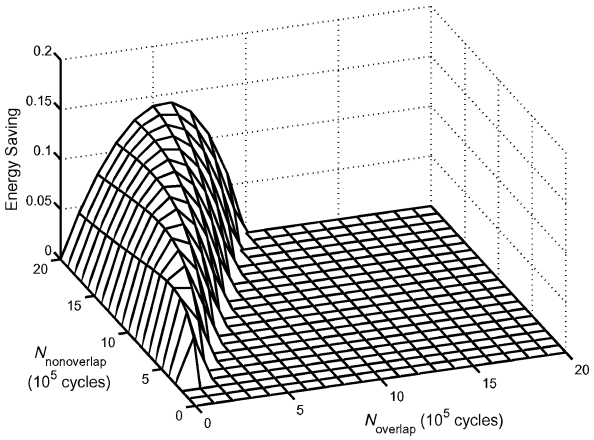


Fig. 7. Continuous case: Energy saving ratio with respect to different N_{overlap} and $N_{\text{nonoverlap}}$ ($t_{\text{deadline}} = 3$ ms, $t_{\text{invariant}} = 1$ ms).

different parts of the computation to achieve power savings over the single-voltage case. Further, the analysis tells us that two voltages suffice for this case. Energy savings is a function of N_{overlap}, $N_{\text{nonoverlap}}$, $t_{\text{invariant}}$, and t_{deadline}. To visualize the dependence, we now consider various surfaces in this space, keeping three of these parameters at fixed values and varying two of them.

Energy savings for different N_{overlap} and $N_{\text{nonoverlap}}$ are illustrated in Figure 7. For fixed t_{deadline} and $t_{\text{invariant}}$, for most cases, there is no energy savings. Recall that single frequency outcomes offer no energy savings over a fixed a priori frequency choice. As N_{overlap} increases, some computations can be executed within $t_{\text{invariant}}$ without increasing execution time. Two frequencies instead of one can then be used to achieve minimum energy, as shown in Figure 5(b). As N_{overlap} keeps increasing, eventually computation operations dominate. At this point, the "virtual" deadline set by memory operations is of no consequence and a single frequency (this time due to computation dominance) will once again be optimal. This corresponds to Figure 1(a). Thus there are again no energy savings when compared to the best static single-frequency setting.

In Figure 8, $N_{\text{nonoverlap}}$ and t_{deadline} are fixed. As $t_{\text{invariant}}$ increases, energy saving increases. This is intuitive because as the memory bottleneck increases, the opportunities for voltage scaling due to overlap slack increase.

Memory operations are cache miss memory operations, slowing down the overlap computations does not dilate the memory time-line, and thus does not impede the start of the nonoverlapping operations.

Figure 9 shows energy savings with respect to different t_{deadline} and N_{overlap}. Other parameters are fixed. When N_{overlap} is small, as t_{deadline} increases, energy savings increase. Once again, this makes intuitive sense because the greater slack gives more opportunities for energy savings. As N_{overlap} gets bigger, however, energy savings go up, achieve a maximum point and then go down again. This is because as N_{overlap} increases, the slowdown over the $t_{\text{invariant}}$ has more

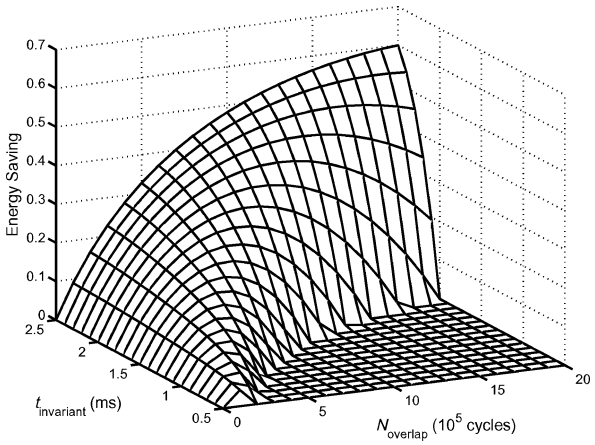


Fig. 8. Continuous case: Energy saving ratio with respect to different N_{overlap} and $t_{\text{invariant}}$ ($N_{\text{nonoverlap}} = 10^6$ cycles, $t_{\text{deadline}} = 3$ ms).

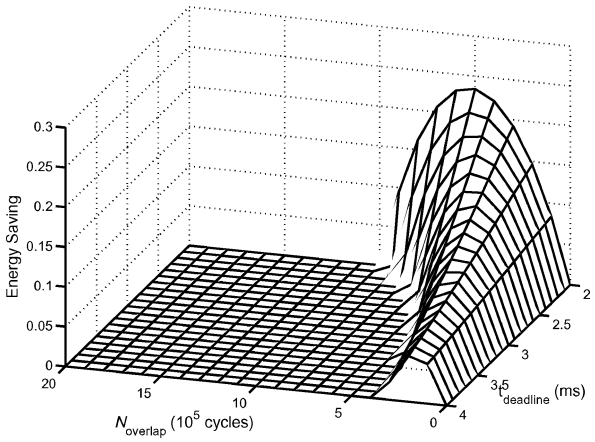


Fig. 9. Continuous case: Energy saving ratio with respect to different t_{deadline} and N_{overlap} ($N_{\text{nonoverlap}} = 10^6$ cycles, $t_{\text{invariant}} = 1$ ms).

impact on the execution time. So it is less likely two frequencies will reduce energy.

5. DISCUSSION OF MODEL DETAILS: DISCRETE VOLTAGE SETTING CASE

In the case where voltage is continuously scalable, optimal settings can always be obtained with either one or two voltage choices. In real processors, however, supply voltages are much more likely to be scalable only in discrete steps. Thus, rather than having free choice of v_1 and v_2, they must be selected from a set of values $(V_1, V_2, \ldots, V_h)$. $(F_1, F_2, \ldots, F_h)$ are the corresponding paired

frequencies. The problem becomes one of minimizing:

$$E = \sum_{i}^{h} V_i^2 x_{1i} + \sum_{j=0}^{h} V_j^2 x_{2j} = \sum_{i}^{h} V_i^2 (x_{1i} + x_{2i})$$

subject to constraint:

$$\begin{aligned} \sum_{i}^{h} \frac{(x_{1i} + x_{2i})}{F_i} &= t_{\text{deadline}} \\ \sum_{i}^{h} \frac{x_{2i}}{F_i} &\leq t_{\text{deadline}} - t_{\text{invariant}} \\ \sum_{i}^{h} x_{1i} &= N_{\text{overlap}} \\ \sum_{i}^{h} x_{2i} &= N_{\text{nonoverlap}} \end{aligned}$$

Here x_{1i} and x_{2i} are the number of cycles at voltage level i for the overlapped part and nonoverlapped part, respectively.

Leveraging off prior work by Ishihara and Yasuura [1998] allows us to progress on the problem. We have already used the result that for computation with a fixed deadline and no memory operations, with continuous voltage scaling, a single-voltage level results in the least energy. We now use a second result provided there. For the discrete case they show that the minimum energy can be obtained by selecting the two immediate neighbors of the optimum voltage in the continuous case that are available in the discrete set. Thus, for the computation-bound case, which needed a single optimum frequency, f_{opt}, in the continuous case, we know that the discrete case will require the two immediate neighbors of f_{opt} from the available voltages. What remains to be determined is the number of cycles each of these frequencies is used for.

We determine this for the computation-dominated case from Section 4. Consider the two neighboring values for voltage and frequency: $v_{\text{a}} < v_{\text{ideal}} < v_{\text{b}}$ and $f_{\text{a}} < f_{\text{ideal}} < f_{\text{b}}$. Say that x_{a} cycles are executed with voltage v_{a} and x_{b} cycles are executed with voltage v_{b}. The values for x_{a} and $_{\text{b}}$ are determined by solving for the following constraints:

$$\begin{aligned} x_{\text{a}}/f_{\text{a}} + x_{\text{b}}/f_{\text{b}} &= t_{\text{deadline}} \\ x_{\text{a}} + x_{\text{b}} &= N_{\text{overlap}} + N_{\text{nonoverlap}} \end{aligned}$$

By solving these simultaneous equations, we get

$$\begin{aligned} x_{\text{a}} &= \frac{f_{\text{a}} f_{\text{b}}}{f_{\text{b}} - f_{\text{a}}} \left(t_{\text{deadline}} - \frac{N_{\text{overlap}} + N_{\text{nonoverlap}}}{f_{\text{b}}} \right) \\ x_{\text{b}} &= \frac{f_{\text{a}} f_{\text{b}}}{f_{\text{b}} - f_{\text{a}}} \left(\frac{N_{\text{overlap}} + N_{\text{nonoverlap}}}{f_{\text{a}}} - t_{\text{deadline}} \right) \end{aligned}$$

The minimum energy consumption occurs at the point:

$$\begin{aligned} E_{\min} &= v_a^2 x_a + v_b^2 x_b \\ &= \frac{f_a f_b}{f_b - f_a}\left[\left(\frac{N_{\text{overlap}} + N_{\text{nonoverlap}}}{f_a} - t_{\text{deadline}}\right) v_b^2 \right. \\ &\quad \left. + \left(t_{\text{deadline}} - \frac{N_{\text{overlap}} + N_{\text{nonverlap}}}{f_b}\right) v_a^2\right] \end{aligned} \tag{6}$$

Consider next the memory-dominated case that resulted in two frequencies in the continuous-scaling approach. $f_{\text{invariant}} = \frac{N_{\text{overlap}}}{t_{\text{invariant}}}$ is the optimal frequency for the overlapped part in the continuous case. Similarly, $f_{\text{nonoverlap}} = \frac{N_{\text{nonoverlap}}}{t_{\text{deadline}} - t_{\text{invariant}}}$ is the optimal frequency for the nonoverlapped part. This gives us: f_a, f_b the immediate discrete neighbors of $f_{\text{invariant}}$ and f_c, f_d the immediate discrete neighbors of $f_{\text{nonoverlap}}$, as the four frequencies required in this case. What remains to be determined is the number of cycles executed at each frequency. These are obtained by solving for the following constraints:

$$\begin{aligned} x_a/f_a + x_b/f_b &= t_{\text{invariant}} \\ x_c/f_c + x_d/f_d &= t_{\text{deadline}} - t_{\text{invariant}} \\ x_a + x_b &= N_{\text{overlap}} \\ x_c + x_d &= N_{\text{nonoverlap}} \end{aligned}$$

We can get x_a, x_b, x_c, and x_d similarly and express the minimum energy as

$$\begin{aligned} E_{\min} &= v_a^2 x_a + v_b^2 x_b + v_c^2 x_c + v_d^2 x_d \\ E_{\min} &= \frac{f_a f_b}{f_b - f_a}\left[\left(\frac{N_{\text{overlap}}}{f_a} - t_{\text{invariant}}\right) v_b^2 + \left(t_{\text{invariant}} - \frac{N_{\text{overlap}}}{f_b}\right) v_a^2\right] \\ &\quad + \frac{f_c f_d}{f_d - f_c}\left[\left(\frac{N_{\text{nonoverlap}}}{f_c} - (t_{\text{deadline}} - t_{\text{invariant}})\right) v_d^2 \right. \\ &\quad \left. + \left(t_{\text{deadline}} - t_{\text{invariant}} - \frac{N_{\text{nonoverlap}}}{f_d}\right) v_c^2\right] \end{aligned}$$

f_a, f_b, f_c, f_d are staircase functions of program parameters. As program parameters change, $f_{\text{invariant}}$ and $f_{\text{nonoverlap}}$ will change. As long as they are sandwiched by the same two available frequencies, f_a, f_b, f_c, and f_d remain the same. When changes in program parameters will change $f_{\text{invariant}}$ and $f_{\text{nonoverlap}}$ so much that the frequencies sandwiching them change as well, then f_a, f_b, f_c, f_d will become the new neighbor frequencies.

5.1 Discrete Voltage Settings: Summary and Results

The main results in this case are that for the compute bound case, we can use the two voltages from the available set that are the nearest neighbors of the single optimal voltage in the continuous case. If the single optimal voltage in the continuous case is an available voltage, then only one voltage is needed. For the memory-bound case, at most four voltages are needed, the two immediate neighbors of the two optimal voltages in the continuous case. One exception is when one of the optimal frequencies from the continuous case is greater than

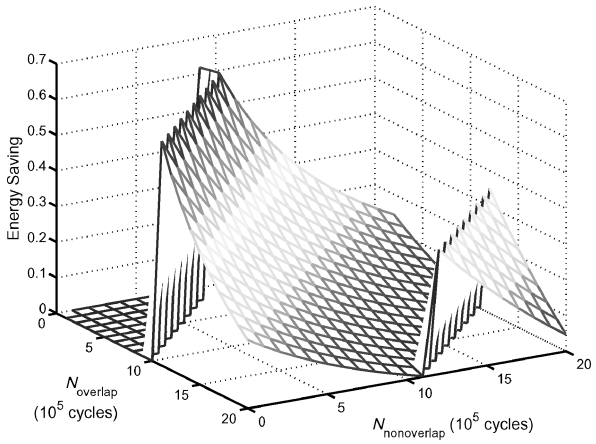


Fig. 10. Discrete case: Energy savings with respect to different N_{overlap} and $N_{\text{nonverlap}}$ relative to best single-frequency setting that meets the deadline (three voltage levels, $t_{\text{deadline}} = 5200\ \mu$s, $t_{\text{invariant}} = 1000\ \mu$s).

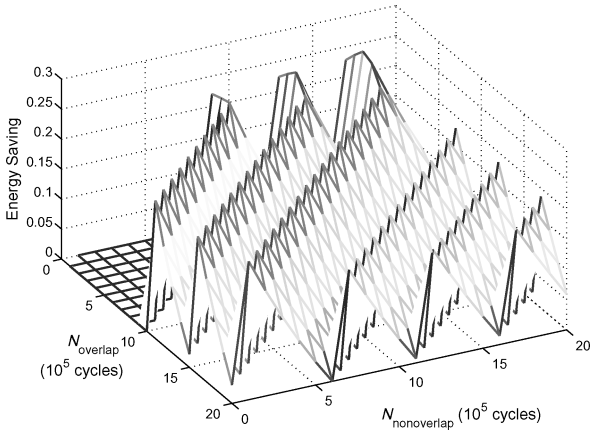


Fig. 11. Discrete case: Energy savings with respect to different N_{overlap} and $N_{\text{nonverlap}}$ relative to best single-frequency setting that meets the deadline (seven voltage levels, $t_{\text{deadline}} = 5200\ \mu$s, $t_{\text{invariant}} = 1000\ \mu$s).

the available highest frequency. In this situation, the entire program will run at the highest frequency and the program will miss its deadline anyway.

We now examine the surfaces for the energy savings obtained in terms of the dependent parameters (in Figures 10–13). The figures do a good job of conveying the complexity of the energyoptimization space when discrete voltage settings are involved. Benefits of intraprogram DVS peak and drop as one moves into regions that are either poorly served or well served by a single static frequency setting. In fact, one of the main motivations of the MILP-based DVS formulation presented in the next section is that it offers a concrete way of navigating this complex optimization space.

When more voltage settings are available, the number of peaks in the graphs increases. When there are three voltage levels as shown in Figure 10, there are

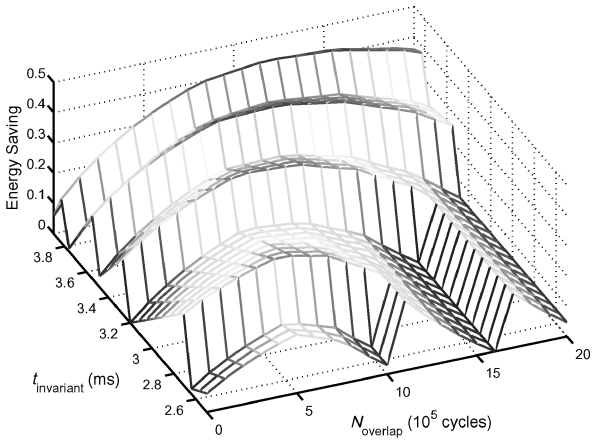


Fig. 12. Discrete voltage levels: Energy savings for different N_{overlap} and $t_{\text{invariant}}$ relative to best single-frequency setting that meets the deadline (seven discrete voltage levels, $N_{\text{nonoverlap}} = 10^6$ cycles, $t_{\text{deadline}} = 5.2$ ms).

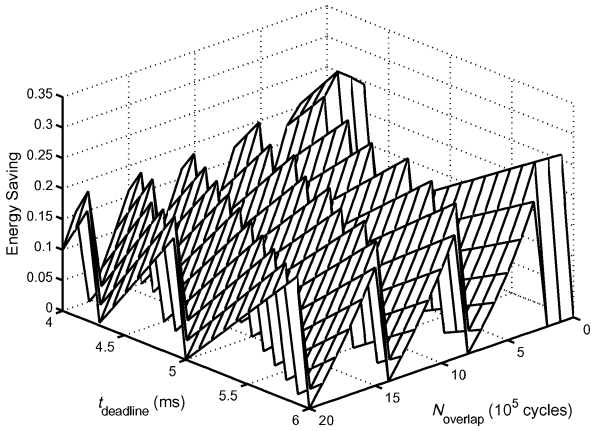


Fig. 13. Discrete case: Energy savings for different t_{deadline} and N_{overlap} relative to best single-frequency setting that meets the deadline. This graph is plotted for a case with seven possible discrete voltage levels ($t_{\text{invariant}} = 1$ ms, $N_{\text{nonoverlap}} = 10^6$ cycles).

two sets of peaks. For seven voltage levels, there are six sets of peaks as shown in Figure 11. When the step size between voltage settings is smaller, the amplitude of each peak becomes smaller as well. This follows fairly intuitively from the fact that fine-grained voltage settings allow one to do fairly well just by setting a single voltage for the whole program run; intraprogram readjustments are of lesser value if the initial setting can be done with sufficient precision.

Figure 12 shows the energy savings for different N_{overlap} and $t_{\text{invariant}}$. For a fixed $t_{\text{invariant}}$, when N_{overlap} is small so that cache miss memory operations dominate, the number of best frequencies will be four. So energy savings increase smoothly as N_{overlap} increases. As N_{overlap} gets larger, the difference between the two frequencies for the overlapped part and the nonoverlapped part gets

smaller. So energy savings drop smoothly till to the point that a single frequency minimizes the energy consumption. As N_{overlap} increases further so that computation operations dominate, the number of best frequencies will be two or one. The energy savings peak and drop correspondingly. Figure 13 shows the energy savings for different N_{overlap} and t_{deadline}. Energy savings is not monotonic with the deadline because we compare not to the highest-frequency operation, but to the best single frequency that meets the deadline.

5.2 Analytical Model Summary

Overall, the key message of the analytic model, particularly for the case of discrete voltage settings, is that while DVS offers significant energy savings in some cases, the optimization space is complex. Achieving energy savings via compile-time, intraprogram DVS seems to require a fairly intelligent optimization process. Our MILP-based proposal for managing this optimization is presented in the next section.

6. PRACTICAL ENERGY SAVINGS USING MATHEMATICAL OPTIMIZATION

Sections 4 and 5 provide a detailed analysis for the maximum energy savings possible using profile-based intraprogram DVS. As they use some simplifying assumptions, it leaves open the question as to how much of the predicted savings can actually be extracted in practice. In this and the following section, we answer this question using a practical implementation of a mathematical optimization formulation of the problem.

6.1 Overview

Here we assume that instructions or system calls are available to allow software to invoke changes in clock frequency and supply voltage. For example, in the Intel XScale processor, the clock configuration is specified by writing to a particular register [Intel Corp. 2003b]. Throughout the paper we refer to these invocations generically as "mode-set instructions," although their implementations may range from single instructions to privileged system calls. For this study we use a MILP-based technique to determine optimal placements of the mode-set instructions in the code such that total program energy is minimized, subject to meeting a constraint, or deadline, regarding the program run-time. Overall, the goal is to operate different parts of the program at the lowest possible frequency that will allow the program to meet its deadline with the least power consumption.

This MILP formulation extends the one presented by Saputra et al. [2002] by including the energy cost of a mode switch, considering finer-grain control over code regions controlled by a single setting, and considering multiple input data categories to drive the optimization.

Since executing a mode-set instruction has both time and energy cost, we wish to place them at execution points that will allow the benefit of invoking them (improvements in energy or in ability to meet the deadline) to outweigh the time/energy cost of invoking them. Thus, some knowledge is needed of the execution time and frequencies for different parts of the program. As shown

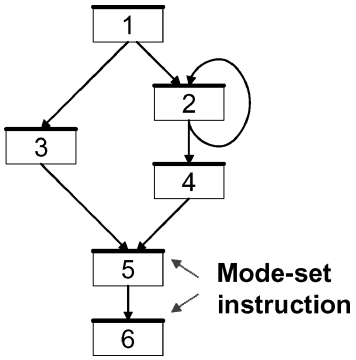


Fig. 14. An example control-flow graph.

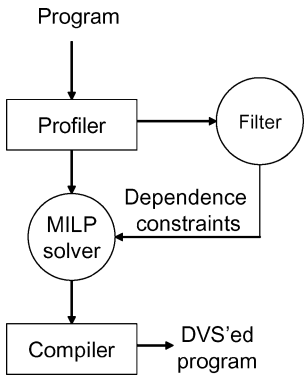


Fig. 15. Flow diagram of the technique.

in Figure 14, an initial approach might involve considering the beginning of each basic block as a potential point for inserting a mode-set instruction. Some blocks, however, such as blocks 2 or 5 in the diagram, may benefit from different mode settings depending on the path by which the program arrives at them. For example, if block 5 is entered through block 4, and this flow is along the critical path of the program, then it may be desirable to run this at a different mode setting than if it is entered through block 3, in which case it is not on the critical path.

For reasons like this, our optimization is actually based on program edges rather than basic blocks. Edge-based analysis is more general than block-based analysis; it allows us to incorporate context regarding which block preceded the one we are about to enter. Figure 15 gives the general flow of our technique. The MILP formulation, briefly described in the next section, presumes that we have profiled the program and have a set of transition counts that correspond to how often we execute a basic block by entering it through a specific edge and leaving it through a specific edge. This is referred to as the local path through a basic block. We also profile to determine the execution time and energy of each basic block for each possible voltage–frequency pair. Section 6.2 discusses our methodology further, and Section 6.3 discusses how this methodology can

be generalized to allow for profiling multiple input sets or categories of input types. We assume, as is common in current processors, that there are a finite number of discrete voltage/frequency settings at which the chip can operate. (This improves upon some prior work that relied upon a continuous tradeoff function for voltage/frequency levels [Lorch and Smith 2001]; such continuous (V, f) scaling is less commercially-feasible.) Figure 15 also shows a step where the possible set of mode instructions is passed through a filtering set, where some of them are restricted to be dependent on other instructions based on the program flow. This independent set of mode instructions is used to formulate the MILP program, which will determine the value for each mode instruction. Subsequent sections discuss the criteria used in restriction as well as its implementation.

6.2 The MILP Formulation

We start by accounting for the transition energy and time costs. Let $S_E(v_i, v_j)$ be the energy transition cost in switching from voltage v_i to v_j and $S_T(v_i, v_j)$ be the execution time switching cost from v_i to v_j.

$$S_E = (1-u) \times c \times \left|v_i^2 - v_j^2\right|$$
$$S_T = \frac{2 \times c}{I_{\text{MAX}}} |v_i - v_j|$$

Equations for S_E, S_T have been taken from Burd and Brodersen [2000], and are considered to be an accurate modeling of these transition costs. The variable c is the voltage regulator capacitance and u is the energy efficiency of the voltage regulator. I_{MAX} is the maximum allowed current. There is runtime overhead related to mode-set instruction executions but compared to the high switching overhead (thousands of cycles per switch), it is negligible. As for CPU energy during switches, we assume that clock gating techniques can reduce this significantly. Thus we ignored the overheads associated with them.

Let there be N possible modes that can be set by the mode-set instruction. For an edge (i, j) in the control-flow graph there are N binary-valued (0/1) mode variables k_{ijm}, $m = 1, 2, \ldots, N$. $k_{ijm} = 1$ if and only if the mode-set instruction along edge (i, j) sets the mode to m as a result of the DVS scheduling, and is 0 otherwise. Since each edge sets the mode to at most one value, we have the following constraint among the mode variables for a given edge (i, j): $\sum_{m=1}^{N} k_{ijm} = 1$. With this, the optimization problem to be solved is to minimize:

$$\sum_{i=1}^{R}\sum_{j=1}^{R}\sum_{m=1}^{N} k_{ijm} G_{ij} E_{jm} + \sum_{h=1}^{R}\sum_{i=1}^{R}\sum_{j=1}^{R} D_{hij} S_E(\overrightarrow{k_{hi}}, \overrightarrow{k_{ij}})$$

subject to the following constraint:

$$\sum_{i=1}^{R}\sum_{j=1}^{R}\sum_{m=1}^{N} k_{ijm} G_{ij} T_{jm} + \sum_{h=1}^{R}\sum_{i=1}^{R}\sum_{j=1}^{R} D_{hij} S_T(\overrightarrow{k_{hi}}, \overrightarrow{k_{ij}}) \leq \text{deadline}$$

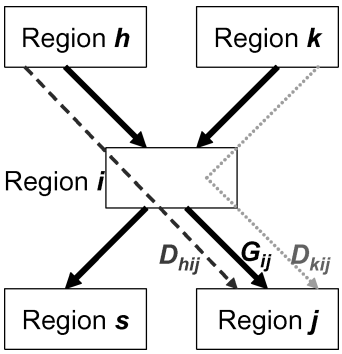


Fig. 16. Edge graph and corresponding definitions.

where:

R The number of regions, that is, nodes such as basic blocks in a control-flow graph. It is different for different programs.

N The number of mode settings. It is determined by the processor's available mode settings.

k_{ijm} The mode variable for mode m on edge (i, j). $k_{ijm} = 1$ if and only if the mode-set instruction along edge (i, j) sets the mode to m as a result of the scheduling, and is 0 otherwise. For example, if edge i, j chooses mode 2, then $k_{ij2} = 1$ and other $k_{ijm} = 0$ $(m \neq 2)$. $\overrightarrow{k_{ij}}$ is the set of mode variables (N in all) for edge (i, j).

E_{jm} The energy consumption for a single invocation of region j under mode m.

G_{ij} The number of times region j is entered through edge (i, j) as shown in Figure 16.

D_{hij} The number of times region i is entered through edge (h, i) and exited through edge (i, j) as shown in Figure 16.

T_{jm} The execution time for a single invocation of region j under mode m.

These last four values are all constants for a given program determined by profiling. If we let V_m be the supply voltage of mode m, then S_E is the transition energy cost for one mode transition, such that

$$S_E(\overrightarrow{k_{hi}}, \overrightarrow{k_{ij}}) = c \times (1 - u) \left| \sum_{m=1}^{N} k_{him} V_m^2 - \sum_{m=1}^{N} k_{ijm} V_m^2 \right|$$

Likewise, S_T, the transition time cost for one mode transition, is represented as

$$S_T(\overrightarrow{k_{hi}}, \overrightarrow{k_{ij}}) = \frac{2 \times c}{I_{\text{MAX}}} \left| \sum_{m=1}^{N} k_{him} V_m - \sum_{m=1}^{N} k_{ijm} V_m \right|$$

In the objective function the first summation represents the energy consumption of all regions and the second represents the energy cost in transition between modes. The constraint ensures that the execution time deadline is met.

Again, here the first summation represents the execution time for the regions and the second represents the execution time cost in transition between modes.

The introduction of the mode variables instead of the voltage variables linearizes the energy and execution time costs E_i and T_i for region i. While S_E and S_T are still nonlinear due to the absolute value term, there is no quadratic dependence on the variables; the V_m term in S_E is now a constant. The absolute value dependence can be linearized using a straightforward technique. To remove the absolute value, $|x|$, we introduce a new variable y to replace $|x|$ and add the following additional constraints: $-y \leq x \leq y$. Applying this technique to S_E and S_T, the formulation is completely linearized as follows:
Minimize:

$$\sum_{i}^{R}\sum_{j}^{R}\sum_{m}^{N} G_{ij}k_{ijm}E_{jm} + \sum_{h}^{R}\sum_{i}^{R}\sum_{j}^{R} D_{hij}e_{hij}C_E$$

subject to the constraints:

$$\sum_{i}^{R}\sum_{j}^{R}\sum_{m}^{N} G_{ij}k_{ijm}T_{jm} + \sum_{h}^{R}\sum_{i}^{R}\sum_{j}^{R} D_{hij}t_{hij}C_T \leq \text{deadline}$$

$$\sum_{m}^{N} k_{ijm} = 1$$

$$-e_{hij} \leq \sum_{m}^{N}(k_{him}V_m^2 - k_{ijm}V_m^2) \leq e_{hij}$$

$$-t_{hij} \leq \sum_{m}^{N}(k_{him}V_m - k_{ijm}V_m) \leq t_{hij}$$

The absolute value operations in the switching time and energy relationships have been removed; the new variables e_{hij} and t_{hij} are part of constraints introduced for their removal, and $C_E = c \times (1-u)$, $C_T = \frac{2\times c}{I_{\text{MAX}}}$ are constants related to switching energy and time in the linearized form.

Note that while each edge has a mode-set instruction, if at run-time the mode value for an edge is the same as the previous one, no transition cost is incurred. This is due to the nature of the transition cost functions S_E and S_T, which, as expected, have nonzero value only if the two modes are distinct. Thus, a mode-set instruction in the backward edge of a heavily executed loop will be "silent" for all but possibly the first iteration. A post-pass optimization within a compiler can easily hoist some such instructions out of the loop.

As mentioned in Section 6.1, the run-time for the MILP solver can be significantly reduced by a careful restriction of the solution space. The mode instruction on some edge (i, j) can be forced to have the same value as the mode instruction on some other edge (u, v), so that $\overrightarrow{k_{ij}} = \overrightarrow{k_{uv}}$. This reduces the number of independent variables for the MILP solver, and consequently its run-time. While this restriction can potentially result in some loss of optimality in the objective function, the deadline constraints are still met. The practical issues

in deciding which edges to select for this restriction are discussed in the experimental section.

6.3 Handling Multiple Data Sets

The formulation described thus far optimizes based on a single profile run from a single input data set. Here, we extend the methodology to cover multiple categories of inputs. While different data inputs typically cause some variation in both execution time and energy, one can often sort types of inputs into particular categories. For example, with mpeg, it is common to consider categories of inputs based on their motion and complexity.

The MILP-based scheduling algorithm can be adapted to handle multiple categories of inputs. For each category of inputs, a "typical" input data set is chosen. The goal is to minimize the weighted average of energy consumption of different input data sets while making sure that the execution time using different typical input data sets meets a common or individual deadlines.

The formulation is remodeled as working to minimize:

$$\sum_g p_g \left(\sum_i \sum_j \sum_m k_{ijm} G_{ijg} E_{jmg} + \sum_h \sum_i \sum_j D_{hijg} e_{hij} C_E \right)$$

subject to the following constraints:

$$\begin{aligned}
\forall g \quad \sum_i \sum_j \sum_m k_{ijm} G_{ijg} T_{jmg} + \sum_h \sum_i \sum_j D_{hijg} t_{hij} C_T &\le \text{deadline} \\
\sum_m k_{ijm} &= 1 \\
-e_{hij} \le \sum_m \left(k_{him} V_m^2 - k_{ijm} V_m^2 \right) &\le e_{hij} \\
-t_{hij} \le \sum_m (k_{him} V_m - k_{ijm} V_m) &\le t_{hij}
\end{aligned}$$

where

- p_g The possibility of input category g as input.
- k_{ijm} The mode variable for mode m on edge (i, j) as in the single data set case. Since k_{ijm} is assigned regardless of the input, k_{ijm} is independent of input.
- E_{jmg} The energy consumption of region j in mode m for input data in category g.
- T_{jmg} The execution time of region j in mode m for input data in category g.
- G_{ijg} The number of times region j is entered through edge (i, j) for input data in category g.
- D_{hijg} The number of times region i is entered through edge (h, i) and exited through edge (i, j) for input data in category g.

The other terms are the same as before.

These modifications retain the linearity of the objective function and constraints. The objective function now minimizes the weighted average energy

over the different categories, and the deadline constraints ensure that this is done while obeying the deadline over all categories. If applicable, this formulation also allows for having a separate deadline for different categories if needed.

7. DVS IMPLEMENTATION USING PROFILE-DRIVEN MILP

As shown in Figure 15, our optimal frequency setting algorithm works by profiling execution, filtering down to the most important frequency-setting opportunities, and then sending the results to an MILP solver. This section describes this flow in greater detail, with the following subsections discussing the profiler, filter, and solver steps respectively.

7.1 Simulation-Based Program Profiling

As already described, our MILP approach requires profiling data on the per-block execution time, per-block execution energy, and local path (the entry and exit for a basic block) frequencies through the program being optimized. While the local path frequencies need only be gathered once, the per-block execution times and energies must be gathered once per possible mode setting. This is because the overlap between CPU and memory instructions will mean that the execution time is not a simple linear scaling by the clock frequency. (That is, we assume that memory is asynchronous relative to the CPU and that its absolute response time is unaffected by changes in the local CPU clock.)

To gather the profile data for the experiments shown here, we use simulation. We note, however, that other means of profiling would also work well. One could, for example, use hardware performance counters to profile both performance and energy data for real, not simulated, application runs [Joseph and Martonosi 2001].

The data shown here have been gathered using the Wattch power/performance simulator [Brooks et al. 2000], which is based on SimpleScalar [Burger et al. 1996]. Our simulations are run to completion for the provided inputs, so we get a full view of program execution. (Sampling methods might be accurate enough to give good profiles while reducing profile time.) For both our time/energy profiles and for our experimental results in subsequent sections, we used the simulation configuration listed in Table I. We assume that the CPU has three scaling levels for (V, f). They are a frequency of 200 MHz paired with a supply voltage of 0.7 V, 600 MHz at 1.3 V, and a maximum performance setting of 800 MHz at 1.65 V. This is similar to some of the voltage–frequency pairings available in Intel's XScale processors [Clark 2001].

Four runs are needed to gather the required information for the program graph. During the first run, the information about basic blocks of the program is collected. Basic blocks are identified and labeled and their starting and ending addresses are recorded. During the next three runs, energy consumption and execution time for each basic block are recorded for three (V, f) pairs, respectively. During the profiling for (0.7 V, 200 MHz), the local path frequencies are gathered.

Table I. Configuration Parameters for CPU Simulation

Parameter	Value
RUU size	64 instructions
LSQ size	32 instructions
Fetch Queue size	8 instructions
Fetch width	4 instructions/cycle
Decode width	4 instructions/cycle
Issue width	4 instructions/cycle
Commit width	4 instructions/cycle
Functional Units	4 integer ALUs 1 integer multiply/divide 1 FP add, 1 FP multiply 1 FP divide/sqrt
Branch Predictor	Combined, bimodal 2K table 2-level 1K table, 8bit history 1K chooser
BTB	512-entry, 4 way
L1 data-cache	64K, 4-way (LRU) 32B blocks, 1 cycle latency
L1 instruction-cache	same as L1 data-cache
L2	Unified, 512K, 4-way(LRU) 32B blocks, 16-cycle latency
TLBs	32-entry, 4096-byte page

7.2 Filtering Edges to Reduce MILP Solution Time

While our MILP approach generally works in practice for even large programs, the MILP solution time may take hours, which is not desirable. The run-time can be reduced by adding filtering rules to the MILP approach as discussed in Section 6.2. The frequencies in certain regions may be linked to (i.e., the same as) the frequencies in other regions. This reduces the number of independent variables for the ILP solver. We started with filtering the small basic blocks with negligible energy consumption. In other words, we forced those small basic blocks to have the same mode as their precedent basic blocks. This filtering rule has an unbounded effect on the optimality because it did not take into consideration the local path information.

We next considered filtering the edges that are considered as candidates for mode-set instructions. A simple and intuitive rule for doing this is as follows.

Our goal is to identify edges (i, j) such that the total power consumption of block j when entered from (i, j) is relatively negligible. In this case, not much is lost by giving up the flexibility of independently setting the mode instruction along (i, j). If this edge is selected, then its mode value can be made to be the same as that for edge (k, i), which has the largest count (obtained during profiling) for all incoming edges to block i. The motivation for this is that it will result in edge (i, j) not changing its mode whenever block i is entered from edge (k, i). This edge filtering rule works better than the filtering rule for basic blocks. We refine the filter rule further to make the impact local. We still identify edge (i, j) and edge (k, i) as described above. However, only if the energy consumption of block i when entered from edge (k, i) is also relatively negligible, we assign the mode value for (i, j) the same as (k, i). This filtering rule is the rule we used in all experiments.

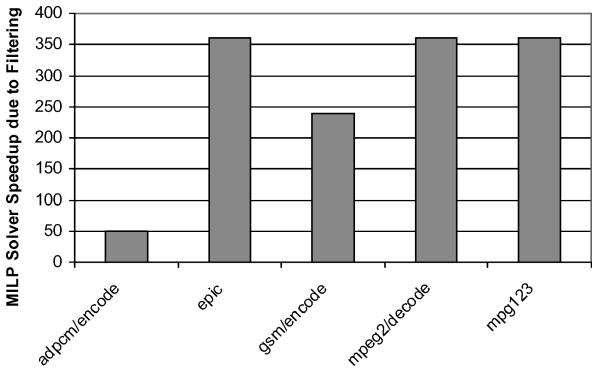


Fig. 17. Speedup in MILP solution time when edge filtering is applied.

Table II. Energy Consumption when the MILP Solver is Run on the Full Set of Program Edges (left) or the Filtered Subset of Transition Edges (right)

Benchmark	All:Energy (μJ)	Subset:Energy (μJ)
adpcm/encode	10194.3	10195.4
epic	33021.9	33021.9
gsm/encode	72287.6	72287.6
mpeg/decode	122392.8	122392.8
mpg123	37291.4	37291.4

The selection rule is as follows. We filter out all edges whose total destination energy is in the tail of the energy distribution that cumulatively comprises less than 2% of the total energy (for an arbitrarily selected mode). Filtered edges are still considered as far as timing constraints are concerned, so all deadlines are met. Filtering only affects the energy achieved.

7.3 Mathematical Programming: Details

Once profiles have been collected and filtering strategies have been applied, the transition counts and the program graph structure are used to construct the equations that express DVS constraints. We use AMPL [Fourer et al. 1993] to express the mathematical constraints and to enable pruning and optimizations before feeding the MILP problem into the CPLEX solver [ILOG CPLEX 2002].

As shown in Figure 17, our edge filtering method greatly prunes the search space for the MILP solver, and brings optimization times down from hours to seconds. (We gather these data for five of the MediaBench applications [Lee et al. 1997], with a transition time of 12 μs, and transition energy of 1.2 μJ.)

Table II shows that for the benchmarks considered the minimum energy determined by the solver remain essentially unchanged from the case when the full set of edges is considered. As discussed in Section 6.2, the deadlines will still be met exactly, even with the filtering in place.

8. EXPERIMENTAL RESULTS

This section provides experimental results showing the improvements offered by “real-world optimal” DVS settings chosen by MILP.

8.1 Benchmarks and Methodology

Our method is based on compile-time profiling and user-provided (or compiler-constructed) timing deadlines. To evaluate it here, we focus on multimedia applications in which one can make a solid case for the idea that performance should meet a certain threshold, but beyond that threshold, further increases in performance are of lesser value. For example, once you can view a movie or listen to an audio stream in real-time, there is lesser value in pushing speed beyond real-time playback; as long as a specified speed level has been reached, we argue that energy savings should be paramount.

The benchmarks we have chosen are applications from the MediaBench suite [Lee et al. 1997; MediaBench II 2003], which represent a wide range of memory access behavior from very little (adpcm/encode) to very intense (mpg123 and epic). The benchmark gsm has a medium memory access rate, while mpeg is chosen because it provides multiple input data sets. Unless otherwise specified, we use the inputs provided with the suite, and we run the programs to completion.

8.2 Impact of Transition Cost

Changing a processor's voltage and frequency has a cost both in terms of delay and in terms of energy consumption. Thus, the time or energy required to perform a DVS mode setting instruction can have an important impact on the DVS settings chosen by the MILP approach, and thus the total execution time and energy. Frequent or heavyweight switches can have significant time/energy cost, and thus the MILP solver is less likely to choose DVS settings that require them.

The first experiment we discuss here shows the impact of transition cost on minimum energy. As given by the equations in Section 6.2, transition time and transition energy are both functions of the power regulator capacitance as well as the values of the two voltages that the change is between. Thus, for a given voltage difference, one can explore the impact of different switching costs by varying the power regulator capacitance, c. As c drops, so do both transition costs.

In the data shown here, we examine five power regulator capacitances. They show a range of transition costs from much higher to much lower than those typically found in real processors. A typical capacitance c of 10 μf yields 12 μs transition time and 1.2 μJ transition energy cost for a transition from 600 MHz/1.3 V to 200 MHz/0.7 V. This 12 μs transition time corresponds well to published data for XScale [Intel Corp. 2003b]. We used a wide range of c from 100 to 0.01 μf in our experiments. In order to focus on transition cost in this experiment, we hold the deadline constant. In particular, all benchmarks are asked to operate at a deadline that corresponds to point 5 in Figure 19. This is given, for each benchmark, by the time in the "Deadline 5" column of Table III. This range of deadlines will be discussed shortly in more detail when we examine the impact of different deadlines on energy savings.

Results for five MediaBench benchmarks are shown in Figure 18. For each benchmark, the energy is normalized to that program running at a fixed

Table III. Deadline Boundaries and Chosen Deadlines for Benchmarks (ms)

	Exec Time (MHz)			Deadline				
Benchmark	200	600	800	5	4	3	2	1
adpcm/encode	29.5	9.9	7.4	29.0	20.0	10.0	8.1	7.6
epic	152.6	53.6	41.0	150.0	100.0	60.0	50.0	45.0
gsm/encode	334.0	111.4	83.6	333.0	220.0	120.0	100.0	90.0
mpeg/decode	557.6	187.3	141.0	557.6	300.0	190.0	181.0	151.0
mpg123	177.7	59.2	44.4	177.6	100.0	60.0	58.0	45.0

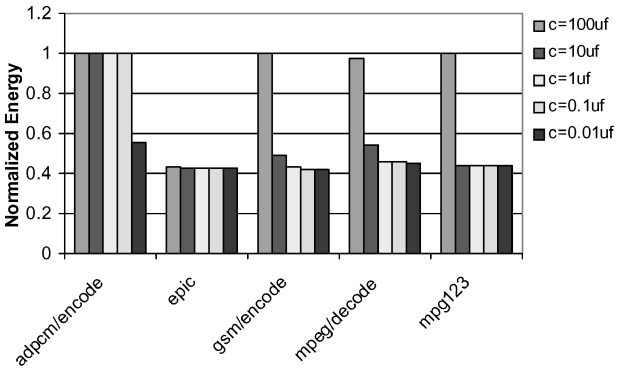


Fig. 18. Impact of transition cost. Energy is normalized to minimum energy without transition.

Fig. 19. Positions of deadlines.

600 MHz clock rate. This clock rate is sufficient to meet the deadline, so for very high transition costs ($c = 100$ μf), there are few or no transitions and so the energy is the same as in the base case. At the highest transition cost shown, there are fewer than 10 transitions executed for most of the benchmarks across their whole run.

As c decreases, transition costs drop, and so does the minimum energy. This is because when transition cost drops, there are more chances to eliminate the slack by having more and more of the program execute at 200 MHz. For example, in the mpeg benchmark, zero transitions are attempted at the highest transition cost, while at the lowest one, a run of the benchmark results in a total of over 112,000 mode-setting instructions being executed (dynamic counts).

8.3 Impact of Deadline on Program Energy

The second experiment shows the impact of deadline choice on minimum energy. Although the absolute values of the deadlines vary for each benchmark, the deadline positions we choose are illustrated abstractly and generally in Figure 19. For the most aggressive deadlines (these smaller times are towards the left-hand side) the program must run at the fastest frequency to meet the

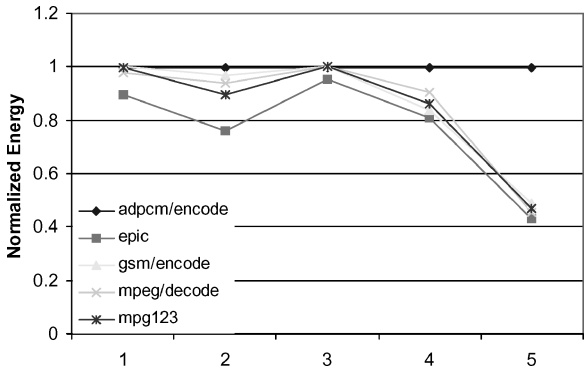


Fig. 20. Impact of deadline on energy. Energy is normalized to the energy of the best of the three possible single-frequency settings.

deadline. Toward the right-hand side of the figure, denoting the very lax deadlines, programs can run almost entirely at the low-energy 200 MHz frequency and yet still meet the deadline. Between these two extremes, programs will run at a mix of frequencies, with some number of transitions between them.

To make this more concrete for the benchmarks we consider, Table III includes the run-times of each benchmark when operating purely at 800, 600, or 200 MHz without any transitions. To test MILP-based DVS on each benchmark, we choose five application-specific deadlines per benchmark that are intended to exercise different points along the possible range. These chosen deadlines are also given in Table III. The results here are shown for a "typical" transition cost of $c = 10$ μf.

Figure 20 shows the optimized energies for these experiments. Energies are normalized to the energy of the best single frequency based on interprogram DVS scheduling. Moving from deadline 1 (stringent) towards deadline 5 (lax) the program energy is reduced by nearly a factor of 2 or more. Across the range, the MILP solver is able to find the operating point that offers minimal energy while still meeting the deadline. Benchmark adpcm/encode has few memory operations and the program body consists of big loops entangled with each other, so no intraprogram adjustment is needed and the single frequency chosen by interprogram DVS is optimal.

As shown in Figure 21, the chosen deadline can sometimes have an effect on the required solution time. In some cases (e.g., gsm/encode), the solution time can dramatically change with changing deadlines, reflecting the changing complexity of the solution space. While the effect varies with the benchmarks, in general, deadline 1 is likely to have relatively short solution times because there are relatively few possible frequency settings that satisfy it. Deadline 5, on the other hand, tends to have short solution times because the minimum energy case can be gotten with fairly uniformly slow clock speeds. Deadlines towards the middle of the space are more likely to cause long run-times because there is a rich collection of frequency settings that may be possible solutions. In this middle portion of the optimization space, MILP run-times are heavily sensitive to the MILP solver's algorithmic approaches. For gsm/encode,

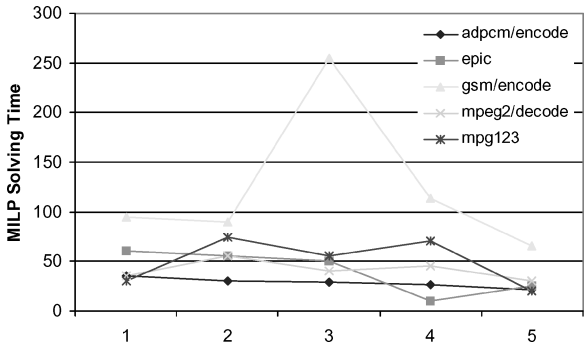


Fig. 21. MILP solution time (in seconds) for different deadlines.

Table IV. Dynamic Mode Transition Counts

	adpcm/encode	epic	gsm/encode	mpeg/decode	mpg123
Deadline 1	0	4	1	5	190
Deadline 2	0	519	2777	2645	1559
Deadline 3	0	552	85	5	936
Deadline 4	0	492	8206	2645	1550
Deadline 5	0	4	1845	5	6

the CPLEX solver generates more simplex iterations for the optimization at deadline 3.

Table IV shows the variations in the dynamic mode transition counts for the benchmarks for different deadlines (for $c = 10$ μf transition cost). At the extremes (deadlines 1 and 5) there are few choices and thus not too many mode transitions. However, closer to the middle, we see significant mode transitions for most benchmarks as they have all three (V, f) choices to draw from. This demonstrates the ability of the formulation to navigate the range of choices, and switching many times to find the best (V, f) choice for each part of the program.

8.4 Results for Multiple Profiled Data Inputs

The results here demonstrate the resilience of energy choices across different input data sets as well as the result of optimization for average energy as formulated previously. We focus here on the mpeg benchmark, and we examine four different data inputs. The inputs can be considered to fall into two different categories, based on different encoding options. The first category uses no 'B' frames; it includes 100b.m2v and bbc.m2v. The second category uses two 'B' frames between I and P frames; it includes flwr.m2v and cact.m2v. All mpeg files are Test Bitstreams from MpegTv [1998].

Figure 22 looks in detail at the energy (in mJ) for different input data and profiling runs for the mpeg benchmark. In particular, the x-axis shows four different input files for the benchmark. For each benchmark, we show the energy results from four different profiling approaches. The leftmost bar shows the energy dissipation for an mpeg run on that input file when the profiling run is also on that input file. The second bar shows the energy dissipation in

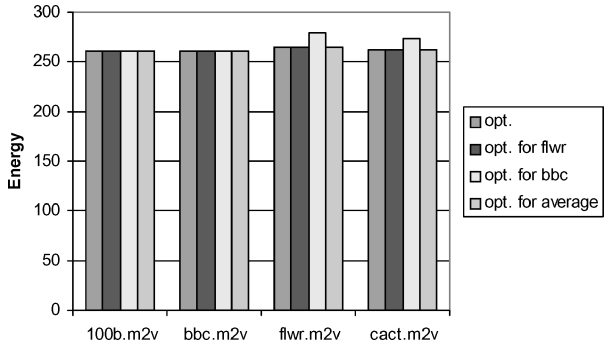


Fig. 22. Dependence of program energy on input data used for MILP profiling.

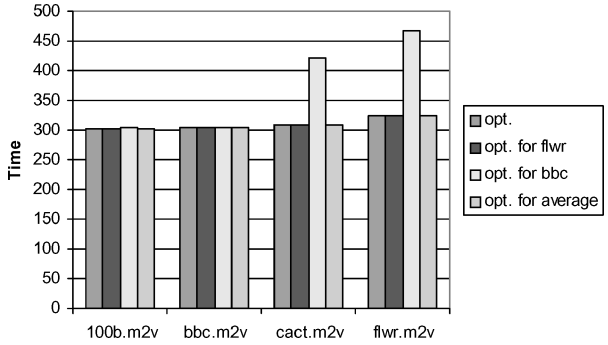


Fig. 23. Dependence of program runtime on input data used for MILP profiling.

which the profiling data were collected using the flwr input set for all runs. The third bar shows the energy dissipation when the bbc input was used for the profiling runs. The rightmost bar shows the energy dissipation when optimization is done for the average of flwr and bbc input sets (with equal weight). The sensitivity of the energy results to the specific profile inputs is fairly modest overall.

Figure 23 shows program execution times for different input data and profiling runs for the mpeg benchmark. Once again, optimizing based on the flwr profile data is nearly as good as optimizing based on the identical input data across the board. The data show that the multi-input case is often nearly as good as optimizing based on the identical input. An exception, however, is that optimizing based on the bbc input leads to poor execution time estimation. We believe this is because the bbc input is from the input category with no 'B' frames, so the MILP solver does poorly in estimating the time and energy impact of the code related to their processing. Finally, optimizing for the average case makes sure that the deadlines are met for both the cases being considered. Further, Figure 23 also illustrates the representative nature of these two input sets. Using the average case (rightmost bar) works as well as using the single profile data set (leftmost bar) across the board—even when the specific data sets are not included in the average as with cact and 100b. We have similarly

Table V. Simulation Results of Program Parameters

Benchmark	N_{overlap} (kcycles)	$N_{\text{nonoverlap}}$ (kcycles)	$t_{\text{invariant}}$ (μs)
adpcm/encode	1.6	5415.1	1.0
epic	2240.7	19431.6	3567.3
gsm/encode	7.6	43518.4	50.6
mpeg/decode	1443.0	71136.6	2577.0
mpg123	4157.5	16646.1	5615.0

Table VI. Analytically Derived Upper Bound of Energy Saving Ratio for Different Voltage Levels

Benchmark	Voltage Levels	Deadline				
		1	2	3	4	5
adpcm/encode	3	0.62	0.37	0.02	0.15	0.06
	7	0.23	0.02	0.05	0.19	0.08
	13	0.11	0.03	0.06	0.09	0.09
epic	3	0.61	0.32	0.04	0.30	0.08
	7	0.21	0.22	0.12	0.13	0.10
	13	0.10	0.11	0.02	0.03	0.11
gsm/encode	3	0.60	0.37	0.10	0.33	0.12
	7	0.21	0.02	0.03	0.17	0.15
	13	0.10	0.02	0.05	0.06	0.05
mpeg/decode	3	0.66	0.37	0.03	0.26	0.07
	7	0.25	0.02	0.10	0.09	0.08
	13	0.13	0.03	0.11	0.11	0.09
mpg123	3	0.65	0.25	0.05	0.31	0.09
	7	0.24	0.12	0.18	0.15	0.11
	13	0.12	0.02	0.07	0.04	0.02

measured the sensitivity of energy results to specific profile inputs and have found results as good or better than the run-time results presented here; the sensitivity is fairly modest overall.

8.5 A Comparison of Analytical and Profile-Driven Results

To better understand energy trends in the discrete voltage case, we compare the analytical results with the profile-driven results for the same set of benchmarks considering situations with 3, 7, or 13 available voltage levels. For seven voltage levels, we used 200, 300, 400, 500, 600, 700, and 800 MHz. For 13 voltage levels, the range is from 200 to 800 MHz with 50 MHz steps. In all experiments, $\alpha = 1.5$, $v_t = 0.45$ V, and we considered five different deadlines as elaborated above. For the profile-driven results, the voltage regulator capacitance $c = 0.01$ μf is used.

By using cycle-level CPU simulation to get the key program parameters N_{overlap}, $N_{\text{nonoverlap}}$, and $t_{\text{invariant}}$ for every instruction as shown in Table V, we plugged values into the analytic models generated in Section 5 and discussed the resulting maximum energy savings predicted by the models in Table VI. Now, Table VII gives the energy savings results for the same programs when run through the MILP-based optimization process. Because the analytical model makes optimistic assumptions about switching time and energy as well as the abilities to control for every independent dynamic instruction execution, it is

Table VII. Simulation Results for Optimized Mode Settings of Energy Savings for Different Numbers of Voltage Levels

Benchmark	Voltage Levels	Deadline				
		1	2	3	4	5
adpcm/encode	3	0.49	0.23	0.00	0.03	0.01
	7	0.16	0.01	0.01	0.04	0.01
	13	0.09	0.01	0.02	0.02	0.02
epic	3	0.57	0.30	0.03	0.27	0.05
	7	0.18	0.19	0.10	0.10	0.07
	13	0.10	0.09	0.01	0.01	0.08
gsm/encode	3	0.57	0.37	0.09	0.32	0.13
	7	0.18	0.02	0.03	0.16	0.14
	13	0.10	0.02	0.05	0.06	0.05
mpeg/decode	3	0.60	0.34	0.03	0.24	0.05
	7	0.21	0.02	0.09	0.08	0.07
	13	0.13	0.02	0.11	0.10	0.08
mpg123	3	0.60	0.26	0.06	0.31	0.07
	7	0.20	0.12	0.17	0.14	0.10
	13	0.12	0.02	0.06	0.04	0.00

expected to be an optimistic bound, and indeed, the savings predicted by the analytical model exceed those of MILP-based approaches at all but a few points. (For those points, the simulation energy savings exceeds that of the analytical model by 0.01. This is because the processor we simulate allows multiple outstanding cache misses, so not all program parameters are strictly constants. However, the difference is 0.01, so we feel the estimation from the analytical model is close enough.) Nonetheless, the general trends in both tables are similar. Further, the comparison shows that the analytical bounds are close enough to be of practical value.

Because energy savings is not monotonic with deadline and because the optimization space is relatively complex, an MILP-based approach seems to be an important enabling technique for compile-time, intraprogram DVS.

A second message here is that as we increase the number of available voltage levels, the benefits of DVS scheduling decrease significantly. In fact it could well be argued that if circuit implementations permit a very large number of DVS settings, it may not be worth resorting to intraprogram DVS—a single-voltage selection can come close enough. This is not surprising given the results for our model with continuous voltage scaling, which is the limiting case of increasing the number of discrete levels. We would like to highlight this important by-product of our modeling—for the case of only interprogram and no intraprogram DVS, our model can help determine a single optimal voltage based on a few simple parameters.

We need to point out here that our conclusions do not contradict the conclusions from the Power-Management Point (PMP) scheme [AbouGhazaleh et al. 2003]. The difference comes from the base case. In AbouGhazaleh's work, the energy savings are based on the non-DVS case, while the energy in our results is compared to the minimum energy obtained from the interprogram DVS.

We focus on single-task environments in this paper. However, we also explore the applicability of intraprogram DVS for real-time multitasking environments. This is discussed qualitatively in the next section.

9. CONSIDERING REAL-TIME MULTITASKING ENVIRONMENTS

A real-time system is a system that can support the execution of applications with time constraints. Time constraints are central to a real-time system because the failure of timely completion of certain tasks may cause malfunctions. For example, if a flight controller does not respond to a signal on time, it may lead to a crash. The time constraints are typically represented as deadlines on repeatedly invoked tasks [Liu and Layland 1973]. Some tasks, referred to as periodic tasks, are invoked at regular intervals, while other tasks, referred to as sporadic tasks, are invoked at arbitrary times with a specified minimum time interval between invocations. An example of a periodic task is the screen refresh program, which refreshes a computer screen once every 1/60 s. Hence 1/60 s, the period of the periodic task, is the deadline. The task that responds to a user clicking a mouse is a sporadic task. It occurs repeatedly but the time interval between two clicks varies. There is a minimum time interval between two occurrences, which is determined by human response time. The task has to be finished within the minimum time interval in order to deal with next possible mouse movement, so the minimum time interval can be treated as the deadline.

The goal of a real-time multitasking system is to schedule the tasks so that each task completes the execution before a specified deadline. This is a scheduling problem. The scheduling model for real-time systems has been proposed and well studied [Jeffay et al. 1991; Pillai and Shin 2001]. Our discussion is based on the established real-time system model and we will start by describing the model. In the following discussion, we will use the terms program and task alternately.

9.1 Scheduling Model for Multitask Environments

Given a real-time system, a task set is schedulable if every task meets its specified deadline. For periodic tasks, deadlines are defined as the next invocation (release) times. For sporadic tasks, deadlines are defined as the sum of the current invocation time and the specified minimal time interval. Since determining the feasibility or schedulability of a task set is beyond our scope, we consider schedulable task sets only. In particular, we consider independent task sets that are schedulable under the early deadline first (EDF) scheduling policy using the highest frequency.

An energy-aware scheduling aims at minimizing the energy consumption of all tasks while guaranteeing every task meets its deadline. We assume the scheduler schedules the tasks in EDF order and thereby we focus on voltage scaling scheduling only. The voltage scaling scheduling can be done by a real-time scheduler or by mode-set instructions inserted as long as the task set is schedulable using the assigned DVS settings.

For sporadic tasks, the future invocation time is unknown at compile time. Since compile-time intraprogram DVS requires everything to be known a priori, its applicability for sporadic tasks is limited to the following scenario: intraprogram DVS applied on top of dynamic interprogram DVS to get extra energy savings by slowing down memory-bounded regions. This does not require any

additional concepts over what has already been established. Thus we will focus on periodic tasks from now on.

We divide scheduling policies into preemptive scheduling and non-preemptive scheduling. In preemptive scheduling, a low-priority task can be suspended on the arrival of a higher priority task. A non-preemptive scheduler, on the other hand, executes the current task till completion, even if a higher priority task has arrived. Based on this classification, we will discuss two cases separately:

(1) non-preemptive scheduling on periodic tasks
(2) preemptive scheduling on periodic tasks

9.2 Non-preemptive Scheduling on Periodic Tasks

9.2.1 *Fixed Execution Time.* If we assume the execution time of each task is fixed, we can extend the MILP formulation to multiple tasks easily if every task is treated as a "region" as shown below.

Minimize:

$$\sum_i \sum_j \sum_m E_{im} k_{ijm}$$

Subject to the following constraint:

$$\begin{aligned} du_{ij} + \sum_m k_{ijm} T_{im} &\le dv_{ij} \\ dv_{ij} &\le (j+1)P_i \\ du_{ij} &\ge jP_i \\ dv_{hg} &\le du_{ij} \quad \text{if } (g+1)P_h \le (j+1)P_i \text{ and } h \ne i \\ dv_{ij} &\le du_{hg} \quad \text{if } (g+1)P_h > (j+1)P_i \text{ and } h \ne i \\ \sum_{m=1}^{N} k_{ijm} &= 1 \end{aligned}$$

where

P_i The period of task i.
k_{ijm} The mode variable for the jth invocation of task i. $k_{ijm} = 1$ if and only if the mode is set to m as a result of the scheduling, and is 0 otherwise.
E_{im} The energy consumption of task i under mode m.
T_{im} The execution time of task i under mode m.
du_{ij} The start time of the jth invocation of task i.
dv_{ij} The finish time of the jth invocation of task i.

The objective is to minimize the total energy consumption of all tasks corresponding to one scheduling cycle. This then repeats itself due to the periodicity of the schedule. The start time and finish time of each task are introduced to indicate the execution order of tasks. The deadline of each task is defined as the next release time of the task. The first three time constraints guarantee the deadline of each task is met and no task gets executed before its release

time. The next two constraints make sure only one task is executed at one time. These two constraints also imply the EDF execution order [Jeffay et al. 1991]. Now we can express the multitask scheduling problem as an MILP problem. Note that only one frequency is assigned for each task in each invocation, so the MILP formulation is for interprogram DVS and voltage scaling happens between programs. We do not take into consideration the switching overhead, thus the MILP formulation actually gives the upper bound of energy savings for interprogram DVS.

We could also express the multitask voltage scheduling as an intraprogram DVS MILP problem as shown:
minimize:

$$\sum_i \sum_j \left(\sum_r \sum_m E_{irm} k_{ijrm} + \sum_r \sum_s D_{irs} S_E(\overrightarrow{k_{ijr}}, \overrightarrow{k_{ijs}}) \right)$$

subject to the following constraint:

$$\begin{aligned}
du_{ij} + \sum_r \sum_m k_{ijrm} T_{irm} + \sum_r \sum_s D_{irs} S_T(\overrightarrow{k_{ijr}}, \overrightarrow{k_{ijs}}) &\le dv_{ij} \\
dv_{ij} &\le (j+1)P_i \\
du_{ij} &\ge jP_i \\
dv_{hg} &\le du_{ij} \quad \text{if } (g+1)P_h \le (j+1)P_i \text{ and } h \ne i \\
dv_{ij} &\le du_{hg} \quad \text{if } (g+1)P_h > (j+1)P_i \text{ and } h \ne i \\
\sum_{m=1}^{N} k_{ijrm} &= 1
\end{aligned}$$

where

- P_i The period of task i.
- k_{ijrm} The mode variable for region r of task i in the jth invocation. $k_{ijrm} = 1$ if and only if the mode is set to m as a result of the scheduling, and is 0 otherwise.
- E_{irm} The energy consumption for region r of task i under mode m.
- T_{irm} The execution time of region r for task i under mode m.
- D_{irs} The transition times from region r to region s for task i.
- du_{ij} The starting time of the jth invocation of task i.
- dv_{ij} The finish time of the jth invocation of task i.

Every program is divided into regions by voltage scaling points (mode-set instructions). A region is a piece of static code between two scaling points, which can be a basic block or a function or even the whole program. The switching costs are incorporated into the formulation as overheads. Other items are similar to the interprogram formulation: the goal is to minimize the energy consumption for all tasks and the constraints guarantee meeting the deadlines. The optimality of intraprogram DVS over interprogram DVS in term of energy savings can be easily proved because interprogram MILP formulation is only a subcase of

intraprogram formulation by forcing all regions within a program to have the same voltage/frequency. In summary, compile-time intraprogram DVS is optimal compared to interprogram DVS if we assume non-preemptive scheduling for periodic tasks with fixed execution time.

If the execution time is not fixed but the execution time for each invocation is known, it is similar to the fixed execution time case and can be handled in a similar way.

9.2.2 *Variable Execution Time.* If the execution time of the tasks is not known at compile time, scheduling policies will make an estimation of the execution time. The worst-case execution time (WCET) is usually used for this. While using WCET guarantees the deadline requirements, the static scheduling is not optimal in terms of reducing energy by utilizing the slacks produced by early completion. Dynamic scheduling policies can take advantage of these slacks and thus reduce more energy. As a static scheduling technique, compile-time intraprogram DVS is not optimal. However, it is possible to apply compile-time intraprogram DVS on top of a dynamic DVS scheduling. As illustrated in the analytical model in Section 5, intraprogram DVS can slow down the memory-bounded regions without increasing the execution time. If the memory-bounded regions do not change in each invocation, we can apply intraprogram DVS to slow down those regions. The slowdown will not affect DVS decisions made by dynamic scheduling, so compile-time intraprogram DVS will contribute extra energy savings to energy savings from dynamic interprogram DVS.

The MILP formulation for this subcase is the same as in Section 6 except for deadlines. Deadlines are substituted by the actual execution times and then the MILP solver will identify those memory-bounded regions and assign the lowest frequency to those regions.

9.3 Preemptive Scheduling on Periodic Tasks

Preemptive scheduling allows low-priority tasks to be preempted, so different parts of the program can run at different frequencies. It behaves somewhat "intraprogram" DVS and usually obtains more energy savings than non-preemptive scheduling. We will discuss the applicability of intraprogram DVS for two subcases: fixed execution time and variable execution time.

9.3.1 *Fixed Execution Time.* If the execution time of each task is fixed in each invocation, for preemptive-scheduling-based EDF scheduling, the task set is schedulable when the following necessary and sufficient condition is satisfied [Liu and Layland 1973]:

$$\sum_i \frac{C_i}{P_i} \leq 1$$

where C_i is the execution time of task i and P_i is the period of task i. Once the condition is satisfied, the deadline requirements will be met.

The MILP formulation of this case is given below:
minimize

$$\sum_{\mathrm{t}} \left(\sum_{r} \sum_{m} k_{trm} E_{trm} + \sum_{r} \sum_{s} D_{trs} S_E \right)$$

subject to the following constraints:

$$\sum_{\mathrm{t}} 1/P_{\mathrm{t}} \left(\sum_{r} \sum_{m} k_{trm} T_{trm} + \sum_{r} \sum_{s} D_{trs} S_T \right) \leq 1$$

$$\sum_{m} k_{trm} = 1$$

where

k_{trm} The mode variable in mode m of region r for task t.
E_{trm} The energy consumption of region r in mode m for task t.
T_{trm} The execution time of region r in mode m for task t.
D_{trs} The number of transition times from region r to region s for task t.

Compared to the non-preemptive case, the formulation is much simpler. The objective function is to minimize the total energy consumption of all tasks. We only need to guarantee that the necessary and sufficient condition for schedulability is met. The deadline requirements for each task in each invocation are met automatically once we use the EDF scheduling policy.

Here compile-time intraprogram DVS is more efficient than interprogram DVS for two reasons:

(1) Intraprogram will take advantage of memory-bounded regions.
(2) Interprogram DVS scheduling is a subcase of intraprogram DVS.

9.3.2 *Variable Execution Time.* If the execution time of each task varies in each invocation, we can apply the same scheme as the non-preemption case: if memory-bounded regions do not change in each invocation, intraprogram will be used on top of interprogram DVS and set the mode setting for memory-bounded regions only.

9.4 Summary for Multitask Environments

In this section, we discussed cases where compile-time intraprogram DVS can be applied alone to reduce energy consumption and cases where intraprogram DVS can be used on top of interprogram DVS to get extra energy savings. For all cases, MILP can be used as the optimization tool to find out optimal voltage scaling values.

When everything (the task set, the actual execution time and the invocation time of each task) is known a priori, compile-time intraprogram DVS alone can achieve more energy savings than interprogram DVS. This includes

two subcases:

—non-preemptive scheduling on periodic tasks with fixed execution time;
—preemptive scheduling on periodic tasks with fixed execution time.

When some information (the release time or the actual execution time) is unknown at compile-time, intraprogram DVS, if possible, will slow down memory-bounded regions to get extra energy savings. Two subcases belong to this category:

—sporadic tasks with stable memory-bounded regions;
—periodic tasks with variable execution time and stable memory regions.

In summary, interprogram DVS is efficient in dynamic multitasking environments while intraprogram DVS is useful in static multitasking environments and certain dynamic environments.

10. DISCUSSION

This paper examines the opportunities and limits of DVS scaling through detailed modeling for analysis, and exact mathematical optimization formulations for compiler optimization. While this study offers useful insight into, and techniques for compiler-optimized DVS, there are subtleties and avenues for future work that we will touch on briefly here.

In the analytical model we ignore delay and energy penalties for DVS. This was required because it was not possible to a priori predict how many times, and between what voltages, the switches will happen. This potentially made the model optimistic in terms of achievable energy savings. It remains open to see if we can extend the model to account for these costs.

The second optimistic assumption of fine-grain control on the level of granularity of control for DVS mode setting, while optimistic, is not practical. We cannot insert mode-setting instructions dynamically. However, we can potentially insert mode-setting instructions for every instruction, and that represents reasonably fine grain control.

On the optimization side, a key issue in the formulation of the problem concerns which code locations are available for inserting mode-set instructions. While our early work focused on methods that considered possible mode sets at the beginning of each basic block, we feel that edges are more general because MILP solutions may assign a different frequency to a basic block depending on the entry path into it. On the other hand, this generalization will warrant certain code optimizations when actually implemented in a compiler. First, annotating execution on an edge would, if implemented naively, add an extra branch instruction to each edge because one would need to branch to the mode-set instruction and then branch onward to the original branch destination. Clearly, optimizations to hoist or coalesce mode-set instructions to avoid extra branches can potentially improve performance.

11. SUMMARY

This paper seeks to address the basic questions regarding the opportunities and limits for compile-time mode settings for DVS. When and where (if ever) is this useful? What are the limits to the achievable power savings?

We start by providing a detail analytical model that helps determine the achievable power savings in terms of simple program parameters, the memory speed, and the number of available voltage levels. This model helps point to scenarios, in terms of these parameters, for which we can expect to see significant energy savings, and scenarios for which we cannot. One important result of this modeling is that as the number of available voltage levels increase, the energy savings obtained decrease significantly. If we expect future processors to offer fine-grain DVS settings, then compile-time intraprogram DVS settings will not yield significant benefit and thus will not be worth it.

For the scenarios where compile-time DVS is likely to yield energy savings—few voltage settings, lax program deadlines, memory-bound computation—selecting the locations and values of mode settings is nonobvious. Here, we show how an extension of the existing MILP formulation for this can handle fine-grain mode settings, use accurate energy penalties for mode switches and deal with multiple input data categories. Through careful filtering of independent locations for mode-setting instructions, we show how this optimization can be done with acceptable solution times. Finally, we apply this to show how the available savings can be achieved in practice.

REFERENCES

AbouGhazaleh, N., Childers, B., Mosse, D., Melhem, R., and Craven, M. 2003. Energy management for real-time embedded applications with compiler support languages. In *Proceedings of the Languages, Compilers, and Tools for Embedded Systems(LCTES'03)*.

Advanced Micro Devices Corporation. 2002. AMD-K6 Processor Mobile Tech Docs. Available at `http://www.amd.com`.

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*.

Burd, T. and Brodersen, R. 2000. Design issues for dynamic voltage scaling. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED-00)*.

Burger, D., Austin, T. M., and Bennett, S. 1996. *Evaluating Future Microprocessors: The SimpleScalar Tool Set*. Tech. rep. TR-1308. University of Wisconsin-Madison, Computer Sciences Department.

Clark, L. 2001. Circuit design of Xscale™ microprocessors. In *2001 Symposium on VLSI Circuits, Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits*.

Flautner, K., Reinhardt, S., and Mudge, T. 2001. Automatic performance setting for dynamic voltage scaling. In *Mobile Computing and Networking*. 260–271.

Fourer, R., Gay, D., and Kernighan, B. 1993. *AMPL: A Modeling Language for Mathematical Programming*. Boyd and Fraser Publishing Company, Danvers, MA.

Ghiasi, S., Casmira, J., and Grunwald, D. 2000. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design*.

Hsu, C. and Kremer, U. 2002. Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches. In *Proceedings of Workshop on Power-Aware Computer Systems (PACS'02)*.

Hsu, C. and Kremer, U. 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, Design, and Implementation (PLDI'03)*.

HUGHES, C., SRINIVASAN, J., AND ADVE, S. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*.

ILOG CPLEX. 2002. Web Page for ILOG CPLEX Mathematical Programming Software. Available from `http://ilog.com/products/cplex/`.

IM, C., KIM, H., AND HA, S. 2001. Dynamic voltage scheduling technique for low-power multimedia applications using buffers. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*. 34–39.

INTEL CORP. 2003a. Intel® SA-1110 Processor Developer's Manual. Available at `http://developer.intel.com/design/strong/`.

INTEL CORP. 2003b. Intel Xscale™ Core Developer's Manual. Available at `http://developer.intel.com/design/intelxscale/`.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design (ISLPED-98)*. 197–202.

IYER, A. AND MARCULESCU, D. 2002. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*. 379–386.

JEFFAY, K., STANAT, D. F., AND MARTEL, C. U. 1991. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th IEEE Symposium on Real-Time Systems*. 129–139.

JEJURIKAR, R. AND GUPTA, R. 2002. Energy aware task scheduling with task synchronization for embedded real time systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 164–169.

JOSEPH, R. AND MARTONOSI, M. 2001. Run-time power estimation in high-performance microprocessors. In *International Symposium on Low Power Electronics and Design (ISLPED)*.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the 30th International Symposium on Microarchitecture*.

LEE, S. AND SAKURAI, T. 2000. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation (DAC'00)*.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM 20*, 1, 46–61.

LORCH, J. AND SMITH, A. 2001. Improving dynamic voltage algorithms with PACE. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2001)*.

LUO, J. AND JHA, N. K. 2003. Power-profile driven variable voltage scaling for heterogeneous distributed real-time embedded systems. In *International Conference on VLSI Design*.

MAGKLIS, G., SCOTT, M., SEMERARO, G., ALBONESI, D., AND DROPSHO, S. 2003. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of The 30th Annual International Symposium on Computer Architecture (ISCA-30)*.

MARCULESCU, D. 2000. On the use of microarchitecture-driven dynamic voltage scaling. In *Workshop on Complexity-Effective Design*.

MEDIABENCH II. 2003. Web page for MediaBench II. Available at `http://cares.icsl.ucla.edu/MediaBenchII/`.

MPEGTV. 1998. Mpeg Video Test Bitstreams. Available at `http://www.mpeg.org/MPEG/video.html`.

PILLAI, P. AND SHIN, K. G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*.

QU, G. 2001. What is the limit of energy saving by dynamic voltage scaling? In *Proceedings of the International Conference on Computer Aided Design*.

SAKURAI, T. AND NEWTON, A. 1990. Alpha-power model, and its application to CMOS inverter delay and other formulas. *IEEE J. Solid-State Circ. 25*, 584–594.

SAPUTRA, H., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M., HU, J., HSU, C.-H., AND KREMER, U. 2002. Energy-conscious compilation based on voltage scaling. In *Joint Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*.

SEMICONDUCTOR INDUSTRY ASSOCIATION. 2001. International Technology Roadmap for Semiconductors. Available at `http://public.itrs.net/Files/2001ITRS/Home.htm`.

SHIN, D., KIM, J., AND LEE, S. 2001. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Des. Test Comput. 18*, 2 (March/April), 20–30.

SINHA, A. AND CHANDRAKASAN, A. 2001. Dynamic voltage scheduling using adpative filtering of workload traces. In *Proceedings of the 14th International Conference on VLSI Design*.

SWAMINATHAN, V. AND CHAKRABARTY, K. 2001. Investigating the effect of voltage switching on low-energy task scheduling in hard real-time systems. In *Asia South Pacific Design Automation Conference (ASP-DAC'01)*.

SWAMINATHAN, V., SCHWEIZER, C., CHAKRABARTY, K., AND PATEL, A. 2002. Experiences in implementing an energy-driven task scheduler in rt-linux. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*. 229.

WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. 1994. Scheduling for reduced CPU energy. In *the 1st Symposium on Operating Systems Design and Implementation (OSDI-94)*. 13–23.

XIE, F., MARTONOSI, M., AND MALIK, S. 2003. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, Design, and Implementation (PLDI'03)*.

ZHANG, Y., HU, X., AND CHEN, D. 2003. Energy minimization of real-time tasks on variable voltage processors with transition energy overhead. In *Proceedings of the ASP-DAC 2003 Design Automation Conference*. 65–70.

Received September 2003; revised December 2003, April 2004, June 2004; accepted June 2004