Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits

Fen Xie
Department of Electrical Engineering
Princeton University
Princeton, NJ
fxie@ee.princeton.edu

Margaret Martonosi
Department of Electrical Engineering
Princeton University
Princeton, NJ
mrm@ee.princeton.edu

Sharad Malik
Department of Electrical Engineering
Princeton University
Princeton, NJ
sharad@ee.princeton.edu

ABSTRACT

With power-related concerns becoming dominant aspects of hardware and software design, significant research effort has been devoted towards system power minimization. Among run-time power-management techniques, dynamic voltage scaling (DVS) has emerged as an important approach, with the ability to provide significant power savings. DVS exploits the ability to control the power consumption by varying a processor's supply voltage (V) and clock frequency (f). DVS controls energy by scheduling different parts of the computation to different (V, f) pairs; the goal is to minimize energy while meeting performance needs. Although processors like the Intel XScale and Transmeta Crusoe allow software DVS control, such control has thus far largely been used at the process/task level under operating system control. This is mainly because the energy and time overhead for switching DVS modes is considered too large and difficult to manage within a single program.

In this paper we explore the opportunities and limits of compile-time DVS scheduling. We derive an analytical model for the maximum energy savings that can be obtained using DVS given a few known program and processor parameters. We use this model to determine scenarios where energy consumption benefits from compile-time DVS and those where there is no benefit. The model helps us extrapolate the benefits of compile-time DVS into the future as processor parameters change. We then examine how much of these predicted benefits can actually be achieved through optimal settings of DVS modes. This is done by extending the existing Mixed-integer Linear Program (MILP) formulation for this 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 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 Modelling—*Model Validation and Analysis*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'03, June 9–11, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-662-5/03/0006 ...$5.00.

General Terms

Design, Experimentation

Keywords

Low Power, Compiler, Dynamic Voltage Scaling, Mixed-integer Linear Programming, Analytical Model

1. INTRODUCTION

The International Technology Roadmap for Semiconductors highlights system power consumption as a limiting factor in our ability to develop designs below the 50nm technology point [26]. 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.

Dynamic voltage and frequency scaling (DVS) is a technique that allows the system to explicitly trade off performance for energy savings, by providing a range of voltage and frequency operating points. DVS allows one to reduce the supply voltage at run time to reduce power/energy consumption. However, reducing the voltage (V) increases the device delay and so must be accompanied by a reduction in clock frequency (f). Thus, the voltage and frequency must be varied together. Proposals have been made for purely-hardware DVS [21] as well as for schemes that allow DVS with software control [7, 14, 12]. 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 [14], StrongArm SA-2 and AMD mobile K6 Plus [1]. 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 since 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 [10]. 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.* [25] provides an exact mixed-integer linear programming (MILP) technique that can determine the appropriate (V,f) setting for each 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 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 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. Towards 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 analytical model and analysis in Section 3. Section 4 derives the MILP formulation used to determine the values of the optimal mode setting instructions. Section 5 discusses some implementation details for our MILP-based approach, and Section 6 provides the results of various experiments. Critical and unresolved issues are the focus of Section 7. Finally, we present some concluding remarks in Section 8.

2. RELATED WORK

DVS scheduling policies have been studied exhaustively at the operating system, micro-architecture and compiler levels. Algorithms at the OS level using heuristic scheduling include an interval-based algorithm like Lorch and Smith's proposal [19] and a task-based algorithm like Luo and Jha's work [20]. Integer Linear Programming (ILP) based scheduling has also been used in algorithms at the OS level. For example, Ishihara and Yasuura [15] give an ILP formulation that does not take into account the transition costs. Swaminathan and Chakrabarty [28] incorporate the transition costs into the ILP formulation but make some simplifications and approximations in order to make the formulation linear. At the micro-architecture level, Ghiasi [9] suggests the use of IPC (instructions per cycle) to direct DVS, and Marculescu [21] proposes the use of cache misses to direct DVS. Both are done through hardware support at run time.

Some research efforts have targeted the use of mode-set instructions at compile time. Mode-set instructions are inserted either evenly at regular intervals in the program like Lee and Sakurai's work [18], or on a limited number of control flow edges as proposed by Shin [27]. In the latter, the mode value is set using worst-case execution time analysis for each basic block. Hsu and Kremer [10] suggest lowering voltage/frequency in memory-bound regions using power-down instructions and provide a non-linear optimization formulation for optimal scheduling. Saputra *et al.* [25] 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 [11] have introduced a simple model to estimate theoretical bounds of energy reduction any DVS algorithm can produce. In [15], a simple ideal model which 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 [23]. 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 DVS.

In this paper, we present a realistic analytical model incorporating features of both real program behavior and compile time DVS scheduling. 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 FROM COMPILE-TIME 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 intra-program DVS scheduling.

There has been some research on potential energy savings for DVS scheduling. Analytical models have been studied in [15] and [23]. However, that research only models computation operations and not memory operations, thus not capturing the 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 overcome the restrictions of prior modeling efforts.

In deriving this model we make the following assumptions about the program, micro-architecture 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) [24].
5. Computation can be assigned to different frequencies at an arbitrarily fine grain, i.e. a continuous partitioning of the computation and its assignment to different voltages is possible.
6. There is no energy or delay penalty associated with switching between different (V,f) pairs.

While the first 4 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. Of course, the optimism should not result in the bounds being too loose and thus useless in practice. We will review the tightness of the bounds in Section 6.

3.2 Basic Model

The existence of memory operations complicates the analysis. Cache misses provide an opportunity for intra-program DVS, since execution time will not change with voltage/frequency if the memory is asynchronous with the processor. Slowing down the frequency in that region will save energy without impacting performance. However, savings are limited by the compile-time aspect of intra-program DVS. As mode-set instructions are inserted statically, it applies to all executions of a specific memory reference,

both cache misses as well as hits. It is rare to have a reference always suffer a miss. Slowing down all executions for that reference may save energy for cache misses, but will result in loss of performance for cache hits. Our model captures this effect.

For any piece of static code, we can divide it into two major operations: computation and memory operations. For computation operations, some operations depend on the result of pending memory operations, referred to as the dependent part, and some can run concurrently with memory operations, referred to as the overlap part. For memory operations, some end up being cache hits and others need to go to memory to fetch the operands. We define the following program parameters of the model based on the observation.

$N_{overlap}$ The number of execution cycles of computation operations that can run in parallel with memory operations.

$N_{dependent}$ The number of execution cycles of computation operations that are dependent on memory operations.

N_{cache} The number of execution cycles of memory operations due to cache hits.

$t_{invariant}$ 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.

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. Note that in actual program execution the memory operations and the overlapping and dependent computations will be interleaved in some way. However, we can lump all the occurrences of each category together as shown in the above figures since, as we will show, for purposes of the analysis it is sufficient to consider the total time taken by each of these categories. Also, we can consider the vertical dimension to be time, and order the three categories as shown, even though in actual execution they will be interleaved.

In Figure 1(a), parallel computation operations determine the execution time of the overlap part and total execution time is $N_{overlap}/f + N_{dependent}/f$. In Figures 1(b) and 1(c), memory operations dominate the overlap part and total execution time is $t_{invariant} + N_{cache}/f + N_{dependent}/f$. The finer distinction between Figures 1(b) and 1(c) will be made later in the analysis. The total execution time expression for any of these cases is:

$$\max(t_{invariant} + N_{cache}/f, N_{overlap}/f) + N_{dependent}/f$$

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 dependent computation operations cannot overlap with the time for the memory operations $t_{invariant} + \frac{N_{cache}}{f}$. This respects the fact that dependent calculations 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 computation finishes, t_2 be the time the dependent 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 (i) $t_3 \leq t_{deadline}$, (ii) $t_2 \geq t_{invariant} + \frac{N_{cache}}{f_1}$ and (iii) $t_1 \leq t_2$. The first integral represents energy consumption during the overlapping computation. The second integral represents energy during the dependent computation 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 constant independent of processor frequency. Unlike models proposed in [15] or [23], the presence 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.

3.3 Continuously Scalable Supply Voltage

We first consider the case where the supply voltage can be scaled continuously, i.e. 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 which adjusts the total execution time to just meet the deadline [15]. 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.

3.3.1 *Computation Dominated and Memory Dominated Cases*

The first two cases illustrated in Figure 1 arise depending on the balance of computation and memory operations. In the computation dominated case, the memory time is largely irrelevant, because meeting the deadline mainly revolves around getting the computation done in time. Because this case has no slack due to asynchrony with memory, this case is similar to the pure computation case in [15] and thus a *single frequency* leads to optimal energy performance.

In the memory dominated case, the computation is broken into two parts, the overlapped and the dependent part, each with its own time constraint. Thus, the optimal operating point arises when *two frequencies* are chosen: one for the overlapped computation, and a different one for the dependent computation. For both the computation and memory dominated cases, the first case we consider is when the overlapping computation operations take longer than cache hit memory operations, which means $N_{cache} < N_{overlap}$. This means that any frequency adjustments done in the upper part of the timeline will dilate the computation side more than the memory side. Section 3.3.2 handles the (relatively uncommon) other possibility later.

A key dividing line between the computation dominated and memory dominated cases concerns the inflection point when memory time begins to matter. We use the term $f_{invariant}$ to refer to a clock frequency at which the execution time of $N_{overlap} - N_{cache}$ cycles of computation operations just balances the cache miss service time $t_{invariant}$. (Therefore, we use $v_{invariant}$ to refer to the voltage setting paired with that frequency). If the optimal energy point can be reached with a frequency slower than $f_{invariant}$ we say that the program is computation dominated. If the speed is slower than $f_{invariant}$, then the $N_{overlap} - N_{cache}$ cycles of computation operations take up all cache miss period $t_{invariant}$ and go beyond as shown in 1(a).

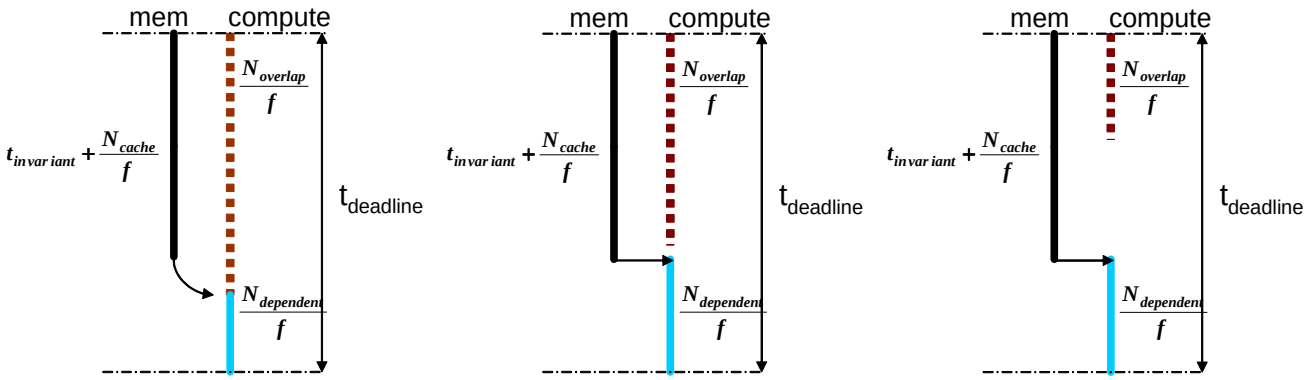
The problem becomes one of minimizing:

$$E(v_1, v_2) = N_{overlap} * v_1^2 + N_{dependent} * v_2^2$$

subject to different constraints depending on

If $f_1 > f_{invariant}$

$$\frac{N_{cache}}{f_1} + \frac{N_{dependent}}{f_2} + t_{invariant} = t_{deadline}$$



(a) Computation-Dominated Case. (b) Memory-Dominated Case. (c) Sub-Case: Memory-Dominated, with slack.

Figure 1: Possible Overlaps of Memory and Computation

If $f_1 \leq f_{invariant}$

$$\frac{N_{overlap}}{f_1} + \frac{N_{dependent}}{f_2} = t_{deadline}$$

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

If $\frac{k(v_1-v_T)^\alpha}{v_1} > f_{invariant}$

$$\frac{N_{cache}v_1}{k(v_1-v_T)^\alpha} + \frac{N_{dependent}v_2}{k(v_2-v_T)^\alpha} + t_{invariant} = t_{deadline}$$

If $\frac{k(v_1-v_T)^\alpha}{v_1} \leq f_{invariant}$

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

Due to the time constraints, 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.

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

$\frac{dv_2}{dv_1}$ is obtained from the constraint equations.

If $\frac{k(v_1-v_T)^\alpha}{v_1} > f_{invariant}$

$$\frac{dv_2}{dv_1} = -\frac{N_{cache}v_1}{N_{dependent}v_2}\frac{(1-\alpha-\frac{v_T}{v_1})}{(1-\alpha-\frac{v_T}{v_2})}\frac{(v_2-v_T)^{\alpha+1}}{(v_1-v_T)^{\alpha+1}})$$

If $\frac{k(v_1-v_T)^\alpha}{v_1} \leq f_{invariant}$

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

So, if $f_1 = \frac{k(v_1-v_T)^\alpha}{v_1} > f_{invariant}$, we get:

$$\frac{dE}{dv_1} = 2N_{overlap}v_1(1 - \frac{N_{cache}}{N_{overlap}}\frac{(1-\alpha-\frac{v_T}{v_1})}{(1-\alpha-\frac{v_T}{v_2})}\frac{(v_2-v_T)^{\alpha+1}}{(v_1-v_T)^{\alpha+1}})$$

On the other hand if $\frac{k(v_1-v_T)^\alpha}{v_1} \leq f_{invariant}$ then:

$$\frac{dE}{dv_1} = 2N_{overlap}v_1(1 - \frac{(1-\alpha-\frac{v_T}{v_1})}{(1-\alpha-\frac{v_T}{v_2})}\frac{(v_2-v_T)^{\alpha+1}}{(v_1-v_T)^{\alpha+1}})$$

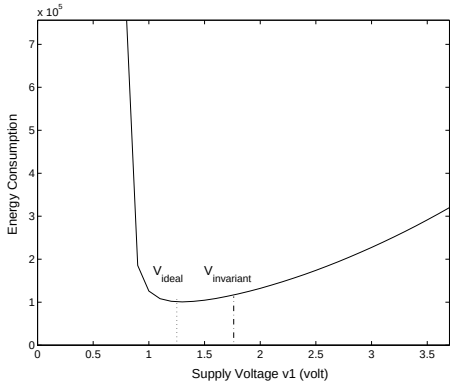


Figure 2: Computation Dominated: Energy consumption versus supply voltage (v_1) of overlapped compute/memory region.

We define $f_{ideal} = \frac{N_{overlap}+N_{dependent}}{t_{deadline}}$. The conditions for which minimum energy is achieved depend on the relationship between $f_{invariant}$ and f_{ideal}.

If $f_{invariant} \geq f_{ideal}$, then using the single frequency f_{ideal}, $t_{invariant}$ is completely filled up with computation operations as shown in Figure 1(a). It is obvious that $\frac{dE}{dv_{ideal}} = 0$. So $v_1 = v_2 = v_{ideal}$ is the required condition to minimize energy. This is consistent with the equation for the case $f_1 \leq f_{invariant}$.

Figure 2 shows the relationship between energy consumption and supply voltage v_1 for a set of parameters that satisfy these conditions (i.e. $N_{cache} < N_{overlap}, f_{invariant} \geq f_{ideal}$). The energy consumption using $v_{invariant}$ and v_{ideal} are shown. When $v_1 < v_{ideal}$ or $v_1 > v_{ideal}$, energy consumption increases as you move away from v_{ideal}. When $v_1 = v_2 = v_{ideal}$, energy is minimized. The minimum energy is $E = (N_{overlap} + N_{dependent})v_{ideal}^2$. Since a single (V,f) setting is optimal for this case, intra-program DVS will not provide energy savings here.

If $f_{invariant} < f_{ideal}$, Figure 3 shows the energy consumption with respect to different v_1. While the computation-dominated case had a single frequency setting as its optimal point, this case requires two different settings for optimality. It is hard to give a closed-form expression for the optimal settings, but we can use tools to get numerical solutions. To plot the energy relationship, we selected various values of v_1, and for each v_1 we compute the optimal value of v_2 from the relationships above. Therefore, the minimum energy point in Figure 3 shows the best v_1 choice, this

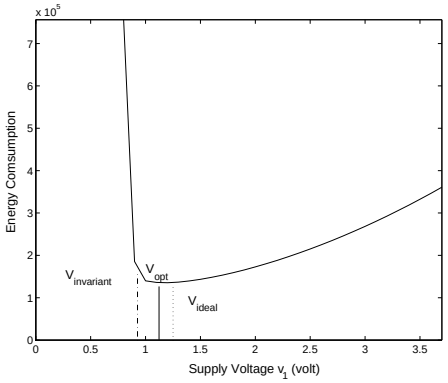


Figure 3: Memory Dominated: Energy consumption versus supply voltage (v_1) of mixed compute/memory region.

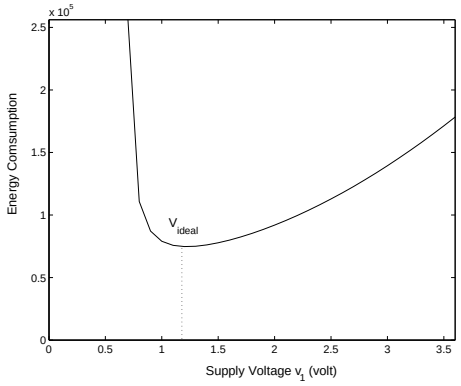


Figure 4: Memory Dominated, with slack: Energy consumption versus supply voltage (v_1) of mixed compute/memory region.

allows us to select the overall optimal v_1, v_2 pair. What we see is that the optimum point in this case is for a v_1 that is less than v_{ideal} from the computation-dominated case, and a v_2 that is greater. This corresponds to low-frequency operation while the overlapped computation is hidden by the memory latency operation, followed by high-frequency "hurry-up" execution of the dependent computation when memory finally returns.

3.3.2 Special Case: Memory-Dominated, with slack.

Thus far, we have dealt with cases where N_{cache} is relatively small compared to $N_{overlap}$. If $N_{cache} \geq N_{overlap}$, the cache hit memory operations take longer than the overlapping computation operations, and indeed any attempt to slow down the overlap execution will actually dilate the memory time by an even greater amount due to N_{cache}. This case is illustrated in Figure 1(c). This effect can be thought of as related to the fact that when we assign clock frequencies to code regions statically, we are fixing the clock frequency both for executions of the static code that result in many cache misses as well as for other executions that may result in many cache hits.

The total execution time here is $\frac{N_{cache}}{f_1} + \frac{N_{dependent}}{f_2} + t_{invariant}$. The problem then reduces to minimizing:

$$E(v_1, v_2) = N_{cache} v_1^2 + N_{dependent} v_2^2$$

subject to the following deadline constraint:

$$\frac{N_{cache} v_1}{k(v_1 - v_T)^\alpha} + \frac{N_{dependent} v_2}{k(v_2 - v_T)^\alpha} + t_{invariant} = t_{deadline}$$

(Here, we have assumed the (V,f) relationship given in Section 3.1.) From this we get:

$$\frac{dE}{dv_1} = 2N_{cache} v_1 * (1 - \frac{(1 - \alpha - \frac{v_T}{v_1})}{(1 - \alpha - \frac{v_T}{v_2})} * \frac{(v_2 - v_T)^{\alpha+1}}{(v_1 - v_T)^{\alpha+1}})$$

Let $f_{ideal} = \frac{N_{cache} + N_{dependent}}{t_{deadline} - t_{invariant}}$ and v_{ideal} the corresponding supply voltage for this case. Energy consumption satisfying the above time constraints is a convex function of v_1 as shown in Figure 4. It is easy to deduce that when $v_1 = v_2 = v_{ideal}$, $\frac{dE}{dv_1} = 0$. A single frequency f_{ideal} minimizes energy consumption.

$$E_{min} = (N_{cache} + N_{dependent}) v_{ideal}^2$$

3.3.3 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 $N_{overlap} > N_{cache}$ and $f_{ideal} > f_{invariant}$. The latter of these conditions translates to: $(N_{overlap} + N_{dependent})/t_{deadline} > (N_{overlap} - N_{cache})/t_{invariant}$. When all the parameters are consistent with the above conditions, it is possible to use multiple voltages for 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.

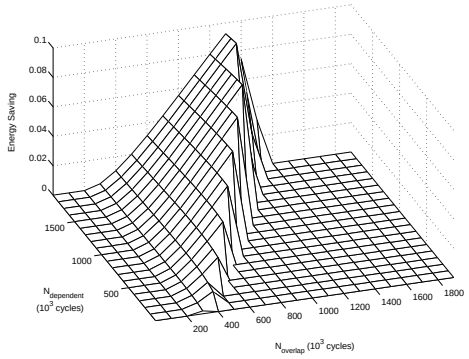


Figure 5: Continuous Case: Energy Saving Ratio with respect to different $N_{overlap}$ and $N_{dependent}$ ($N_{cache} = 3 \times 10^5 cycles, t_{deadline} = 3000\mu s, t_{invariant} = 1000\mu s$).

Energy savings is a function of $N_{overlap}$, $N_{dependent}$, N_{cache}, $t_{invariant}$ 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 $N_{dependent}$ are illustrated in Figure 5. For fixed $N_{cache}, t_{deadline}$ and $t_{invariant}$, for most cases, there is no energy savings. When $N_{overlap}$ is less than N_{cache}, computation operations will not extend over the full $t_{invariant}$ period, so one single frequency can achieve the minimum energy. This corresponds to Figure 4. 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 $t_{invariant}$ without increasing execution time. Two frequencies instead of one can then used to achieve minimum energy, as shown in Figure 3. 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 2. Thus there are again no energy savings when compared to the best static single frequency setting.

In Figure 6, $N_{overlap}, N_{dependent}$ and $t_{deadline}$ are fixed. As $t_{invariant}$ increases, energy saving increases. This is intuitive because as the memory bottleneck increases, the opportunities for

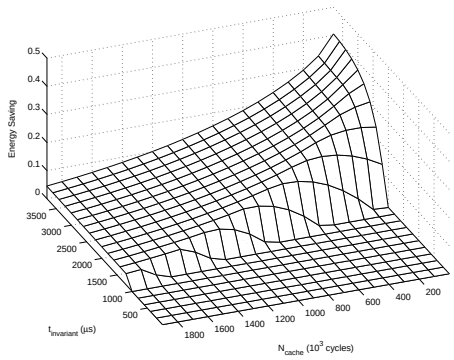


Figure 6: Continuous Case: Energy Saving Ratio with respect to different N_{cache} and $t_{invariant}$ ($N_{overlap} = 4 \times 10^6 cycles, N_{dependent} = 5.8 \times 10^6 cycles, t_{deadline} = 5000\mu s$).

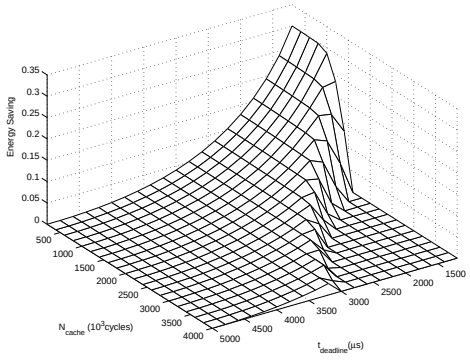


Figure 7: Continuous Case: Energy Saving Ratio with respect to different $t_{deadline}$ and N_{cache} ($N_{overlap} = 4 \times 10^6 cycles, N_{dependent} = 5.7 \times 10^6 cycles, t_{invariant} = 1000\mu s$).

voltage scaling due to overlap slack increase. Usually when $N_{cache} = 0$, energy saving is maximized. This is because when all memory operations are cache miss memory operations, slowing down the overlap computations does not dilate the memory timeline, and thus does not impede the start of the dependent operations.

Figure 7 shows energy savings with respect to different $t_{deadline}$ and N_{cache}. Other parameters are fixed. When N_{cache} 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_{cache} gets bigger, however, energy savings go up, achieve a maximum point and then go down again. This is because as N_{cache} increases, the slowdown over the $t_{invariant}$ has more impact on the execution time. So it is less likely two frequencies will reduce energy.

3.4 Scaling Voltage with Discrete Settings

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 $(V_L, V_{L2}...V_h)$. 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:

$$\sum_i^h \frac{(x_{1i} + x_{2i})}{F_i} \quad \leq \quad t_{deadline}$$

Here x_{1i} and x_{2i} are the number of cycles at voltage level i for the two parts of the computation respectively. The constraint above is just the minimum constraint. Other constraints will depend on the values of certain parameters and will be added on a case by case basis.

Leveraging off prior work by [15] 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 and memory bound with slack cases, both of 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 3.3. The memory bounded with slack case is similar and will not be discussed here. Consider the two neighboring values for voltage and frequency: $v_a < v_{ideal} < v_b$ and $f_a < f_{ideal} < f_b$. 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 x_b are determined by solving for the following constraints:

$$\begin{aligned} x_a/f_a + x_b/f_b &= t_{deadline} \\ x_a + x_b &= N_{overlap} + N_{dependent} \end{aligned}$$

Consider next the memory-dominated case that resulted in two frequencies in the continuous-scaling approach. We cannot use the two frequencies from the continuous case to find discrete frequencies that minimize the energy as we did for the single frequency case. Instead, this needs a fresh analysis approach.

Let variable y represent the execution time for N_{cache}. Then $\frac{N_{cache}}{y}$ must be less than $f_{invariant}$ to stay in the memory-dominated case. Since y is a deadline for N_{cache}, we know that $f_1 = \frac{N_{cache}}{y}$ is the optimal frequency for this code in the continuous case. Similarly, $f_2 = \frac{N_{dependent}}{t_{deadline} - t_{invariant} - y}$ is the optimal frequency for $N_{dependent}$. This gives us: f_a, f_b the immediate discrete neighbors of f_1 and f_c, f_d the immediate discrete neighbors of f_2, 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:

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

Note that thus far, all the frequencies and cycle counts determined are a function of y, i.e. they depend on the value of y selected. We can also express the minimum energy as a function of y. We run as many execution cycles as possible from $N_{overlap} - N_{cache}$ at the lower frequency f_a and the remaining (if any) at frequency f_b.

$$E_{min}(y) = v_a(y)^2 x_a + v_b(y)^2 x_b + v_c(y)^2 x_c + v_d(y)^2 x_d + \frac{t_{invariant}}{y} x_a v_a(y)^2 + \max((N_{overlap} - N_{cache} - \frac{t_{invariant} x_a}{y}) v_b(y)^2, 0)$$

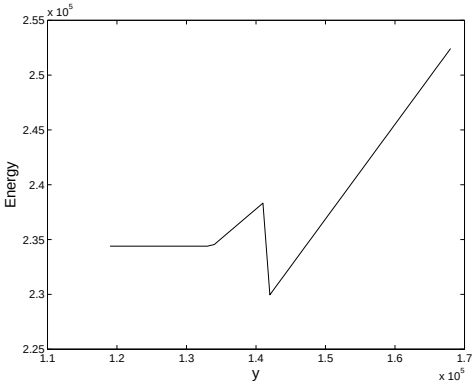


Figure 8: Discrete case: energy consumption versus the execution time of N_{cache}, i.e. y.

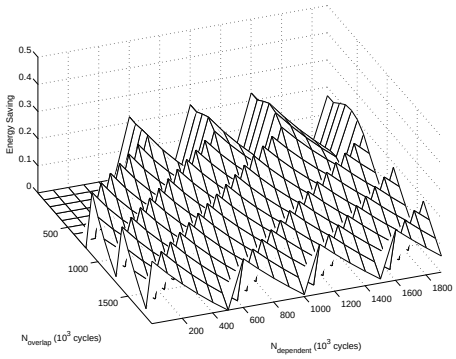


Figure 9: Discrete Case: Energy savings with respect to different $N_{overlap}$ and $N_{dependent}$ relative to best single-frequency setting that meets the deadline. (7 voltage levels, $N_{cache} = 2 \times 10^5 cycles, t_{deadline} = 5200\mu s, t_{invariant} = 1000\mu s$).

$$E_{min}(y) = \frac{f_a(y)f_b(y)}{f_b(y)-f_a(y)}[(\frac{N_{cache}}{f_a(y)} - y)v_b(y)^2 + (1 + \frac{t_{invariant}}{y})(y - \frac{N_{cache}}{f_b(y)})v_a(y)^2] + \frac{f_c(y)f_d(y)}{f_d(y)-f_c(y)}[(\frac{N_{dependent}}{f_c(y)} - (t_{deadline} - t_{invariant} - y))v_d(y)^2 + (t_{deadline} - t_{invariant} - y - \frac{N_{dependent}+y}{f_d(y)})v_c(y)^2] + \max((N_{overlap} - N_{cache} - \frac{t_{invariant}}{y}\frac{f_a(y)f_b(y)}{f_b(y)-f_a(y)}(y - \frac{N_{cache}}{f_b(y)}))v_b(y)^2, 0)$$

$f_a(y), f_b(y), f_c(y), f_d(y)$ are staircase functions of y. It is hard to determine analytically for what value of y is $E_{min}(y)$ minimized. However, we can do this numerically for a specific instance. Figure 8 shows how $E_{min}(y)$ changes with y for a particular case, which enables us to select the y for which energy is minimized.

3.4.1 Discrete Voltage Settings: Summary and Results

The main results in this case are that for the compute bound and memory bound with slack cases, we can use the two voltages from the available set that are nearest neighbors of the single optimal voltage in the continuous case. For the memory bound case, four voltages are needed, and can be determined using the techniques described.

We now examine the surfaces for the energy savings obtained in terms of the dependent parameters in Figures 11, 9 and 10. The figures do a good job of conveying the complexity of the energy optimization space when discrete voltage settings are involved. Benefits of intra-program 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.

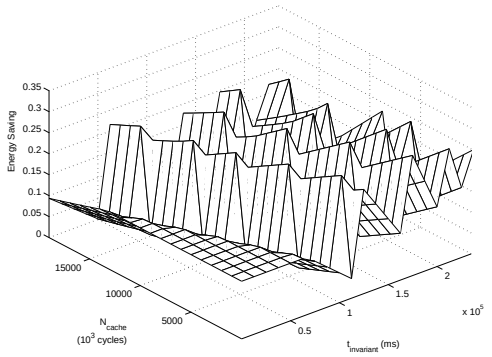


Figure 10: Discrete Voltage Levels: Energy savings for different N_{cache} and $t_{invariant}$ relative to best single-frequency setting that meets the deadline. (7 discrete voltage levels, $N_{overlap} = 1.3 \times 10^7 cycles, N_{dependent} = 7 \times 10^7 cycles, t_{deadline} = 3.5 \times 10^5 \mu s$).

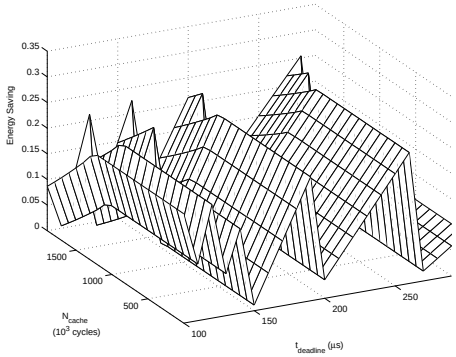


Figure 11: Discrete Case: Energy savings for different $t_{deadline}$ and N_{cache} relative to best single-frequency setting that meets the deadline. This graph is plotted for a case with seven possible discrete voltage levels. ($t_{deadline} = 1340\mu s, N_{overlap} = 1.3 \times 10^7 cycles, N_{dependent} = 7 \times 10^7 cycles$).

When more voltage settings are available, the number of peaks in the graphs increases. When the stepsize 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; intra-program readjustments are of lesser value if the initial setting can be done with sufficient precision.

To better understand energy trends in the discrete voltage case, we study here a set of benchmarks considering situations with 3, 7 or 13 available voltage levels. In all experiments, $\alpha = 1.5, v_t = 0.45V$, and we considered 5 different deadlines as elaborated on in Section 6. By using cycle-level CPU simulation to get program parameters N_{cache}, $N_{overlap}$,$N_{dependent}$ and $t_{invariant}$, we can plug the values into the analytic models generated in this section. The maximum energy savings predicted by the models is illustrated in Table 1.

Interestingly, energy savings is not monotonic with deadline because we compare not to the highest-frequency operation, but to

Benchmark	Voltages levels	Deadline1	Deadline2	Deadline3	Deadline4	Deadline5
adpcm	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	0.62	0.33	0.04	0.31	0.09
	7	0.22	0.23	0.14	0.14	0.12
	13	0.10	0.12	0.03	0.04	0.13
gsm	3	0.60	0.37	0.10	0.33	0.12
	7	0.21	0.02	0.03	0.16	0.15
	13	0.10	0.03	0.05	0.06	0.05
mpeg/decode	3	0.66	0.38	0.03	0.26	0.07
	7	0.26	0.03	0.10	0.10	0.09
	13	0.14	0.03	0.12	0.11	0.10

Table 1: Analytical Results of Energy Saving Ratio for different voltage levels

the best single frequency that meets the deadline. Nonetheless, lax deadlines (e.g., Deadline 1) and few voltage levels offer the best scenario for compile-time DVS. 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, intra-program DVS seems to require a fairly intelligent optimization process. Our MILP-based proposal for managing this optimization is presented in the next section.

4. PRACTICAL ENERGY SAVINGS USING MATHEMATICAL OPTIMIZATION

Section 3 provides a detailed analysis for the maximum energy savings possible using profile-based intra-program DVS. As it uses 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.

4.1 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 [14]. 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 Mixed-Integer Linear Programming (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.* [25] 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 in Figure 12, 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.

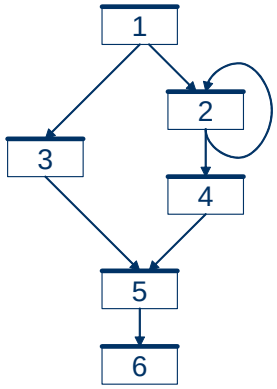


Figure 12: An Example Control Flow Graph

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 13 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. Section 4.2 discusses our methodology further, and Section 4.3 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 [19]; such continuous (V,f) scaling is less commercially-feasible.) Figure 13 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.

4.2 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

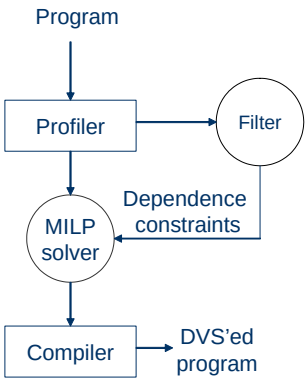


Figure 13: Flow Diagram of the Technique

from v_i to v_j.

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

Equations for S_E, S_T have been taken from [4], 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.

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}, $m = 1, 2, ...N$. $k_{ijm} = 1$ 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 can be set to at most one mode, 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:

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

In the relationships above, R is the number of regions, i.e., nodes such as basic blocks in a control-flow graph. N is the number of mode settings, k_{ijm} is the mode variable for mode m on edge (i,j) and $\overrightarrow{k_{ij}}$ is the set of mode variables (N in all) for edge (i,j). E_{jm} is the energy consumption for a single invocation of region j under mode m. G_{ij} is the number of times region j is entered through edge (i,j) and D_{hij} is the number of times region i is entered through edge (h,i) and exited through edge (i,j). T_{jm} is the execution time for a single invocation of region j under mode m. These last four values are all constants 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 $S_E(\overrightarrow{k_{hi}}, \overrightarrow{k_{ij}}) = c * (1-u) |\sum_{m=1}^{N} k_{him} V_m^2 - \sum_{m=1}^{N} k_{ijm} V_m^2|$. Likewise, S_T, the transition time cost for one mode transition, is represented as: $S_T(\overrightarrow{k_{hi}}, \overrightarrow{k_{ij}}) = \frac{2*c}{I_{MAX}} |\sum_{m=1}^{N} k_{him} V_m - \sum_{m=1}^{N} k_{ijm} V_m|$.

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 non-linear 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

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

subject to the constraints:

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

$$\sum_{m} k_{ijm} = 1$$
$$-e_{hij} \leq \sum_{m} (k_{him} V_m^2 - k_{ijm} V_m^2) \leq e_{hij}$$
$$-t_{hij} \leq \sum_{m} (k_{him} V_m - k_{ijm} V_m) \leq t_{hij}$$

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 $C_E = c * (1-u)$, $C_T = \frac{2*c}{I_{MAX}}$ 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 non-zero 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 4.1, 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 runtime. 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.

4.3 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:

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

In these relations, p_g is the possibility of input category g as input. E_{jmg} is the energy consumption of region j in mode m for input data in category g and likewise T_{jmg} is the execution time of region j in mode m for input data in category g. G_{ijg} is the number of times region j is entered through edge (i, j) for input data in category g and the path counter D_{hijg} refers to 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. The following section describes our actual MILP-based implementation in further detail before we present our energy results.

5. DVS IMPLEMENTATION USING PROFILE-DRIVEN MILP

As shown in Figure 13, 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.

5.1 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 [16].

The data shown here have been gathered using the Wattch power/-performance simulator [3], which is based on SimpleScalar [5]. 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 2. We assume that the CPU has three scaling levels for (V,f). They are a frequency of 200MHz paired with a supply voltage of 0.7V, 600MHz at 1.3V, and a maximum performance setting of 800MHz at 1.65V. This is similar to some of the voltage-frequency pairings available in Intel's XScale processors [6].

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, 521K, 4-way(LRU)
	32B blocks, 16-cycle latency
TLBs	32-entry, 4096-byte page

Table 2: Configuration parameters for CPU simulation.

5.2 Filtering Edges to Reduce MILP Solution Time

While our MILP approach generally works in practice for even large programs, its runtime can be reduced by filtering the edges that are considered as candidates for mode-set instructions. As discussed in Section 4.2, 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. 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).

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.

5.3 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 [8] to express the mathematical constraints and to enable pruning and optimizations before feeding the MILP problem into the CPLEX solver [13].

As shown in Figure 14, 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 six of the MediaBench applications [17], with a transition time of 12 μs, and transition energy of 1.2μJ.)

Table 3 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 4.2, the deadlines will still be met exactly, even with the filtering in place.

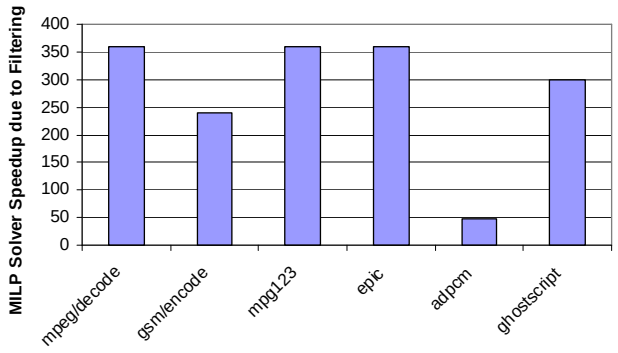


Figure 14: Speedup in MILP solution time when edge filtering is applied.

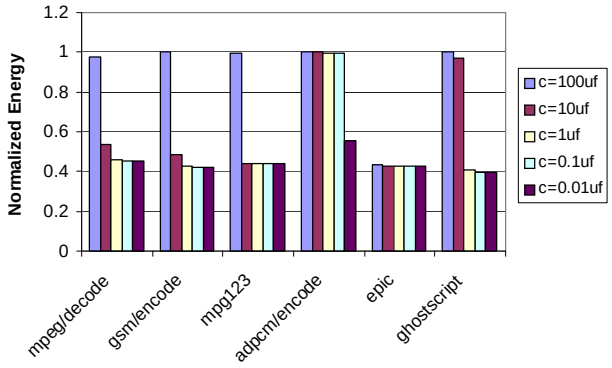


Figure 15: Impact of transition cost. Energy is normalized to minimum energy without transition.

benchmark	All:Energy	Subset:Energy
mpeg	122392.8 μJ	122392.8 μJ
gsm	72287.6 μJ	72287.6 μJ
mpg123	37291.4 μJ	37291.4 μJ
adpcm	10194.3 μJ	10195.4 μJ
epic	33021.9 μJ	33021.9 μJ
ghostscript	357.3μJ	357.3μJ

Table 3: Energy consumption when the MILP solver is run on the full set of program edges (left) or the filtered subset of transition edges (right).

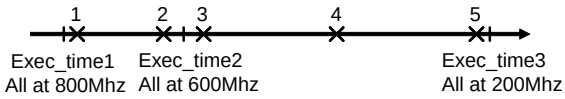


Figure 16: Positions of deadlines

6. EXPERIMENTAL RESULTS

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

6.1 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 [17] except for mpg123. Unless otherwise specified, we use the inputs provided with the suite, and we run the programs to completion.

6.2 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 4.2, 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 10μf yields 12μs transition time and 1.2μJ transition energy cost for a transition from 600MHz/1.3V to 200MHz/0.7V. This 12μs transition time corresponds well to published data for XScale [14]. We used a wide range of c from 100μ to 0.01μf 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 16. This is given, for each benchmark, by the time in the "Deadline 5" column of Table 4. This range of deadlines will be discussed shortly in more detail when the examine the impact of different deadlines on energy savings.

Results for six Mediabench benchmarks are shown in Figure 15. For each benchmark, the energy is normalized to that program running at a fixed 600MHz clock rate. This clock rate is sufficient to meet the deadline, so for very high transition costs (c = 100μf), 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 200MHz. 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). If there were no transition energy costs at all, the maximum energy saving would be bounded by the ratio of the $V^2 f$ values, or $0.7^2/1.3^2$ which equals 0.29. For the smallest possible c values, one can see that we approach this value, since transition costs are quite small.

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

Benchmark	Exec time at 200 MHz	Exec time at 600 MHz	Exec time at 800 MHz	Deadline 5	Deadline 4	Deadline 3	Deadline 2	Deadline 1
adpcm/encode	29.5	9.9	7.4	29.0	20.0	10.0	8.1	7.6
mpeg/decode	557.6	187.3	141.0	557.6	300.0	190.0	181.0	151.0
gsm/encode	334.0	111.4	83.6	333.0	220.0	120.0	100.0	90.0
epic	152.6	53.6	41.0	150.0	100.0	60.0	50.0	45.0
ghostscript	2.0	0.89	0.74	2.0	1.5	1.0	0.81	0.76
mpg123	177.7	59.2	44.4	177.6	100.0	60.0	58.0	45.0

Table 4: Deadline boundaries and chosen deadlines for benchmarks (ms)

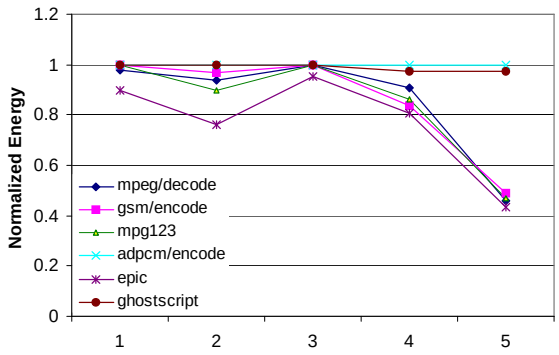


Figure 17: Impact of deadline on energy. Energy is normalized to the energy of the best of the three possible single frequency settings.

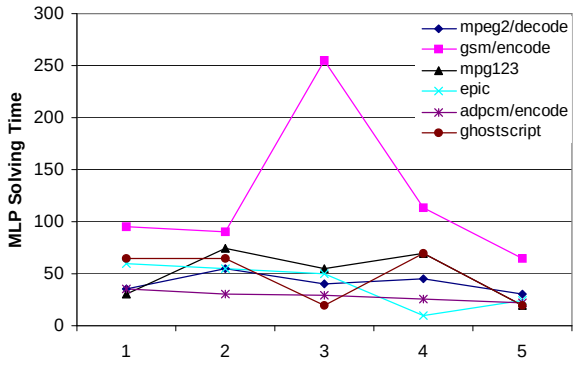


Figure 18: MILP solution time (in seconds) for different deadlines.

number of transitions between them.

To make this more concrete for the benchmarks we consider, Table 4 includes the runtimes of each benchmark when operating purely at 800MHz, 600MHz, or 200MHz without any transitions. To test MILP-based DVS on each benchmark, we choose 5 application-specific deadlines per benchmark that are intended to exercise different points along the possible range. These chosen deadlines are also given in Table 4. The results here are shown for a "typical" transition cost of $c = 10\mu$f.

Figure 17 shows the optimized energies for these experiments. 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.

As shown in Figure 18, 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

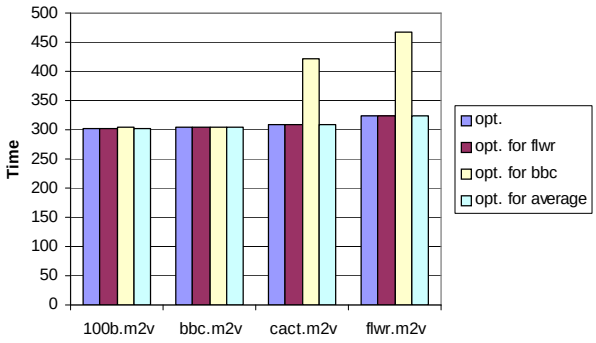


Figure 19: Dependence of program runtime on input data used for MILP profiling.

deadlines, reflecting the changing complexity of the solution space.

Table 5 shows the variations in the dynamic mode transition counts for the benchmarks for different deadlines (for c=10μf 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.

6.4 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 2 'B' frames between I and P frames; it includes flwr.m2v and cact.m2v. All mpeg files are Test Bitstreams from [22].

Figure 19 shows program execution times 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 runtime results from four different profiling approaches. The leftmost bar shows the runtime for an mpeg run on that input file when the profiling run is also on that input file. The second bar shows the runtime in which the profiling data was collected using the flwr input set for all runs. The third bar shows the runtime when the bbc input was used for the profiling runs. The rightmost bar shows the runtime when optimization is done for the average of flwr and bbc input sets (with equal weight). 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, however. 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

	mpeg/decode	gsm/encode	mpg123	epic	adpcm/encode	ghostscript
Deadline 1	5	1	190	4	0	2
Deadline 2	2645	2777	1559	519	0	7
Deadline 3	5	85	936	552	0	3
Deadline 4	2645	8206	1550	492	0	23
Deadline 5	0	1845	6	4	0	2

Table 5: Dynamic Mode Transition Counts

Benchmark	N_{cache} (Kcycles)	$N_{overlap}$ (Kcycles)	$N_{dependent}$ (Kcycles)	$t_{invariant}$ (μs)
adpcm	732.7	735.6	4302.0	915.9
epic	8835.6	12190.4	9290.1	4955.9
gsm	13979.6	13383.0	29438.3	389.0
mpeg/decode	42621.1	44068.7	27592.1	2713.4

Table 7: Simulation results of program parameters

case makes sure that the deadlines are met for both the cases being considered. Further, Figure 19 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 measured the sensitivity of energy results to specific profile inputs and have found results as good or better than the runtime results presented here; the sensitivity is fairly modest overall.

6.5 A Comparison of Analytical and Profile-Driven Results

By using cycle-level CPU simulation to get the key program parameters as shown in Table 7, we plugged values into the analytic models generated in Section 3 and discussed the resulting maximum energy savings predicted by the models in Table 1. Now, Table 6 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, it is expected to be an optimistic bound, and indeed, the savings predicted by the analytical model exceed those of MILP-based approaches at all but one point. (For the gsm benchmark with three voltage levels at Deadline 5, the simulation energy savings exceeds that of the analytical model by 0.01, apparently a rounding issue.) 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, intra-program 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 intra-program 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 inter-program and no intra-program DVS, our model can help determine a single optimal voltage based on a few simple parameters.

7. DISCUSSION AND FUTURE WORK

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 particularly so. 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 since 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.

More generally, we hope to broaden our MILP formulation to target larger code regions or paths [2]. Moving from edges to paths would allow us to build more program context into our analysis of mode-set positioning. Furthermore, it would also allow us to more accurately profile the time/energy costs of code regions; by not breaking execution on basic block boundaries, our profile would more faithfully reflect execution on deeply-pipelined machines with extensive branch speculation.

8. 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 intra-program 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 non-obvious. 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.

Benchmark	Voltage levels	Deadline 1	Deadline 2	Deadline 3	Deadline 4	Deadline 5
adpcm	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	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

Table 6: Simulation results of energy savings for different numbers of voltage levels.

9. REFERENCES

[1] Advanced Micro Devices Corporation. AMD-K6 processor mobile tech docs, 2002. http://www.amd.com.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

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

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

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Tech. Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept., July 1996.

[6] L. T. Clark. Circuit Design of XScale (tm) Microprocessors, 2001. In 2001 Symposium on VLSI Circuits, Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits.

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

[8] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A modeling language for mathematical programming*. Boyd and Fraser Publishing Company, Danvers, Massachusetts, 1993.

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

[10] C. Hsu and U. Kremer. 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)*, February 2002.

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

[12] C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, 2001.

[13] ILOG CPLEX. Web page for ILOG CPLEX mathematical programming software, 2002. http://ilog.com/products/cplex/.

[14] Intel Corp. Intel XScale (tm) Core Developer's Manual, 2002. http://developer.intel.com/design/intelxscale/.

[15] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 197–202, August 1998.

[16] R. Joseph and M. Martonosi. Run-time power estimation in high-performance microprocessors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[17] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th International Symp. on Microarchitecture*, Dec. 1997.

[18] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation (DAC'00)*, June 2000.

[19] J. Lorch and A. Smith. Improving dynamic voltage algorithms with PACE. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2001)*, June 2001.

[20] J. Luo and N. K. Jha. Power-profile driven variable voltage scaling for heterogeneous distributed real-time embedded systems. In *Int. Conf. VLSI design*, Jan. 2003.

[21] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *Workshop on Complexity-Effective Design*, June 2000.

[22] MpegTv. Mpeg video test bitstreams. http://www.mpeg.org/MPEG/video.html, 1998.

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

[24] T. Sakurai and A. Newton. Alpha-power model, and its application to CMOS inverter delay and other formulas. *IEEE Journal of Solid-State Circuits*, 25:584–594, Apr 1990.

[25] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. Irwin, J. Hu, C.-H. Hsu, and U. Kremer. 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)*, June 2002.

[26] Semiconductor Industry Association. International Technology Roadmap for Semiconductors, 2001. http://public.itrs.net/Files/2001ITRS/Home.htm.

[27] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18(2):20–30, March/April 2001.

[28] V. Swaminathan and K. Chakrabarty. 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)*, January/February 2001.