This paper appeared in Proceedings of LCTES02-SCOPES02, June 19-21, 2002.

Energy-Conscious Compilation Based on Voltage Scaling *

H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu
Dept. of Computer Science & Engineering
Pennsylvania State University
University Park, PA 16802, USA
{saputra,kandemir,vijay,mji,jhu}@cse.psu.edu

C-H. Hsu, U. Kremer
Dept. of Computer Science
Rutgers University
Piscataway, NJ 08855, USA
{chunghsu,uli}@cs.rutgers.edu

ABSTRACT

As energy consumption has become a major constraint in current system design, it is essential to look beyond the traditional low-power circuit and architectural optimizations. Further, software is becoming an increasing portion of embedded/portable systems. Consequently, optimizing the software in conjunction with the underlying low-power hardware features such as voltage scaling is vital.

In this paper, we present two compiler-directed energy optimization strategies based on voltage scaling: static voltage scaling and dynamic voltage scaling. In static voltage scaling, the compiler determines a single supply voltage level for the entire input program. We primarily aim at improving the energy consumption of a given code without increasing its execution time. To accomplish this, we employ classical loop-level compiler optimizations. However, we use these optimizations to create opportunities for voltage scaling to save energy, rather than increase program performance.

In dynamic voltage scaling, the compiler can select different supply voltage levels for different parts of the code. Our compilation strategy is based on integer linear programming and can accommodate energy/performance constraints. For a benchmark suite of array-based scientific codes and embedded video/image applications, our experiments show average energy savings of 31.8% when static voltage scaling is used. Our dynamic voltage scaling strategy saves 15.3% more energy than static voltage scaling when invoked under the same performance constraints.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers; Optimization*

General Terms

Design, Experimentation, Performance

*This work was supported in part by grants from PDG, GSRC and NSF grants CAREER 0093082 and 0093085

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

Keywords

Voltage Scaling, Energy-aware Compilation, Loop Transformations, Optimizing Compilers

1. INTRODUCTION AND MOTIVATION

With the increasingly widespread use of embedded/portable devices, minimizing energy consumption is being promoted to a first-class status in system design [3]. Research community is in consensus that both hardware based and software based work is necessary to improve energy behavior of applications. Over the years, significant progress has been made in the area of low-power circuit and system design (see, for example, [9, 8] and the references therein). The software work is, however, still in its infancy. One of the areas that are in much need of work is energy-aware compilers. This is particularly true for a class of embedded systems where software constitutes a significant portion. A critical issue is to design and implement energy- conscious compilers that improve not only execution time but also energy/power behavior. In this direction, there are two important questions that need to be answered:

- Are current performance-oriented compiler optimizations suitable from the energy perspective? And, if not, how should they be modified to become energy efficient?
- Are there pure energy-oriented compiler optimizations that do not affect execution time? And, how do these optimizations interact with pure performance-oriented techniques?

Both these questions are important, and we believe that they will be the focus of the energy-oriented compiler work for the next couple of years. Note that even high-end desktop systems and servers can make use of energy optimizations for economical and environmental reasons. An example of studies related to the first question posed above is investigating the energy behavior of classical compiler optimizations. For example, a few studies (e.g., [36, 26]) indicate that well-known loop-level optimizations such as iteration space tiling (also known as loop blocking) improve memory energy behavior (in addition to improving the time spent in memory accesses). This is in a sense an ideal scenario where we get both performance and energy benefits by employing a well-studied compiler optimization for which robust implementations exist. Exploiting low-power operating modes that exist in some memory systems is a study that helps us

to answer the second question above. Previous work [12] shows that the compiler is in a good position (in some embedded systems) to place unused memory banks into low-power operating modes (e.g., sleep mode) to conserve energy. While there is a danger that some performance penalty might be incurred if a bank in the sleep mode is called back to service a memory request, the compiler can use a pre-activation strategy that brings the bank back to the active mode (fully-operational mode) before it is required. A similar pre-activation strategy is used in [18] on managing a wireless card through hibernation in the context of remote virtual memory. Such pre-activation strategies are particularly suitable for systems which execute image and video processing applications that exhibit regular data access patterns.

In this paper, we focus on voltage scaling and show how an optimizing compiler can take advantage of this technique to reduce energy consumption. More specifically, we present two different compiler-directed energy optimization strategies based on voltage scaling. In the first optimization strategy, called *static voltage scaling*, we primarily aim at improving the energy consumption of a given code without increasing its execution time. To accomplish this, we employ classical loop-oriented compiler optimizations such as loop permutation, tiling, and loop fusion and distribution [41]. However, in contrast to popular performance-oriented optimization strategies adopted by current optimizing compilers in which these optimizations are used to reduce the number of execution cycles, we use these optimizations explicitly for improving energy behavior. Specifically, our approach first takes a given code and optimizes it using loop-level transformations (as if optimizing for high performance). If the code is run after these transformations, it would normally run much faster than the original (input) code. However, instead of doing so, our approach scales down the supply voltage to the CPU so that the execution time of this optimized code becomes the same as the execution time of the original code. However, since reducing supply voltage has a quadratic effect on dynamic energy consumption (that is, it reduces dynamic energy quadratically), the optimized code (with voltage scaling) would have much lower energy consumption than the original one (without voltage scaling). Note that this optimization strategy uses the same (scaled) voltage for the entire execution of the optimized code. In the second optimization strategy, called *dynamic voltage scaling*, we relax this constraint and allow different parts of the application to be executed under different supply voltage levels. This strategy is based on integer linear programming and can accommodate energy/performance constraints. Clearly, our strategies employ a deadline-based approach where we assume that there is a performance upper-bound, and going beyond this bound (i.e., increasing the performance further) is not necessary. Instead, we try to convert the potential performance surplus to energy benefits. Note that this deadline-based approach makes sense in many real-time embedded environments.

This paper makes the following major contributions:

- It presents a strategy where classical loop-level optimization techniques are used for improving energy consumption rather than improving execution time. Our experiments with a suite of array-dominated codes from scientific and embedded image/video domains indicate that, on the average, 31.8% energy savings are possible by sacrificing a 20.8% potential performance improvement (had we not used voltage reduction).

- It presents an ILP (integer linear programming) based compilation strategy that compiles a given code under multiple energy and performance constraints. Our dynamic voltage scaling strategy selects the best supply voltage for each loop nest in the code taking into account the overhead of scaling the supply voltage between the neighboring nest. The constraints considered during compilation can be automatically derived by analyzing the application code and its execution environment, or alternately, can be explicitly specified by the user. Our preliminary results reveal that this dynamic voltage scaling strategy saves 15.3% more energy than static voltage scaling when invoked under the same performance (execution time) constraint.

The remainder of this paper is organized as follows. In Section 2, we revise the concept of supply voltage reduction and its impact on energy consumption and execution time. In Section 3, we present our static voltage scaling strategy, show how high-level compiler optimizations can be used for energy reduction, and report experimental data. In Section 4, we present our dynamic voltage scaling strategy that employs different supply voltage levels for different parts of the code. In Section 5, we discuss related work, and we conclude the paper with a summary and a brief outline of future work in Section 6.

2. VOLTAGE SCALING BASICS

Dynamic energy consumption is the dominant energy consumption in today's CMOS circuits, and is proportional to KCV^2, where K is the switching rate, C is the capacitance, and V is the supply voltage [9]. It is easy to see that reducing V has a quadratic effect on dynamic energy consumption. Based on this observation, many previous studies (e.g., [33, 29, 6]) investigated voltage scaling (reduction) for low energy. Different voltage scaling techniques based on reliability concerns, transistor sizing, threshold voltage reduction, and architectural improvements have been proposed and evaluated. While these techniques bring about large reductions in energy consumption, reducing supply voltage has a negative impact on execution time. Specifically, the delay (clock cycle time) for a given circuit is proportional to $V/(V-V_t)^2$, where V_t is the threshold voltage. So, reducing supply voltage increases clock cycle time; that is, it requires a corresponding decrease in clock speed. Since execution time of the code can be written as LT where L is the number of clock cycles and T is the clock cycle time, reducing the supply voltage increases the execution time. In other words, voltage scaling exhibits an interesting tradeoff between energy and performance.

Voltage scaling is useful in many contexts. In operating systems (OS) area, dynamic voltage scaling (e.g., [40]) assigns suitable operating voltages to different tasks if doing so does not violate any deadline. Recently, a couple of compiler-based studies (e.g., [20, 21]) have suggested employing low supply voltage in selected portions of the code to reduce overall energy consumption without an excessive increase in execution time. In [20], the authors identify a single region per application and apply dynamic voltage scaling. Compiler-directed voltage scaling is particularly suitable for

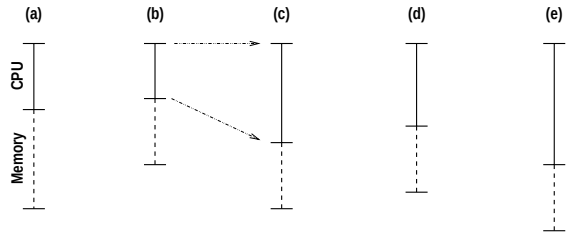

Figure 1: Different execution profiles. (a) Original execution profile. (b) Impact of code optimization. (c) Impact of voltage scaling on the optimized code. (d) Voltage scaling while improving over the original profile. (e) Aggressive voltage scaling (sacrificing performance).

embedded systems that run a single (or a small set of) application(s). Depending on compiler's capability of estimating relative execution times of different portions of the code, it can assign the most suitable voltage for each portion.

3. STATIC VOLTAGE SCALING

In this section, we present a compiler-directed energy optimization strategy based on voltage scaling. We refer to this strategy as static voltage scaling as the supply voltage (once it has been set to a value) is not changed during the course of execution. Our approach is based on the following idea. Applying performance-enhancing optimizations to a given code creates a *performance slack,* which can be defined as the difference between the original (unoptimized) execution time and the optimized execution time. The current trend in optimizing compilers is to exploit this slack to reduce execution time. This slack, however, can also be exploited by converting it to energy reduction as explained below.

3.1 Approach

Let V_o be the original supply voltage. This gives us a dynamic energy consumption of E_o which is proportional to $KCV_o{}^2$ and a clock cycle time of T_o which is proportional to $V_o/(V_o - V_t)^2$. Now, let us assume that, after the optimizations, we have a new (optimized) execution time. Note that this new execution time (like the original one) is obtained using the supply voltage V_o. Suppose now that the supply voltage is reduced to a value of V_r ($< V_o$), which means that the clock cycle time becomes proportional to $V_r/(V_r - V_t)^2$. This does not change the number of the cycles that the (optimized) code will take, but it increases the overall execution time as the execution time is the product of the clock cycle time and the number of cycles. Now, if we are able to select a suitable V_r value to make this last execution time (that of the optimized code after voltage scaling) equal to the original one (with the default supply voltage V_o), we would make the execution time of the optimized code same as that of the original code. However, since $V_r < V_o$, we have a reduction on energy consumption. Note that this is, in a sense, the use of compiler optimizations for reducing energy consumption instead of execution time.

However, the analysis presented above is not very accurate when we consider a computing system with CPU and off-chip memory. This is because it may not be a very good idea to slow down the memory system by applying voltage scaling, as memory components already work with slower frequencies and employ a reduced voltage swing [9]. In fact, the latency formulation mentioned above (that involves supply voltage and threshold voltage) does not apply to memory components directly. Based on this observation, we need to focus on scaling voltage only for the CPU components (on-chip components). This requires a slight modification to our approach. The modified approach proceeds as follows. First, the loop-level compiler optimizations are applied to the input code. Then, the total execution cycles of the resulting optimized code are divided into two portions: CPU cycles and memory cycles, the latter of which are spent during waiting for a memory access to complete. After that, the supply voltage to the CPU is scaled down such that the overall execution time (after scaling) becomes the same as that of the original code.

This scenario is depicted in Figure 1 for an example application. Figure 1(a) shows breakdown of the execution profile of the original (unoptimized) code into two segments: CPU and off-chip memory. After the optimizations, the off-chip memory part is, hopefully, reduced significantly. Depending on the types and overhead of the compiler optimizations used, the CPU portion can increase, decrease, or stay the same. The execution profile after the optimization is illustrated in Figure 1(b). Now, when we apply supply voltage scaling to the CPU part, we obtain the profile in Figure 1(c). The supply voltage is selected such that the length of the profile in Figure 1(c) is equal to that of the original profile in Figure 1(a). In mathematical terms, let $X_o = L_o T_o$ be the original execution time (in terms of cycles). Here, L_o and T_o are the original number of execution cycles and clock cycle time, respectively. Assume further that $X_i = L_i T_o$ is the execution time after the compiler optimizations. We can decompose this optimized time into two portions $X_i = L_{icpu} T_o + X_m$, where L_{icpu} is the number of cycles spent in CPU and and X_m is the time spent in off-chip memory accesses. Obviously, if the optimization is successful, we have $L_i < L_o$. We now apply voltage scaling and obtain a new clock cycle time, T_r, for the CPU such that the execution time, $X_n = L_{icpu} T_r + X_m$, becomes equal to X_o.

There are a couple of issues that need to be clarified before continuing. First, in order for this voltage scaling strategy to be successful, the energy consumption of the optimized code with voltage scaling should be less than its energy consumption without voltage scaling. Otherwise, there is no point in using the voltage scaled version as the code before voltage scaling has typically much better execution time. Second, it is also interesting to study the impact of this energy optimization on energy-delay product. Let X_o be original execution time of the original code and let X_i be the optimized execution time if we do not use any voltage scaling. The energy consumption of the original code E_o is related to the new energy consumption after optimizing the code and scaling the supply voltage, E_n, by $E_n = E_o {V_r}^2/{V_o}^2$. Now, by applying dynamic voltage scaling to the optimized code, we change its energy-delay product from $X_i E_i$ to $X_o E_o {V_r}^2/{V_o}^2$, where E_i is the energy consumption of the optimized code before voltage reduction. We can typically expect that $E_i < E_o$ (as these compiler optimizations decrease the energy consumed in the memory system significantly [26]) and $X_i < X_o$ which means that $X_i E_i < X_o E_o$. Consequently, in order to get an energy-delay improvement using our static voltage scal-

ing approach, the magnitude of the ratio V_r^2/V_o^2 should be small enough to make a difference. Therefore, while the analysis in the previous section reduces dynamic energy consumption without negatively impacting execution time, it does not guarantee that the resulting code has an acceptable energy-delay product. This is because it is possible that $X_i E_i < X_o E_o V_r^2/V_o^2$ may be true even after voltage scaling; that is, the energy-delay product of the optimized code before voltage scaling might be lower than that after voltage scaling. An alternative objective then is to minimize the energy-delay product, which can be done as follows. Let the energy-delay product of a given code be proportional to $KCV_o^2 V_o/(V_o - V_t)^2$, where KCV_o^2 is the energy consumption and $V_o/(V_o - V_t)^2$ is the clock cycle time. To minimize this value, we can take the derivative of this expression with respect to V_o and make it equal to 0. It can be shown that this gives us a V_o value of $3V_t$. While this code may not have a good performance as it is not optimized, we can apply the same idea to the loop-transformed (optimized) code as well. In other words, if we first optimize the code and then scale down the supply voltage to $3V_t$, we obtain a version of the optimized code with the minimum energy-delay product.

It should also be noted that while we have focused so far on a scenario where the execution time of the compiler-optimized code after voltage scaling is equal to the original execution time (without voltage scaling), that is, $X_o = X_n$, our strategy is flexible in the sense that it is possible to improve over the original execution time while still saving energy. Figure 1(d) illustrates this case. The energy savings in this case would not be as high as those in Figure 1(c), but the performance is better. It is also possible to obtain even larger energy savings (than Figure 1(c)) if we are willing to sacrifice more performance. Figure 1(e) depicts a scenario where the execution time after voltage scaling is higher than the original one. In general, we can use our approach to compile a given code under a given performance (execution time) constraint. For example, suppose that the original execution time is X_o and the optimized execution time is X_i. If we have an execution time upper-bound X_{max} which is higher than X_i, then we can scale down the voltage such that the execution time (X_i) becomes X_{max}. In other words, unnecessary performance slack can be converted into energy benefit. It should be emphasized that in many embedded systems energy concerns are as critical as (and sometimes more important than) performance constraints.

3.2 Evaluation

To evaluate the effectiveness of our strategy, we tested it using a set of array-dominated applications. Let us start by giving some information about our simulation environment. In this work, in order to obtain energy values and execution cycles for different code versions (with potentially different supply voltage levels), we use SimplePower [39], an architectural-level, cycle-accurate energy simulator. SimplePower is an execution-driven power estimation tool. It is based on the architecture of a five-stage pipelined data-path. The instruction set architecture is a subset of the instruction set of SimpleScalar, which is a suite of publicly available tools to simulate modern microprocessors [5]. The major components of SimplePower are SimplePower core, RTL power estimation interface, technology dependent switch capacitance tables, cache/bus simulator, and loader. The SimplePower core simulates the activities of all the functional units and calls the corresponding power estimation interfaces to find the switched capacitances. These interfaces can be configured to operate with energy tables based on different micron technologies. Transition sensitive [39], technology dependent switch capacitance tables are available for the different functional units such as adders, ALU, multipliers, shifter, register file, pipeline registers, and multiplexors. The SimplePower core continues the simulation until a predefined program halt instruction is fetched. Once the simulator fetches this instruction, it continues executing all the instructions left in the pipeline, and then dumps the output. The cache simulator of SimplePower is interfaced with an analytical memory energy model derived from that proposed by Shiue and Chakrabarti [36]. The memory energy is divided into that consumed by the cache decoders, cache cell array, the buses between the cache and main memory, and the main memory. The components of the cache energy is computed using analytical energy formulations. An enhanced version of the simulator also reports clock energy using models built in-house [13]. The energy models utilized in the simulator are within 8% error margin of measurement from real systems [7]. The energy simulator can work under the assumption of different supply voltages and micron technologies. It can be configured using the command line to set the caches parameters, output the pipeline trace cycle-by-cycle, and dump the memory image. SimplePower provides the total number of cycles in execution and the energy consumption in different system components (data-path, clock, cache, memory and buses).

Parameter	Value
Default Supply Voltage (without scaling)	2.5V
Threshold Voltage	0.4V
Technology Parameter	0.25 micron
Data Cache Configuration	32 KB, 2-way
Data Cache Hit Latency	1 cycle
Memory Access Latency	100 cycles
Per Access Read Energy for Data Cache	0.20 nJ
Per Access Write Energy for Data Cache	0.21 nJ
Per Access Energy for Memory	4.95 nJ
On-Chip Bus Transaction Energy	0.04 nJ
Off-Chip Bus Transaction Energy	3.48 nJ
Per Cycle Clock Energy	0.18 nJ

Figure 2: Configuration parameters for the energy simulator.

The architecture considered in this work is a single-issue embedded processor with five-stage pipeline operating at 200MHz. The configuration used by the simulator is summarized in Figure 2. In this work, we focus on energy consumed in (on-chip) data cache, off-chip memory, processor core (data-path), and clock circuitry. Consequently, in the rest of the paper, when we say energy consumption, what we mean is the energy consumed in these four components. As mentioned earlier, in this work, we apply voltage scaling to the on-chip components only.

Figure 3 gives important characteristics of the benchmark codes used in this study. We used eight benchmarks to test the effectiveness of our energy saving strategy. The first six benchmarks are array-intensive scientific applications whereas the last two codes are from image processing domain (motion estimation and jpeg encoder/decoder). The third column gives the total dataset size for each code. The fourth and fifth columns in Figure 3 give the energy consumptions (in microjoules) and execution times (in msec)

Benchmark	Source	Input	Energy	Time
aps	Perfect Club	121KB	2481.6	1742.7
btrix	Spec	207KB	298250.3	88492.2
eflux	Perfect Club	290KB	21599.0	15363.1
tomcatv	Spec	274KB	62882.8	16311.3
tsf	Perfect Club	130KB	61680.0	23761.8
vpenta	Spec	147KB	521389.5	234521.0
hier		130KB	9128.4	6628.7
jpeg		224KB	407101.5	186275.3

Figure 3: Benchmark codes used in the experiments.

for the *original* (*unoptimized*) *codes* when executed using a 2.5V supply voltage.

To obtain the optimized versions of these codes, we used four different loop-level compiler optimizations employed by many optimizing compilers from academia and industry: loop permutation, loop tiling, loop fusion, and loop distribution [41]. We selected these specific optimizations because of two reasons: first, these loop-level techniques are well-known and widely used by many optimizing compilers from academia and industry. Second, these optimizations are known to improve the data locality behavior of array-intensive codes significantly. In fact, for applications where the major performance bottleneck is poor cache behavior, these optimizations can be very useful [41]. Our optimization strategy is as follows. First, we apply aggressive loop fission to isolate as many loops as possible. After that, we use loop fusion to combine loops that access the same set of arrays. In a sense, this step converts inter-nest data reuse to intra-nest data reuse. Then, each nest is optimized for locality using the approach explained in Li's thesis [27]. More specifically, for each nest, we first apply linear loop transformations to exploit spatial and temporal reuse in the innermost loop, and then use tiling to take advantage of temporal reuse in outer loop positions. All these loop-level optimizations have been implemented using the SUIF infrastructure [2]. The tile sizes used in tiling are selected after experimenting with a large number of alternative sizes. The resulting C codes are then compiled using the low-level compiler in the SimplePower infrastructure (which is a variant of gcc).

Figure 4 shows the energy consumptions for three different versions of each benchmark code in our experimental suite: the unoptimized code with the original supply voltage (denoted `OR`), the optimized code before voltage scaling (denoted `BV`), and the optimized code after voltage scaling (denoted `AV`). All values given in this figure are *normalized* with respect to the energy consumption of the unoptimized code. We see that, on the average, the `BV` and `AV` versions improve the energy consumption of the original codes by 13.9% and 31.8%, respectively. Figure 5 gives the execution times of these three versions normalized with respect to `OR`. We observe from this figure that, if we do not use voltage scaling, the optimized codes would run 20.8% faster on the average. It should be mentioned that, due to discrete voltage levels that we used, in some cases, it was not possible to scale (reduce) the voltage such that the execution time of the optimized code (after scaling) would exactly be the same as the original execution time (with our original supply voltage of 2.5V).

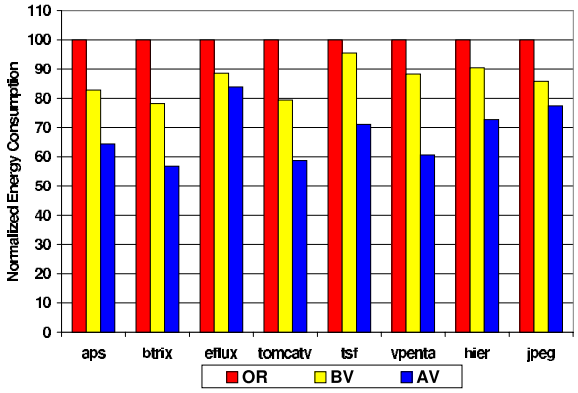

Figure 4: Normalized energy consumption.

4. DYNAMIC VOLTAGE SCALING

The approach presented in the previous section used the same scaled voltage level for the entire execution of the optimized code. In this section, we present a strategy that uses multiple supply voltages within the execution of the same application. Our approach is based on integer linear programming (ILP). An ILP problem is a linear programming problem in which the optimization parameters (solution vector elements) are restricted to be integers. The specific version of the ILP employed in this work is called zero-one ILP (ZILP), where we further restrict each integer variable to be either zero or one. While ILP solutions are known to be time-consuming, the number of variables and constraints used in this work is fairly low; making this a viable approach. In fact, in our experiments, the time spent in the solution of the ILP system was always less than 0.5 sec.

4.1 Approach

Our optimization strategy tries to compile a given input code under (potentially multiple) energy and performance constraints. In doing so, it employs dynamic voltage scaling. It is different from many of the previous studies on dynamic voltage scaling in the sense that the voltage levels to be used in different points in execution are decided by the compiler (rather than the operating system or a kind of hardware mechanism).[1] Our optimization strategy is built upon the idea of pre-computing the energy consumption and execution time of multiple versions of each nest with different (pre-selected) voltage levels and storing them in a table. To be more specific, if we have M different nests in a given code and N different voltage levels, we prepare two tables (one for energy values and the other one for execution cycles) in which each nest has N entries (that is, a total of NM energy values and NM execution time values). While the technique used for selecting the voltage levels to be used might critically influence the success of our strategy, investigating this issue is beyond the scope of this paper. Moreover, our approach can be modified to work with any number of voltage levels and with granularities other than loop nests.

After building such a table (using the energy simulator), our approach takes into account the energy and performance constraints and selects a suitable supply voltage level for each nest. It should be noted that the voltage levels se-

[1] In fact, it would be more accurate to refer to our dynamic scheme as *the static scheme with multiple voltage levels.*

lected for two consecutive nests might be different from each other, and in switching from one voltage level to another, the associated time overhead has to be accounted for. To select voltage levels for the nests, our approach formulates the problem as a ZILP problem and uses a publicly-available solver [35].

Let us first make the following definitions:

- $E_{i,j}$: Estimated energy consumption for nest i (of the optimized code) when the jth voltage level is used; and
- $X_{i,j}$: Estimated execution time for nest i (of the optimized code) when the jth voltage level is used,

where $1 \leq i \leq M$ and $1 \leq j \leq N$. We use zero-one integer variables $s_{i,j}$ to indicate whether the jth voltage level is selected (as the final voltage level) for nest i or not. Specifically, if $s_{i,j}$ is 1 this means that the jth voltage level is selected for nest i, if $s_{i,j}$ is 0, it is not. Since for each nest the final code should contain only one voltage level, the constraint

$$\sum_{j=1}^{N} s_{i,j} = 1 \tag{1}$$

should be satisfied for each nest i. Consequently, the total energy consumption (E) and total execution time (X) of the code can be expressed as:

$$E = \sum_{i=1}^{M} \sum_{j=1}^{N} s_{i,j} E_{i,j} \tag{2}$$

$$X = \sum_{i=1}^{M} \sum_{j=1}^{N} s_{i,j} X_{i,j} \tag{3}$$

However, changing the supply voltage between nests does not come for free. It takes typically 100 msec to 200 msec to switch from one voltage level to another. Consequently, we also define a voltage switching overhead as follows:

$$O = \sum_{i=1}^{M-1} \sum_{j=1}^{N} |s_{i,j} - s_{i+1,j}| (\frac{OVHD}{2}) \tag{4}$$

where $OVHD$ is the time to switch between two different voltage levels (e.g., a fix cost such as 100 msec for any voltage level pair). By re-writing the absolute value operator within a linear programming framework, we can re-express the above overhead formulation as:

$$O = (\frac{OVHD}{2}) \sum_{i=1}^{M-1} \sum_{j=1}^{N} y_{i,j} \tag{5}$$

$$y_{i,j} \geq s_{i,j} - s_{i+1,j} \tag{6}$$

$$-y_{i,j} \leq s_{i,j} - s_{i+1,j} \tag{7}$$

We refer to constraints (1), (2), (3), (5), (6), and (7) as the basic constraints in the rest of this paper. Each optimization problem that we address in the following is expressed by augmenting these basic constraints by some additional constraints (on performance and energy) and an objective function.

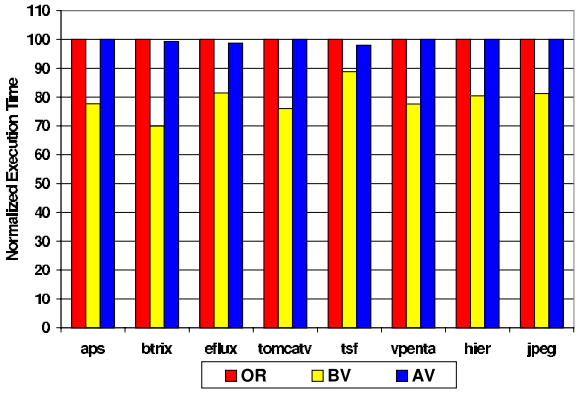

Figure 5: Normalized execution time.

4.2 Evaluation

To illustrate our constraint-based compilation strategy, we first consider the `tomcatv` program (its optimized version) from the Spec benchmarks. Figures 6(a) and (b) give, respectively, the energy consumptions and execution times for each nest of this code under different supply voltage levels. All energy values are in microjoules and all performance numbers are in msec. We see that there is a large variation between energy and performance values for different voltage levels.

Let us assume for the moment that the overhead of switching voltages during execution is zero. We now consider different compilation strategies whose objective functions and additional constraints (if any) are given in the second and third columns, respectively, of Figure 7. First, let us focus on two simple cases: `Case-I` and `Case-II`. In `Case-I`, we try to minimize execution time; this is the classical objective of a performance-oriented compilation strategy. We see from columns 4 through 12 that in this case the highest voltage level is selected for each nest (as expected). `Case-II` represents the other end of the spectrum where we strive for minimizing the energy consumption (without any energy concerns). Obviously, the LP solver selects the lowest voltage level for each nest. The last two columns of Figure 7 give the energy consumption (in microjoules) and execution time (in msec) for each scenario. Now, comparing these two columns for `Case-I` and `Case-II`, we see that compiling for performance and energy can generate very different results. For example, the execution time of `Case-I` is 82.5% less than that of `Case-II`. On the other hand, the energy consumption of `Case-II` is 25.6% less than that of `Case-I`.

The remaining compilation strategies in Figure 7 provide even more interesting cases. Let us focus now on `Case-III` and `Case-IV`, which correspond to minimizing energy consumption under performance constraints. It should be noted that the voltage levels selected for these cases are different from those selected for `Case-II`. We also observe that as the execution time upper-bound is reduced from 20000 to 15000, we see an increase in the selected voltage levels. The next two cases (`Case-V` and `Case-VI`) represent optimizing execution time under energy constraints. When we compare the voltage levels selected by these strategies with those selected by `Case-I`, we observe that there is a general reduction in selected voltage levels (as we now have energy constraints to be accounted for).

(a) Energy Consumptions

Nest	Supply Voltage Levels								
	2.5V	2.3V	2.1V	1.9V	1.7V	1.5V	1.3V	1.1V	0.9V
n1	3329.24	3236.31	3151.12	3073.68	3003.98	2942.03	2887.82	2841.35	2802.63
n2	318.18	315.87	313.76	311.84	310.11	308.58	307.23	306.08	305.12
n3	15807.6	14687.5	13660.7	12727.2	11887.1	11140.4	10487.0	9926.92	9460.20
n4	2909.99	2787.87	2675.93	2574.17	2482.58	2401.17	2329.94	2268.88	2218.00
n5	465.73	462.76	460.04	457.57	455.31	453.36	451.63	450.15	448.91
n6	8660.43	8369.75	8103.30	7861.07	7643.07	7449.28	7279.72	7134.39	7013.27
n7	544.60	540.57	536.88	533.52	530.50	527.82	525.47	523.46	521.78
n8	6570.82	6337.78	6124.17	5929.97	5755.19	5599.83	5463.90	5347.38	5250.28
n9	6074.30	5917.53	5773.82	5643.18	5525.60	5421.09	5329.64	5251.25	5185.93

(b) Execution Times

Nest	Supply Voltage Levels								
	2.5V	2.3V	2.1V	1.9V	1.7V	1.5V	1.3V	1.1V	0.9V
n1	565.66	619.45	688.01	778.24	901.91	1080.95	1360.71	1850.86	2888.74
n2	26.43	27.35	28.54	30.11	32.25	35.35	40.19	48.68	66.65
n3	6508.61	7270.75	8242.32	9520.77	11273.2	13810.00	17774.1	24719.4	39425.8
n4	771.44	850.70	951.75	1084.72	1266.98	1530.83	1943.12	2665.47	4195.03
n5	38.05	39.32	40.93	43.06	45.98	50.21	58.82	68.39	92.89
n6	1979.78	2187.87	2453.15	2802.22	3280.7	3973.35	5055.72	6952.05	10967.5
n7	48.77	50.82	53.41	56.84	61.53	68.32	78.94	97.55	136.94
n8	1649.88	1824.07	2046.12	2338.31	2738.83	3318.63	4224.63	5811.98	9173.16
n9	1125.3	1234.51	1373.72	1556.9	1808.00	2171.49	2739.5	3734.66	5841.89

Figure 6: Energy and performance numbers for each nest of `tomcatv` under different supply voltage levels.

Case	Objective	Additional Constraints	Selected Supply Voltage Levels									Energy Consumed	Exec. Time
			n1	n2	n3	n4	n5	n6	n7	n8	n9		
Case-I	min X	none	2.5V	2.5V	2.5V	2.5V	2.5V	2.5V	2.5V	2.5V	2.5V	44680.9	12713.9
Case-II	min E	none	0.9V	0.9V	0.9V	0.9V	0.9V	0.9V	0.9V	0.9V	0.9V	33206.1	72788.6
Case-III	min E	$X \leq 20000$	1.9V	1.5V	1.7V	1.9V	1.5V	1.9V	1.3V	1.9V	1.9V	38256.6	19998.1
Case-IV	min E	$X \leq 15000$	1.9V	1.9V	2.3V	2.1V	1.9V	2.1V	2.1V	2.1V	2.1V	41745.0	15000.0
Case-V	min X	$E \leq 40000$	1.9V	1.9V	1.9V	2.1V	1.9V	2.1V	2.1V	2.3V	2.1V	40000.0	17026.3
Case-VI	min X	$E \leq 36000$	1.5V	1.3V	1.5V	1.3V	1.3V	1.3V	1.5V	1.5V	1.5V	36000.0	27546.0
Case-VII	min E	$X + O \leq 20000$	1.9V	1.9V	1.9V	1.5V	1.5V	1.7V	1.7V	1.7V	1.7V	38421.4	19800.1
Case-VIII	min E	$X + O \leq 15000$	2.3V	2.3V	2.3V	2.1V	2.1V	2.1V	2.1V	2.1V	2.1V	41913.8	14836.6
Case-IX	min $X + O$	$E \leq 40000$	1.7V	2.1V	2.1V	1.9V	1.9V	1.9V	1.9V	1.9V	1.9V	39977.9	17054.8
Case-X	min $X + O$	$E \leq 36000$	1.5V	1.5V	1.5V	1.3V	1.3V	1.3V	1.3V	1.5V	1.5V	36000.0	27549.7

Figure 7: Different compilation strategies and selected supply voltages for each nest (`tomcatv`).

So far, we assumed that the overhead of switching voltages between nests is zero. The remaining cases in Figure 7 assume the use of an external voltage regulator that requires 100msec for voltages to change from one value to another. As the voltage regulator is external to the design, we do not model the energy consumption of the regulator itself during either regular operation or the transition phases. Further, the transition phase consumes less than 1% more energy than normal operation in some implementations [14]. Note that $X + O$ correspond to the execution time with voltage conversion overhead included. It is easy to see that when the overhead is taken into account, the ILP solver tends to keep the supply voltage at the same level (between neighboring nests) as much as possible. It should be mentioned that the execution times given for the last four cases under the last column does not contain the time spent in voltage conversion (the cumulative voltage conversion time can easily be calculated, though, by counting the number of times the supply voltage switches). We also observe that even in these cases, compiling for energy and compiling for performance generate very different results (both from the selected voltages and from the final energy/performance numbers perspectives).

We next focus on the remaining codes in our experimental suite and show the benefits of compiler-directed dynamic voltage scaling. Note that both static voltage scaling (Section 3) and dynamic voltage scaling (this section) can reduce the energy consumption of a given code under performance (execution time) constraints. However, dynamic voltage scaling has more flexibility in selecting voltages; so, under the same execution time constraint, it can generate better energy results than static voltage scaling. As an example, assume a code that contains two separate nests, `n1` and `n2`, that can execute under two supply voltages, 2.3V and 1.9V. Assume further that the energy consumption of `n1` under 2.3V and 1.9V are 2000 microjoules and 1000 microjoules and that the corresponding execution times are 400 msec and 500 msec. Let `n2` has the same energy and performance values as `n1` for these voltage levels. Now, assuming that we want to minimize energy while keeping the execution time less than or equal to 900 msec (that is, any execution time less than or equal to 900 msec is equally acceptable), the static approach has only one choice: keeping the supply voltage at 2.3V for both the nests. In this case, the total execution time is 800 msec and the energy consumption is 4000 microjoules. Note that using a 1.9V supply voltage for both the nests would violate the performance (execution time) constraint. On the other hand, our dynamic voltage scaling approach can set the supply voltage to 1.9V in one

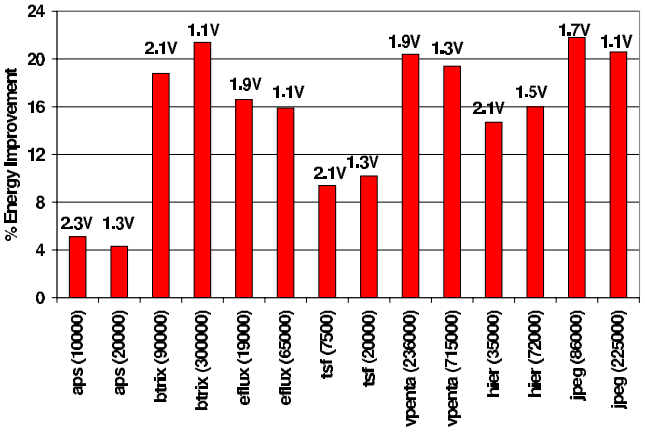

Figure 8: Percentage energy improvements over static voltage scaling.

of the nests (and 2.3V in the other) without violating the execution time constraint. Note that the energy consumption in this case is 3000 microjoules (25% less). This small example shows how dynamic voltage scaling can generate better energy behavior than static voltage scaling.

Figure 8 shows the percentage energy savings brought about by dynamic approach over static voltage scaling. For each benchmark, two different execution time upper limits are tested. The limit values (in msec) are written within parentheses following the name of the benchmark (on the x-axis). Also, for each experiment, the voltage level selected by the static approach is written on top of the corresponding bar. We see from these results that dynamic voltage scalings saves, on the average, 15.3% more energy than static voltage scaling. We also note that as compared to other codes, our approach is less successful with `aps` and `tsf` as these two benchmark codes contain very few nests, making it difficult to apply dynamic voltage scaling.

5. RELATED WORK

Chandrakasan et al. [10] address the problem of automatically finding computational structures that result in the lowest power consumption for a specified throughput given a high-level algorithmic specification. In fact, our work is similar in spirit to their work except for the fact that we would like to optimize software rather than hardware and that the constraints considered in these two studies are different (i.e., execution time in our case and throughput in their case). Also, while they rely on high-level synthesis tools, we employ an optimizing compiler. We believe that in an embedded system design environment, these two studies are complementary.

Microprocessors that support dynamic voltage scaling (DVS) have been available, such as such as Transmeta's Crusoe, Intel's Xscale, AMD's K6-IIIE+, and the prototypes from U.C., Berkeley [4] and from Delft University of Technology [34]. Current implementations have 200-800 MHz CPU core and take about 75-520 microseconds to perform voltage scaling. With such high latency of transition, the frequency of dynamically changing voltage need to be taken extreme care.

There are many proposals for power management of a DVS-capable processor. Most of them are at operating system level and are either task-based (e.g., [42, 25, 19, 32, 28, 37]) or interval-based (e.g., [40, 16]). While some proposals aim at reducing energy without compromising performance, a recent study by Grunwald et al. [17] observed noticeable performance loss for some interval-based algorithms using actual measurements.

Our ILP-based compiler strategy is similar to some of the proposed task-based approaches. However, our methodology differs from theirs in at least two aspects. First of all, unlike their work, we do not use analytical models to evaluate the performance and energy at different voltage levels. Instead, we estimated the performance and energy using the simulator. These analytical models assume a perfect memory model and may predict far off for the codes that involve cache misses. Secondly, these algorithms do not take the scaling overhead into account. As we mentioned above, these long latency overheads need to be considered as well. Otherwise, too many scalings will degrade the performance significantly.

Whether to exploit opportunities for voltage scaling mainly in the operating system or through hardware and/or compilation techniques is still an open question. For the compiler level approach, Hsu et al. implemented a compiler that identifies single or multiple program regions that can be slowed down without significant performance impact [20, 22, 23]. Other methodologies include the work in [31] that instruments program regions to compensate for the worst-case execution time assumption when determining a voltage scaling schedule. For the hardware level approach, Ghiasi et al. [15] and Childers et al. [11] suggested using desired IPC rate to direct the voltage scaling. Marculescu in [30] proposed to use cache misses as the scaling points.

Our work is mostly similar to the approach by Hsu et al. [20] and their compiler implementation [22, 23]. However, there are significant differences. Their compiler uses a non-linear analytical model to predict the performance and energy at different voltage levels, while ours uses simulation results for discrete voltage levels. As a result, their dynamic voltage scaling algorithm is formulated as a non-linear integer programming problem, while we are able to use a linear integer programming formulation. Their work targets superscalar and out-of-order issue architectures, and ours assumes a single-issue embedded processor. Finally, we exploit the difference between the unoptimized execution time and the optimized execution time. In contrast, they assume an user provided performance slack, for instance in the range of 1% to 10% of optimized codes [23].

Extensive research on optimizing compilers has been carried out in many years and targets at high performance. However, there is a growing interest in optimizing software for low power. While most of the researches on compiling for low power investigate the back-end optimizations such as reducing bit switching activity in buses, we address in this paper the high-level optimizations, especially loop-level transformations.

Kandemir et al. [26] found out that loop-level transformations such as tiling tend to reduce memory energy usage at the cost of increasing CPU energy usage [26]. This tradeoff has been recently studied by Hsu and Kremer[21]. They found out that, in some cases, more aggressive optimizations introduce significant CPU workload with marginal performance improvement. As a result, not only it may increase the entire system energy usage, but also it prohibits the dynamic voltage scaling to be applied effectively.

6. CONCLUSIONS AND FUTURE WORK

Voltage scaling is a powerful method for reducing energy consumption in electronic systems. In this paper, we presented two *compiler-directed* voltage scaling strategies: static voltage scaling and dynamic voltage scaling. In static voltage scaling, the application code under consideration is first optimized using an optimizing compiler. After that, the supply voltage is reduced such that the execution time of the optimized code is the same as that of the original code without voltage scaling. In a sense, this strategy converts the performance slack created by compiler optimizations to energy benefit. One limitation of this scheme is that the same (scaled) voltage is used throughout the execution. In dynamic voltage scaling, on the other hand, different portions of the code can operate under different supply voltages. We presented an ILP based, compiler-directed dynamic voltage scaling strategy and showed that, under the same performance constraints, it generates better energy results than static voltage scaling.

This work can be extended in several ways. First, we plan to apply voltage scaling in granularities other than loop nests (e.g., at the procedure level). Second, in the approach presented in this paper, the voltages to be used at runtime are determined at compile time. We also plan to implement a fully-dynamic strategy where the compiler cooperates with a runtime system, and the supply voltage levels to be used are negotiated at runtime. While, for the array-intensive benchmarks, this strategy may not bring too much additional benefit over the techniques discussed in this paper (due to the fact that in array-dominated codes the locality behavior is largely independent from the values of the array elements), it can be very useful for applications with irregular (e.g., input-dependent) data accesses. Third, we would like to investigate the impact of different optimizations on the effectiveness of static and dynamic voltage scaling.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng The SUIF compiler for scalable parallel machines. In Proc.*the Seventh SIAM Conference on Parallel Processing for Scientific Computing,* February, 1995.

[3] L. Benini, R. Hodgson, and P. Siegel. System-level power estimation and optimization. In Proc. *International Symposium Low Power Electronics and Design*, Monterey, CA, 1998.

[4] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of 2000 International Symposium on Low Power Electronics and Design (ISLPED'00)*, July 2000.

[5] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. *Technical Report CS-TR-96-103,* Computer Science Dept., University of Wisconsin, Madison, July 1996.

[6] E. Chan, K. Govil, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In Proc. *the 1st ACM International Conference on Mobile Computing and Networking,* pages 13–25, November 1995.

[7] R. Chen, R. Bajwa and M. J. Irwin. Architectural level power estimation and design experiments. *ACM Transactions on Design Automation of Electronic Systems*, 2001.

[8] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits.* IEEE Press, 2001.

[9] A. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design.* Kluwer Academic Publishers, 1995.

[10] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen. In Proc. *Optimizing power using transformation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 14, No. 1, pp. 12–30, January 1995.

[11] B. Childers, H. Tang, and R. Melhem. Adapting processor supply voltage to instruction-level parallelism. In *Kool Chips 2000 Workshop*, December 2000.

[12] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In Proc. *the 7th International Conference on High Performance Computer Architecture,* Monterrey, Mexico, January 2001.

[13] D. Duarte, N. Vijaykrishnan, M. J. Irwin and M. Kandemir. Formulation and validation of an energy dissipation model for clock generation circuitry and distribution network. In Proc. *International Conference on VLSI Design*, January 2001.

[14] D. Duarte, N. Vijaykrishnan and Irwin, M.J., "A Complete Phase-Locked Loop Power Consumption Model", To appear in Proc. *Design, Automation and Test in Europe Conference*, March 2002.

[15] 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.

[16] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *the 1st ACM International Conference on Mobile Computing and Networking (MOBICOM-95)*, pages 13–25, November 1995.

[17] D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI-2000)*, October 2000.

[18] J. Hom and U. Kremer Energy management of virtual memory on diskless devices. In Proc. *the 2nd Workshop on Compilers and Operating Systems for Low Power, attached to PACT'01,* Barcelona, Spain, September 2001.

[19] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable voltage core-based systems. In *Proceedings of the 35th ACM/IEEE Design Automation Conference(DAC'98)*, pages 176–181, June 1998.

[20] C-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency/voltage scheduling for energy reduction in microprocessors. In Proc. *International Symposium on Low Power*

Electronics and Design, August 2001.

[21] C-H. Hsu and U. Kremer. Dynamic Voltage and frequency scaling for scientific applications. In Proc. *the Workshop on Languages and Compilers for Parallel Computing,* August 2001.

[22] C.-H. Hsu and U. Kremer. Compiler-Directed Dynamic Voltage Scaling Based on Program Regions. Technical Report DCS-TR-461. Department of Computer Science, Rutgers University. Nov 2001.

[23] C.-H. Hsu and U. Kremer.Single Region vs. Multiple Regions: A Comparison of Different Compiler-Directed Dynamic Voltage Scheduling Approaches. Workshop on Power-Aware Computer Systems (PACS'02. Feb 2002.

[24] P. Stanley-Marbell, M. Hsiao and U. Kremer. A Hardware Architecture for Dynamic Performance and Energy Adaption. Workshop on Power-Aware Computer Systems (PACS'02. Feb 2002.

[25] 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.

[26] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In Proc. *the 37th Design Automation Conference,* Los Angeles, California USA, June 5-9, 2000.

[27] W. Li. *Compiling for NUMA parallel machines.* Ph.D. Thesis, Cornell University, 1993.

[28] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. In *Proceeding of the International Conference on Acoustics, Speech and Signal Processing*, June 2000.

[29] D. Marculescu. Power efficient processors using multiple supply voltages. In Proc. *the 1st Workshop on Compilers and Operating Systems for Low Power, attached to PACT'00,* 2000.

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

[31] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compiler and Operating Systems for Low Power (COLP'00)*, October 2000.

[32] T. Okuma, T. Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In *Proceedings of the 12th International Symposium on System Synthesis (ISSS'99)*, 1999.

[33] T. Pering, T. Burd, and R. Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. In Proc. *Power Driven Microarchitecture Workshop, attached to ISCA'98,* June 1998.

[34] J. Pouwelse, K. Langendoen, and H. Sips. Voltage scaling on a low-power microprocessor. In *International Symposium on Mobile Multimedia Systems & Applications (MMSA'2000)*, November 2000.

[35] H. Schwab. *lp_solve Mixed Integer Linear Program Solver,* ftp://ftp.es.ele.tue.nl/pub/lp_solve/

[36] W-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems, *Technical Report CLPE-TR-9-1999-20,* Arizona State University, 1999.

[37] 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.

[38] V. Tiwari, S. Malik, A. Wolfe, and T. C. Lee. Instruction level power analysis and optimization of software, *Journal of VLSI Signal Processing Systems,* Vol. 13, No. 2, August 1996.

[39] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *the International Symposium on Computer Architecture,* Vancouver, British Columbia, June 2000.

[40] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In Proc. *the USENIX Symposium on Operating Systems Design and Implementation,* 1994, pp. 13–23.

[41] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.

[42] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 374–382, October 1995.