Single Region vs. Multiple Regions: A Comparison of Different Compiler-Directed Dynamic Voltage Scheduling Approaches*

Chung-Hsing Hsu and Ulrich Kremer

Department of Computer Science
Rutgers University
Piscataway, New Jersey, USA
{chunghsu, uli}@cs.rutgers.edu

Abstract. This paper discusses the design and implementation of a profile-based power-aware compiler using dynamic voltage scaling. The compiler identifies program regions where the CPU can be slowed down without resulting in a significant overall performance loss. Two strategies have been implemented in `SUIF2`. The single-region strategy slows down a single region for energy savings, while the multiple-region strategy slows down as many regions as needed. A comparison of both strategies based on six `SPECfp95` benchmarks shows that in five cases, the energy-delay product was comparable. In the remaining case, the multiple-region strategy was significantly better. Both strategies achieved energy savings of up to 48% for the five programs at the slowdown between 1% and 16%, and energy savings of 74% for the multiple regions vs. 50% for the single region strategy for the remaining program at the slowdown up to 21%.

1 Introduction

With the advances in technology, power is becoming a first-class architecture design constraint not only for embedded/portable electronic devices but also for high-end computer systems [18]. In this paper, we focus on *software*-controlled power management using dynamic voltage scaling [19]. Dynamic voltage scaling is a technique that varies the CPU supply voltage and frequency on-the-fly to provide multiple power modes with different performance levels. Energy efficient computation can be achieved by dynamically adapting the CPU performance level to the current needs, i.e., reducing the CPU supply voltage and frequency when the CPU is not being fully utilized.

The presented approach uses whole program analyses at compile time to identify CPU slack that can be exploited through dynamic voltage scheduling without resulting in significant performance penalties. The target applications are not real-time systems with hard deadlines, but applications that may tolerate a small performance loss in exchange for power and energy savings. Given a soft execution deadline, i.e., a user supplied acceptable performance penalty, our

* This research was partially supported by NSF CAREER award CCR-9985050.

compilation strategy determines CPU slowdown factors for program regions that are expected to yield the highest energy savings within the performance penalty range.

One factor that distinguishes our work from others is the type of CPU slacks being exploited. Some work (e.g. [19, 15]) identifies the slacks between the processing time and human perception time, while others (e.g. [13, 21]) take advantage of the difference between the conservative performance estimation and the real execution time for applications with hard performance deadlines. In this paper we exploit a *third* type of CPU slacks – in which the memory accesses are on the critical path for performance. Since the CPU processing is not on the critical path, it can be slowed down without introducing significant performance loss. However, the CPU may not be slowed down too much since it issues memory instructions and may alter the criticality of the memory bottleneck.

The presented work is one of the first to use a compiler approach for voltage scheduling. Compilers have the advantage that the entire program can be analyzed, and in addition be modified to exhibit a desired characteristic, thereby enabling further optimizations. Compilation strategies work well if the program behavior can be derived at compile time. For such applications, more aggressive optimizations can be performed, and the performance and energy overhead introduced by operating systems or hardware approaches can be avoided. However, not all programs allow static analyses that yield sufficient information about their runtime characteristics. In such cases, operating system and/or hardware techniques (e.g. [23]) are more promising strategies. We believe that hybrid approaches for voltage scheduling consisting of a combination of compiler, operating system, and hardware strategies will be most effective and necessary, for instance in multiprogramming environments. A discussion of such hybrid approaches is beyond the scope of this paper.

This paper presents the design and implementation of a profile-based power-aware compiler using dynamic voltage scaling. The compiler identifies memory-bound program regions and implements two strategies in `SUIF2`. The single-region strategy slows down a single region for energy savings, while the multiple-region strategy slows down as many regions as needed. A comparison of both strategies based on six `SPECfp95` benchmarks shows that in five cases, the energy-delay product was comparable. In the remaining case, the multiple-region strategy was significantly better. Both strategies achieved energy savings of up to 48% for the five programs at the slowdown between 1% and 16%, and energy savings of 74% for the multiple regions vs. 50% for the single region strategy for the remaining program at the slowdown up to 21%.

2 Basic Compilation Strategy

Our compilation strategy tries to find memory-bound program regions where CPU may be slowed down without affecting significantly the overall program performance. The basic idea is to "hide" the degraded CPU performance behind the memory hierarchy accesses which are on the critical path. Within each such region, a slowdown factor will be selected by the compiler. A slowdown factor δ is defined as a ratio of the peak CPU frequency to the desired frequency. For example, $\delta = 2$ on a 1 GHz machine indicates the desired frequency of 500 MHz. In our model, we assume that the dynamic frequency and voltage changes only occur between regions with different slowdown factors.

Our compilation strategy considers the entire program (P) as the union of program regions (R_i), i.e., $P \stackrel{\text{def}}{=} \bigcup_i R_i$, each of which is characterized by a quadruple $(W_i^c, W_i^b, W_i^m, v_i)$. The values of W_i^c, W_i^b, W_i^m represent the workload (in cycles) of different parts of region R_i, and v_i represents the number of times R_i is accessed for the entire program execution. The total workload for region R_i is then defined as $W_i = W_i^c + W_i^b + W_i^m$, and the total workload for program P is defined as $W = \sum_i W_i$.

A program region is split into three parts to better estimate the performance impact of the CPU slowdown for a region [9]. Specifically, if region R_i is slowed down by a factor of δ, the resulting performance will become

$$W_i(\delta) \stackrel{\text{def}}{=} \delta * W_i^c + \max(\delta \cdot W_i^b, W_i^b + W_i^m)$$

where

- W_i^c is the number of cycles in region R_i that the CPU is busy while the memory is idle (*c*pu_busy); this includes CPU pipeline stalls due to hazards and activities of both level one cache and level two cache,
- W_i^m is the number of cycles in region R_i that the CPU is stalled while waiting for data from memory (*m*emory_busy),
- W_i^b is the number of cycles in region R_i that both CPU and memory are active at the same time (*b*oth_busy).

Slowdown factor δ, by its definition, is never less than one, i.e., $\delta \geq 1$. In addition, the total workload of region R_i, W_i, is treated as an abbreviation of $W_i(1)$.

The model assumes that CPU cycles that did not overlap with memory activities before the slowdown, W_i^c, will also not overlap with memory activities after the CPU slowdown, and that the CPU cycles that did overlap with memory activities before the slowdown, W_i^b, will maintain that property after the slowdown. As a result, a performance penalty of $\delta * W_i^c$ will occur if the entire W_i^b workload can be hidden behind the memory activity workload ($W_i^b + W_i^m$). If only partial hiding is possible, an additional performance penalty will be accounted for.

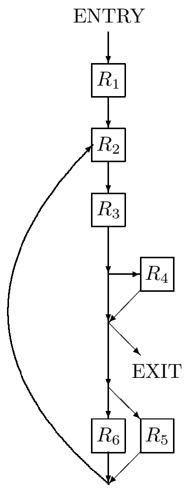
The dynamic voltage scheduling based on program regions is formulated as follows: given a program ($P = \bigcup_i R_i$), we are solving the mixed-integer nonlinear programming (MINLP) problem (P) for variables δ_i's:

$$\text{(P)}\quad \begin{array}{l} \text{minimize } E = \frac{1}{W} \cdot \sum_i W_i/\delta_i^2 \\ \text{subject to} \\ \quad \sum_i W_i(\delta_i) + s \cdot \sum_{i,j} v_{ij} \cdot \theta(\delta_i, \delta_j) \leq (1+r) \cdot W \\ \quad 1 \leq \delta_i \\ \quad \delta_i \in \mathbb{R} \end{array}$$

where s represents the performance cost of each transition, v_{ij} represents the number of transitions between region R_i and R_j, and r represents the user-defined performance penalty, Function $\theta(\cdot, \cdot)$ indicates whether a transition occurs or not and is defined as follows.

$$\theta(\delta_i, \delta_j) \stackrel{\text{def}}{=} \begin{cases} 0 \text{ if } \delta_i = \delta_j \\ 1 \text{ otherwise} \end{cases}$$

The first inequality models the resulting performance of all regions after their respective slowdown, plus the transition costs introduced by switching between different voltages/frequencies. Problem (P) searches for the appropriate δ_i values such that the performance penalty of the voltage scaled program does not exceed the user-specified value and its (relative) energy consumption E is minimized.



R_i	W_i^c	W_i^m	W_i^b	v_i
R_1	116.98	27.35	8.27	1
R_2	52.82	200.67	54.46	10
R_3	59.70	207.52	38.34	10
R_4	2.81	4.43	1.22	1
R_5	0.06	27.98	2.72	1
R_6	9.79	201.25	52.06	8

Fig. 1. $W = 1068$ million cycles, $s = 10000$, and $r = 0.01$. Workloads are given in million cycles. The graph on the left represents the control flows between regions $R_1 - R_6$. On the right, there are workload characteristics for each region R_i, which is recorded by the profile.

For example, the real execution of `swim` on training input indicates the following transitions for Figure 1

$$R_1 \xrightarrow{1} R_2,\ R_2 \xrightarrow{10} R_3,\ R_3 \xrightarrow{1} R_4,\ R_3 \xrightarrow{1} R_5, R_3 \xrightarrow{8} R_6,\ R_5 \xrightarrow{1} R_2,\ R_6 \xrightarrow{8} R_2.$$

where the weight of each edge represents v_{ij}. Given performance tolerance of 1% ($r = 0.01$) and an assumed voltage scaling overhead of 10,000 cycles ($s =$ 10000), the optimal solution for problem (P) is $E = 77.3\%$ with the following δ assignment:

$$\delta_1 = 1,\ \delta_2 = 1.03,\ \delta_3 = 1.03,\ \delta_4 = 1,\ \delta_5 = 4.58,\ \delta_6 = 1.75.$$

Using the optimal δ assignment, the compiler can then insert DVS instructions at appropriate places. In our example, the entries of regions R_2, R_4, R_5, and R_6 are "guarded" with DVS instructions of desired CPU frequency. The simulation result showed that the performance degradation is 2.2% with relative energy consumption of 75.1%.

A special case of the region-based compilation strategy is to find a *single* program region that minimizes the energy consumption within the same $(1+r)\cdot W$ deadline. Issues related to the single-region strategy have been discussed in [8]. An important difference between the single-region strategy and the multiple-region strategy is that the single-region strategy considers the *combined* regions as well, i.e., $R_{i\&j} = R_i \cup R_j$. For example, the single-region strategy not only evaluates regions R_1 - R_6 but also examines combined regions $R_{2\&3}$, $R_{5\&6}$, and R_{1-6}. With the same performance tolerance of 1% and DVS overhead of 10,000 cycles, the compiler found that the combined region $R_{5\&6}$ with $\delta_{5\&6} = 2.07$ gives the best energy saving. That is, the δ assignment derived by the single-region strategy can be described as follows:

$$\delta_1 = 1,\ \delta_2 = 1,\ \delta_3 = 1,\ \delta_4 = 1,\ \delta_5 = 2.07,\ \delta_6 = 2.07.$$

Experimental results showed that this selection resulting in energy consumption of 75.7% and a performance penalty of 2.7%.

The experimental results showed that, for five of six `SPECfp95` benchmarks we tested, both multiple-region strategy and single-region strategy were similar in their effectiveness. Multiple-region strategy did much better in one benchmark. While conceptually the multiple-region strategy should be as effective as the single-region strategy, it is more complicated to implement and it raises different issues from the single-region strategy. More detailed comparisons will be discussed in Section 4.

3 Implementation

The prototype of the dynamic voltage scheduling based on program regions is implemented as part of `SUIF2` [24] using the profile-driven approach. Both multiple-region strategy and single-region strategy are implemented. The prototype implementation has four phases. The first phase instruments the original C program at appropriate program locations. The instrumented code is then executed (the second phase), collecting information needed in the third phase. The third phase uses both the instrumented program and the profile information to determine the best δ assignment for the program. Once the slowdown factors of regions are determined, the final phase restores the instrumented program back to the original one, and inserts speed setting instructions at appropriate region boundaries. The output of the prototype is the original program with a few additional DVS instructions. Figure 2 shows the phases of the implementation.

Phase 1: Instrumentation – Two kinds of program constructs are instrumented in our implementation, namely call sites and explicit loop structures.

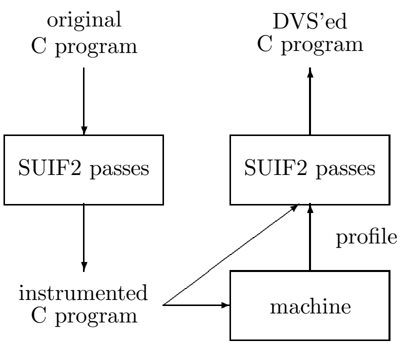


Fig. 2. The flow diagram of the compiler implementation.

Explicit loop structures include `for`, `while` and `do-while` loops. Loops based on `goto`'s are not instrumented in the current implementation.

Phase 2: Profiling – The information collected for each instrumented program construct R is the quadruple $(W_R^c, W_R^m, W_R^b, v_R)$. While our experimental results rely on a simulator, hardware performance counters may also be used, if such counters are available.

Phase 3: Region Selection – The choice of program regions is implementation dependent. A program region can be defined as small as a basic block or as large as a procedure body. While program regions of small granularity may expose more opportunities for energy reduction, the large amount of them may prohibit the solver to find the optimal δ assignment in an efficient way. Our current implementation assumes a program region to be of single entry and single exit. For the multiple-region strategy, only perfect loop nests are considered as the program regions. For the single-region strategy, since combined regions are also taken into account, our implementation considers loop nests, call sites, `if`-statements, and sequence of regions as program regions.

As part of the third phase, the multiple-region strategy is required to solve a MINLP problem (P). Currently, we rely on a MINLP solver, namely `MINLP` [14], on the NEOS server [6] to solve the problem. Since the solver will run for a long time if the problem size is too large, we adopted various techniques to either reformulate the problem or cut down the problem size by approximation. In addition, the profiling phase only records the quadruple of each region but not the transitions between regions. An reaching definition analysis sub-phase has been implemented to estimate the values of v_{ij}. More details are discussed in the following section.

For single-region strategy, a pre-analysis is needed to compute the quadruple of each combined region since it is not profiled. Once all the quadruples of candidate regions are available, the enumeration process begins, to find the δ

assignment (or the region) that has the smallest objective function value E is problem (P_s). The details of the implementation can be found in [8].

Phase 4: Code Generation – Finally, the speed-setting instructions are placed at appropriate program locations. For the single-region strategy, the entry and exit of the slowed-down region are "guarded" with these instructions. The speed at entry of the region is set according to the slowdown factor. At exit, the speed is resumed to the original, non-scaled frequency. For the multiple-region strategy, only the entry of a region is considered as a candidate location to insert the speed-setting instruction. The region is "guarded" if there is an immediately preceding region that has a different slowdown factor.

3.1 Estimate the Transition Graph

For the multiple-region strategy, the information about the number of transitions between regions v_{ij} is needed. While it can certainly be done through edge/path profiling techniques (e.g. [3]), the current prototype implements a reaching definition analysis pass to estimate the values from the profiles of regions. The idea is to treat each region as an assignment to a global variable shared by all the regions. In doing so, reaching definition analysis captures the control flows between basic regions. The number of transitions between two regions is then determined by the minimum of the number of visits for both regions, i.e., $v_{ij} = \min(v_i, v_j)$.

This analysis may over-estimate the values of v_{ij}'s. For example, the analysis result of transitions between regions in Figure 1 derives the following transition graph:

$$R_1 \xrightarrow{1} R_2,\ R_2 \xrightarrow{10} R_3,\ R_3 \xrightarrow{1} R_4,\ R_3 \xrightarrow{1} R_5,$$
$$R_3 \xrightarrow{8} R_6,\ R_4 \xrightarrow{1} R_5,\ R_4 \xrightarrow{1} R_6,\ R_5 \xrightarrow{1} R_2,\ R_6 \xrightarrow{8} R_2.$$

Comparing with the real execution behavior, the estimated transition graph introduces two unrealizable transitions $R_4 \xrightarrow{1} R_5$ and $R_4 \xrightarrow{1} R_6$. A more precise, possibly more costly, analysis to estimate v_{ij}'s is to solve it as a network flow problem, i.e., the total incoming flows are always equal to the total outgoing flows. In this case, the extra transitions introduced by our analysis will not make a big impact because R_4 will be executed only once. In general, the analysis may under-estimate the potential energy reduction using the multiple-region strategy.

3.2 Reformulate the Problem

Problem (P) is difficult to be described in modeling languages such as AMPL and GAMS, which is in turn needed by the MINLP solver. To eliminate the maximum function and the binary function $\theta(\cdot,\cdot)$, problem (P) is reformulated as follows:

(P') minimize $E = \frac{1}{W} \cdot \sum_i W_i/\delta_i^2$
subject to
$$\sum_i (W_i^c \cdot \delta_i + z_i) + s \cdot \sum_{i,j} v_{ij} \cdot \theta_{ij} \leq (1+r) \cdot W$$

$$
\begin{aligned}
& z_i \geq \delta_i \cdot W_i^b,\ z_i \geq W_i^b + W_i^m \\
& \theta_{ij} \cdot (1-u) \leq \delta_i - \delta_j \leq \theta_{ij} \cdot (u-1) \\
& 1 \leq \delta_i \leq u \\
& \delta_i \in \mathbb{R},\ \theta_{ij} \in \{0,1\}
\end{aligned}
$$

Variables z_i and θ_{ij} are introduced to model the results of the maximum function $z_i = \max(\delta_i \cdot W_i^b, W_i^b + W_i^m)$ and the binary function $\theta_{ij} = \theta(\delta_i, \delta_j)$, respectively. The upper bound of a slowdown factor, u, is also introduced to support specifying θ_{ij}. In practice, this upper bound always exists from the hardware features. It is defined to be $u = 5$ throughout the paper.

The size of problem (P') can be characterized by a pair (n, m) that specifies a transition graph of n regions and m transitions. MINLP problems are in general considered as extremely hard problems since they combine the numerical difficulties of nonlinear programming with the combinatorial aspect of integer programming. Experiences tell us that when the number of transitions m exceeds over 50, the solver has a hard time to solve it efficiently. As a result, various techniques have been applied, if necessary, to reduce the problem size. In particular, two techniques have been used to identify large v_{ij}'s and enforce regions R_i and R_j to be of the same slowdown factor.

3.3 Other Issues

There are a couple of other issues involved in the design of the multiple-region strategy, for example,

1. How to model the δ's of regions that behave like the wild card? Our formulation suggests that each region will have a specific δ value. Alternatively, regions can be left "open" without any specific δ assignment.
2. How to model the "conflict" between certain regions? For example, once a procedure has δ assignment for regions inside it, it does not make sense to assign δ's to all call sites to this procedure.

In our problem formulation for multiple-region strategy, all regions are assumed to be *independent*, i.e., none of regions can prohibit the rest from being slowed down. In practice, however, this may not be the case. Consider the C code in Figure 3. In this code, there are five regions $R_1 - R_5$ that can be slowed down, including two call sites R_2 and R_3 to the same procedure `h()`. Suppose the call sites R_2 and R_3 are chosen to be slowed down with slowdown factors δ_2 and δ_3, respectively, and $\delta_2 \neq \delta_3$. It does not make sense to assign slowdown factors to regions in procedure `h()`, i.e., R_4 and R_5. On the other hand, we may slow down regions in a procedure, and enforce all the calls to that procedure excluded from basic region candidates. In summary, there are "conflicts" between regions being slowdown candidates.

Our current implementation only considers the perfect loop nests without calls inside as the candidate regions. For Figure 3, it means that only regions R_4 and R_5 are considered. The conflictness can be modeled by introducing a control

```
      void f() {
R1        for(int i=0; i<n; i++)
R2            { h(); } }

      void g() {
R3        h(); }

      void h() {
R4        for(int i=0; i<n; i++) { /* code without calls */ }

R5        for(int i=0; i<n; i++) { /* code without calls */ } }
```

Fig. 3. A possible code structure that complicates the choices of basic regions. Regions $R_1 - R_5$ are *potential* candidates for our compiler strategy, but not all of them can be slowed down at the same time.

binary variable for each case that there are multiple call sites to a procedure, and is smoothly integrated into our multiple-region problem formulation. All these issues will be addressed in the future.

4 Experiments

The experimental setting is as follows. Six `SPECfp95` benchmarks were used as program inputs. The SimpleScalar out-of-order issue processor timing simulator [5] with memory hierarchy extensions and DVS extensions served as the underlying machine model. The transition costs, i.e., voltage switching overheads were modeled in the simulator. The training data sets (`train.in`) provided with the benchmarks distribution were used during the profiling phase of our compiler. To reduce the simulation time, the *reduced* reference data sets (`std.in`) developed by Burger [4], were used instead of the original reference data inputs. The user-specified relative performance penalty r was varied from 1% to 10%, in order to expose energy performance trade-offs. Finally, a simple analytical energy model was used to estimate the energy consumption of a program.

SimpleScalar provides a cycle-accurate simulation environment for a modern out-of-order superscalar processor with 5-stage pipelines and fairly accurate branch prediction mechanism. The memory extensions model the limitedness of non-blocking caches through finite miss status holding registers (MSHRs) [12]. Bus contention and arbitration at all levels are also taken into account. Table 1 gives the simulation parameters used in the experiments.

The DVS extensions introduce a new speed-setting instruction. The speed setting instruction takes as argument an integer that specifies the desired CPU frequency. Its semantics was implemented in the following way: (1) stop fetching new instructions and wait until CPU enters the *ready* state, i.e., the speed setting instruction is not speculative, the pipeline is drained, all functional units are idle,

Table 1. System simulation parameters.

Simulation parameters	Value
fetch width	4 instructions/cycle
decode width	4 instructions/cycle
issue width	4 instructions/cycle, out-of-order
commit width	4 instructions/cycle
RUU size	64 instructions
LSQ size	32 instructions
FUs	4 intALUs, 1 intMULT, 4 fpALUs, 1 fpMULT, 2 memports
branch predictor	gshare, 17-bit wide history
L1 D-cache	32KB, 1024-set, direct-mapped, 32-byte blocks, LRU, 1-cycle hit, 8 MSHRs, 4 targets
L1 I-cache	as above
L1/L2 bus	256-bit wide, 1-cycle access, 1-cycle arbitration
L2 cache	512KB, 8192-set, direct-mapped, 64-byte blocks, LRU, 10-cycle hit, 8 MSHRs, 4 targets
L2/mem bus	128-bit wide, 4-cycle access, 1-cycle arbitration
memory	100-cycle hit, single bank, 64-byte/access
TLBs	128-entry, 4096-byte page
compiler	gcc 2.7.2.3 -O3

and all pending memory requests are satisfied, (2) wait a fixed amount of cycles to model the process of scaling up/down to the new frequency, and (3) resume the course using the new frequency. Each step has an associated performance penalty. In the simulation we set the step (2) cost as 10,000 cycles (10 μs for a 1GHz processor) if the desired CPU frequency is different from the current one, and zero otherwise.

For our profile-based compilation strategy, it was assumed that the underlying machine provides a *marker* instruction. A marker instruction takes as argument an integer that specifies the marker value. When it is executed, the hardware starts to collect the quadruple for the associated marker value. At any given cycle, only one marker value is alive.

Due to long simulation times, a simple analytical energy model was used to estimate the energy consumption of an entire program execution. It models total CPU energy usage, including both active and idle CPU cycles. The model is based on associating with each CPU cycle an energy cost. Specifically, given a program in which region R is slowed down by δ, the total CPU energy E is defined as:

$$E = \underbrace{(W^c + W^b) - (1 - 1/\delta^2) \cdot (W^c_R + W^b_R)}_{E_1} + \underbrace{\rho \cdot W^m}_{E_2}$$

where E_1 and E_2 models the CPU energy consumed by active cycles and idle cycles, respectively. In our experiments, an idle cycle was assumed to consume

30% of the energy cost of an active cycle, i.e., $\rho = 30\%$. It accounts for the energy consumption of clocked components such as clock tree [10].

Finally, we assume that the operating system uses the following simple formula to determine the appropriate CPU frequency $f(\delta)$ from a compiler-provided slowdown factor δ:

$$f(\delta) \stackrel{\text{def}}{=} \lceil \frac{f_{\text{peak}}}{l_{\text{mem}} \cdot \delta} \rceil \cdot l_{\text{mem}}$$

where f_{peak} is the peak CPU frequency and l_{mem} is the memory latency in peak CPU cycles. The reason l_{mem} is involved in the speed setting is because we had observed the clock skew effects due to mismatch of the memory and CPU cycle times [9]. This simple formula guarantees that the selected frequency is a multiple of memory latency. In our experiments, f_{peak} was set to be 1000 and l_{mem} was set to be 100. Since the compiler sets a limit on the lowest CPU frequency to be used (in terms of u in problem (P')), it amounts to say that we considered a DVS system whose CPU frequency ranges from 200 MHz to 1000 MHz with discrete frequency/voltage levels.

The compilation time for both single-region strategy and multiple-region strategy is in the order of minutes. This does not include the time needed to perform the profiling (phase 2) which may take up to several hours. We are currently investigating compile-time models to derive the information generated by phase 2. Table 2 lists the more detailed timings of various phases for both strategies.

Table 2. The compilation time (in seconds) of both single-region strategy and multiple-region strategy for $r = 1\%$.

	phase 1	phase 2	phase 3&4	
			single	multiple
swim	4	6452	8	16
tomcatv	3	95591	5	9
mgrid	6	96138	11	17
turb3d	19	120721	1313	79
applu	48	4757	88	95
apsi	64	4317	572	244

The user-provided performance tolerance ratio r defines a *soft* deadline, i.e., in some cases, the real performance may exceed this limit. As a result, both single-region and multiple-region strategies have different sets of performance (T) and energy consumption (E) for the same r value. One way to compare them is in terms of the *energy-delay product* ($E \cdot T$) [7]. Figure 4 lists the energy-delay product for six `SPECfp95` benchmarks for various r values. It can be seen that, except for benchmark `swim`, both strategies are very similar to each other in the energy-delay product. In other words, when the multiple-region strategy results in more performance degradation than the single-region strategy, it is

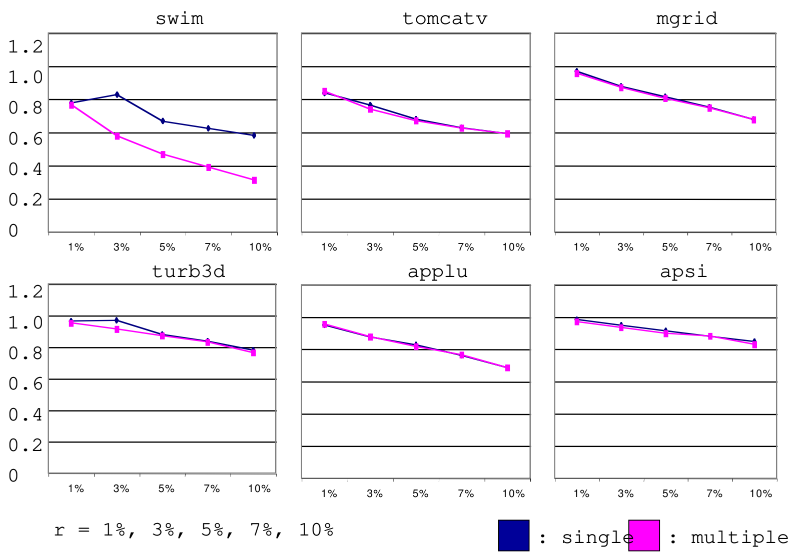


Fig. 4. The comparison of single-region vs. multiple-region approaches for various `SPECfp95` benchmarks and user-defined performance penalty. The energy-delay product is represented relatively with respect to the original program running at full speed.

able to cut down more energy consumption to compensate for the additional performance loss. For benchmark `swim`, the multiple-region strategy is obviously more beneficial than the single-region strategy.

5 Related Work

The closest work we are aware of is the intra-task DVS techniques using checkpoints. Intra-task scheduling is based on the reclamation of the slacks deviated from the compile-time (over-)estimation such as the worst case execution time (WCET). Checkpoints are inserted into the original program at compile time to indicate where the CPU speed (and voltage) should be re-calculated. While more checkpoints allow finer performance tuning, the accumulated overheads of performing CPU re-calculation may become significant and reverses the improvement into degradation. One important issue is then *where* to insert these checkpoints.

Lee and Sakurai in [13] placed checkpoints at the equally sized time slots of the WCET of a task. In [22, 21], Shin et al. put the checkpoints at selected CFG edges of a task to capture the slacks from run-time variations of different execution paths. Mossé et al. proposed to insert checkpoints at boundaries of program constructs such as loops and call sites in [17]. In a follow-up paper [1], they assumed equally spaced checkpoints in time and tried to determine the optimal amount. Azevedo et al. in [2] placed checkpoints at every branch, loop and call site initially, and then used the profile information to guide pruning some of the checkpoints.

The most significant difference between our work and the above is that we identify a *different* type of CPU slacks. We focus on memory-bound regions, while others consider the run-time variations from the estimation. Our work also takes into account the overheads induced by the checkpoints explicitly to prevent from over-doing. These checkpoints are defined at boundaries of program constructs in the source program, rather than points in the time line, which we feel more appropriate to the compiler-directed dynamic voltage scheduling.

Some of the task-based algorithms formulated the dynamic voltage scheduling as a (mixed-)integer linear/nonlinear programming problem. For example, Ishihara and Yasuura in [11] gave an ILP formulation for a set of tasks and a set of discrete voltage levels. In contrast, the paper by Manzak and Chakrabarti [16] assumed continuous voltages and a single voltage/frequency for a task. Raje and Sarrafzadeh [20] formulated the problem for an acyclic task graph and discrete voltage levels. None of the above took into account the transition costs. Swaminathan and Chakrabarty in [25] incorporated transition costs into the problem formulation. They assumed a single dual-speed CPU executes a set of periodic non-preemptive real-time tasks.

Our work is different in at least the analytical performance model. Most of the work assumes pure CPU processing time (i.e., $W^m = W^b = 0$). In contrast, our model breaks down the total execution time into three parts with respect to the memory system. While our model estimates more precisely the performance impact of dynamic voltage scaling on a program, it makes the estimation much harder due to the non-continuity of the maximum function. In addition, the problem formulation in this paper assumes continuous performance levels in terms of the slowdown factors. We expect that the formulation with discrete performance levels is cleaner (the binary function $\theta(\cdot,\cdot)$ is "inlined") and may be easier to solve. However, it remains to be seen how much difference the accuracy of the performance model and the difficulty of the problem formulation would make for the real energy-performance trade-offs.

6 Conclusions

Compile-time directed frequency and voltage scaling is an effective technique to reduce CPU power dissipation. We have discussed a trace-based compiler approach that identifies regions in the program that can be slowed down without significant performance penalties. Two strategies were implemented within `SUIF2`, one that selects a single program regions for CPU slow down (single region approach), and one that allows multiple regions to be executed at different CPU frequencies. Based on cycle accurate simulation using the SimpleScalar tool set, the resulting energy delay products, and the energy savings due to voltage scaling were comparable for five out of our six SPECfp95 benchmark programs. Both approaches achieved energy savings of up to 48%, with performance penalties of up to 16%. For the remaining benchmark program, the multiple regions approach was significantly better in terms of energy-delay product, yielding energy savings of up to 74% vs. 50% for the single region approach, with maximal performance penalties of 21% vs. 17%, respectively.

Although these results are very encouraging, more work needs to be done to improve the compilation efficiency of the single and multiple regions approaches. For some benchmark programs, the MINLP solver used in the multiple regions compiler was not able to compute the optimal solution within a reasonable time. We therefore were not able to include these numbers in this study. We are currently investigating fast approximation strategies that will only insignificantly degrade the quality of the determined voltage schedules.

References

1. N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.
2. A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints in the COPPER framework. In *Proceedings of Design, Automation and Test in Europe Conference*, March 2002.
3. T. Ball and J. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 31(7), 2000.
4. D. Burger. *Hardware Techniques to Improve the Performance of the Processor/Memory Interface*. PhD thesis, Computer Science Department, University of Wisconsin-Madison, 1998.
5. D. Burger and T. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison, June 1997.
6. J. Czyzyk, M. Mesnier, and J. Moré. The NEOS server. *IEEE Journal on Computational Science and Engineering*, 5(3), July–September 1998.
7. R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.
8. 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, November 2001.
9. C.-H. Hsu, M. Hsiao, and U. Kremer. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems*, November 2000.
10. M. Irwin. Low power design: From soup to nuts. Tutorial at the *International Symposium on Computer Architecture*, June 2000.
11. T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design*, pages 197–202, August 1998.
12. D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 81–87, May 1981.
13. S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation*, pages 806–809, June 2000.

14. S. Leyffer. Integrating SQP and branch-and-bound for mixed integer nonlinear programming. *Computational Optimization and Applications*, 18(3):295–309, March 2001.
15. J. Lorch and A. Smith. Improving dynamic voltage algorithms with PACE. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2001.
16. A. Manzak and C. Chakrabarti. Variable voltage task scheduling algorithms for minimizing energy. In *Proceedings of the International Symposium on Low-Power Electronics and Design*, August 2001.
17. 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*, October 2000.
18. T. Mudge. Power: A first class design constraint for future architectures. In *Proceedings of International Conference on High Performance Computing*, December 2000.
19. T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.
20. S. Raje and M. Sarrafzadeh. Variable voltage scheduling. In *International Symposium on Low Power Electronics and Design*, pages 9–14, August 1995.
21. D. Shin and J. Kim. A profile-based energy-efficient intra-task voltage scheduling algorithm for hard real-time applications. In *Proceedings of the International Symposium on Low-Power Electronics and Design*, August 2001.
22. 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), March/April 2001.
23. P. Stanley-Marbell, M. Hsiao, and U. Kremer. A hardware architecture for dynamic performance and energy adaptation. In *Workshop on Power-Aware Computer Systems*, February 2002.
24. SUIF. Stanford University Intermediate Format.
25. 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*, January/February 2001.