Compiler-Directed Dynamic Voltage Scaling Based on Program Regions

Chung-Hsing Hsu and Ulrich Kremer [*]
Department of Computer Science
Rutgers University

DCS-TR-461

November 2001

ABSTRACT

This paper discusses the design and implementation of the first compiler that optimizes programs for power and energy using dynamic voltage scaling. The compiler identifies program regions where the CPU can be slowed down without resulting in a significant overall performance loss. For such regions the lowest CPU voltage is selected that operates correctly under the reduced clock frequency. Our trace-based compiler prototype uses the SUIF2 compiler infrastructure. For the `SPECfp95` benchmark, simulation results show energy savings of up to 24% with performance penalties of less than 2.7%.

1. INTRODUCTION

With the advances in technology, power is becoming a first-class architecture design constraint not only for embedded and portable devices but also for high-end computer systems [23]. Minimizing the power/energy dissipation of programs leads to prolong battery lifetime and reduce heat dissipation and cooling requirements, which in turn reduces design, packaging, and operation costs, including power bills for air conditioning of computing and data centers.

Among the various techniques to address power/energy concerns, Dynamic voltage scaling (DVS) has been identified as one of the most effective ways to reduce power dissipation. Dynamic voltage scaling is a technique that varies the supply voltage and CPU frequency to provide desired performance with the minimum amount of energy consumption.

In the paper we follow the system model by Burd and Brodersen at UC Berkeley [3]. Their DVS system has three key components: an operating system that can vary the processor speed, a regulation loop that can generate the minimum voltage required for the specified speed, and a microprocessor that can operate over a wide voltage range. The operating system controls the processor speed by writing the desired clock frequency into a system control register. The register's value is then used by the regulation loop to adjust the CPU clock frequency with the corresponding minimum voltage level. For simplicity, we assume that all the components of the processor are driven by this single voltage. Finally, we assume that an application has direct control of the processor speed and voltage through explicit system calls or access to appropriate machine registers. This is a reasonable assumption in a single user environment, but not in a multi-user environment where the CPU is scheduled across different active processes. In a multi-user environment, the DVS instructions inserted by the compiler can be considered hints that the operating system can use to negotiate frequency and voltage requests across the set of active processes. In this paper, we assume a single user environment.

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

2. BASIC COMPILATION STRATEGY

Our compilation strategy tries to find memory-bound program regions where a possible CPU slowdown will not significantly affect the overall execution time, i.e., performance of the region. Target architectures are systems that allow overlap of CPU and memory hierarchy activities. The basic idea is to "hide" the degraded CPU performance behind the memory hierarchy accesses which are on the critical path. The compiler identifies program regions as candidates where the CPU may be slowed down. Within such regions, a fixed voltage and frequency will be selected by the compiler, and dynamic frequency and voltage changes may only occur between regions. A candidate regions is either a basic region, or a set of candidate regions that can be combined to form a single entry and single exit program region. A basic region is a loop nest, or a procedure/function call. The following model captures the performance impact of a selected CPU slowdown factor δ for a candidate region R [11]. Specifically, if the CPU frequency is reduced by a factor of δ, $\delta \geq 1$, the performance of the region will become

$$T(\delta, R) = \delta * W_R^c + \max(\delta \cdot W_R^b, W_R^b + W_R^m) \qquad (1)$$

where the total workload for the region W_R (in cycles) is decomposed into three parts $W_R = W_R^c + W_R^b + W_R^m$, in which

- W_R^c is the number of cycles in the region that the CPU is busy while the memory is idle (cpu_busy); this in-

```
(1) Enumerate all candidate regions $R$ in the program:
    (a) Determine $W_R^C$, $W_R^m$, $W_R^b$, $v$ for the region.
    (b) Compute the slowdown factor $\delta$:
            if $(r \cdot W - 2 \cdot v \cdot s) \cdot W_R^b \geq W_R^c \cdot W_R^m$ then
                $\delta = \max(1, (r \cdot W + W_R - 2 \cdot v \cdot s)/(W_R^c + W_R^b)$
            else
                $\delta = \max(1, (r \cdot W + W_R^c - 2 \cdot v \cdot s)/W_R^c)$
    (c) Estimate the relative program energy consumption $E$:
            $E = [(W - W_R) + W_R/\delta^2]/W$
(2) Find the region $R^{best}$ that has the lowest $E$.
(3) Insert voltage/frequency setting instruction according
    to region $R^{best}$ and its slowdown factor.
```

Figure 1: Basic compilation strategy to select a single region for DVS.

cludes CPU pipeline stalls due to hazards and activities of both level one cache and level two cache,

- W_R^m is the number of cycles in the region that the CPU is stalled while waiting for data from memory (*m*emory_busy),
- W_R^b is the number of cycles in the region that both CPU and memory are active at the same time (*b*oth_busy).

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

The following basic compilation strategy finds a single candidate region that maximizes the expected energy savings. Issues related to an extension of this strategy to multi-regions will be discussed in Section 5. Figure 1 shows our single region strategy that essentially enumerates and evaluates all candidate regions to find the region that solves the following problem:

$$\text{minimize } E = \frac{1}{W} \cdot [(W - W_R) + \frac{1}{\delta^2} W_R]$$

subject to

$$T(\delta, R) + (W - W_R) + 2 \cdot v \cdot s \leq (1 + r) \cdot W \text{ and } 1 \leq \delta$$

where E is the relative energy consumption (in percentage) compared to the case where full speed is used, W is the workload for the entire program, $T(\delta, R)$ is the performance of the region after the slowdown, $(W - W_R)$ is the part of the program unaffected by the slowdown, $2 \cdot v$ is the number of times performing voltage scaling, s is the performance cost of a single voltage scaling (voltage scaling overhead), and $(1 + r) \cdot W$ sets up the (soft) deadline.

The current implementation of our basic compilation strategy consists of intraprocedural and interprocedural analysis passes. An exhaustive enumeration of all candidate regions is performed. The values for W_R^c, W_R^m, W_R^b, and v are determined through program profiling for a characteristic input

```
R1          CALL INITAL
            NCYCLE=0
      90    NCYCLE=NCYCLE+1
R2          CALL CALC1
R3          CALL CALC2
            IF (MOD(NCYCLE,MPRINT) ≠ 0) GO TO 370
R4          /* a perfect loop nest */
     370    IF (NCYCLE >= ITMAX) STOP
            IF  (NCYCLE <= 1) THEN
R5              CALL CALC3Z
            ELSE
R6              CALL CALC3
            ENDIF
            GO TO 90
```

Figure 2: Main program of the SPECfp95 `swim` code.

data set. Work on compile-time models for these entities is underway. A detailed description of our prototype is given in Section 3.

In the remainder of this section, the basic compilation strategy is illustrated for the main program of the `SPECfp95` benchmark program `swim`. The main program is shown in Figure 2, There are six "basic" regions R_1 - R_6, and three combined candidate regions $R_{2\&3}$, $R_{5\&6}$, and R_{1-6}. Region $R_{2\&3}$ represents two consecutive procedure calls, and region $R_{5\&6}$ is the if-then-else construct that encloses regions R_5 and R_6. The region R_{1-6} is the entire program. Note that R_1 and R_2, or regions R_4 and $R_{5\&6}$ are not combined into a single candidate region due to (potential) intermediate control flow.

Table 1 shows the profiling results using the `SPECfp95` train input set and a superscalar, out-of-order issue architecture described in Figure 5. The compilation strategy sketched in Figure 1 enumerated all nine regions for the main procedure. Based on a user provided maximal performance penalty of 1% (r = 0.01), and an assumed voltage scaling overhead of 10,000 cycles, the compiler found that the combined region $R_{5\&6}$ can be slowed down by a factor of $\delta = 2.0665$ and gives the best energy saving under the soft deadline constraint of 1% performance degradation. Experimental results showed that in fact this selection is the best choice, with energy savings of 24% over the unoptimized code, and a performance penalty of 2.67%. Experimental results are discussed in detail in Section 4.1.

3. IMPLEMENTATION

The prototype for our profile-driven single-region compiler strategy is implemented as part of the SUIF2 compiler infrastructure [32]. The prototype has five phases. The first phase instruments the original C program at appropriate locations. The instrumented code is then executed (second phase), collecting information needed in the following phases. The third phase uses both the instrumented program and the profile information to determine the run-time behavior of all possible combined candidate regions. As the fourth phase, all the regions are enumerated to select the one region that minimizes the energy model. Once this region is identified, the final phase places speed setting instructions at the boundaries of the selected region, and restores the instrumented program back to the original one. The resulting

Table 1: $W = 1068$ million cycles, $s = 10000$, and $r = 0.01$. We assume the total workload is equal to $\sum_i W_i$ where R_i's are basic regions. Workloads are given in million cycles.

R_i	W_i^c	W_i^m	W_i^b	v	δ	E
basic regions						
R_1	116	27	8	1	1.09	0.98
R_2	52	200	54	10	1.20	0.91
R_3	59	207	38	10	1.18	0.92
R_4	2.8	4.4	1.2	1	4.75	0.99
R_5	0.06	28	2.7	1	14.89	0.97
R_6	9.8	201	52	8	2.08	0.81
combined candidate regions						
$R_{2\&3}$	112	408	92	10	1.09	0.91
$R_{5\&6}$	9.8	229	54	9	2.07	0.79
R_{1-6}	242	669	157	1	1.04	0.92

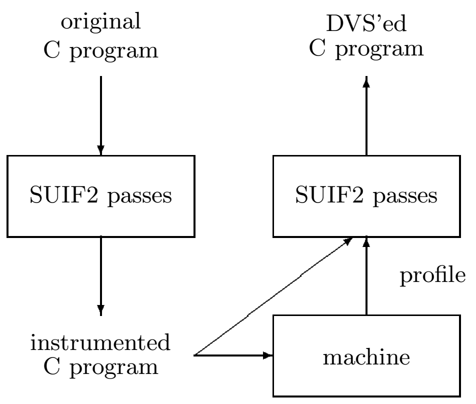

Figure 3: The flow diagram of the compiler implementation.

program is the original input program with a few additional DVS instructions. Figure 3 shows the phases of the implementation.

Two kinds of program constructs are instrumented in our implementation, namely call sites and explicit loop structures. Explicit loop structures include `for` and `while` loops. Loops based on `goto`'s are not instrumented and will not be considered as candidate regions in the current implementation.

The information collected for each instrumented program construct R includes the number of cycles that only the CPU is busy (W_R^c), only the main memory is busy (W_R^m), and both of them are busy (W_R^b), and the number of visits v_R. While our experimental results rely on a simulator, hardware performance counters may also be used, if such counters are available.

The instrumented program constructs are only a subset of candidate regions considered by the selection phase. All others are called *combined* candidate regions, in that their run-time behavior, in terms of W_R^c, W_R^m, W_R^b, and v_R, is approximated by the characteristics of the combined and instrumented regions that it contains. Figure 4 specifies how these values are computed.

To compute the run-time behavior of combined candidate

`if` statement:
R: **if** () **then** R_1 **else** R_2
$$W_R^i = W_{R_1}^i + W_{R_2}^i \text{ for } i \in \{c, m, b\}$$
$$v_R = v_{R_1} + v_{R_2}$$

explicit loop structure:
R: **loop** () R_1
$$W_R^i = W_{R_1}^i \text{ for } i \in \{c, m, b\}$$
v_R is profiled

call site:
R: **call** f()
$$W_R^i = (v_R / v_f) \cdot W_f^i \text{ for } i \in \{c, m, b\}$$
v_R is profiled

sequence of regions:
R: **sequence**($R_1, \ldots, R_n$)
$$W_R^i = \sum\{W_{R_j}^i :\ 1 \le j \le n\} \text{ for } i \in \{c, m, b\}$$
$$v_R = v_{R_1} = \ldots = v_{R_n}$$

procedure: f: **procedure** f() R
$$W_f^i = W_R^i \text{ for } i \in \{c, m, b\}$$
$$v_f = \sum\{v_j :\ j \text{ is a call site to } f()\}$$

Figure 4: The description of how the run-time behavior, in terms of, W_R^c, W_R^m, W_R^b, and v_R, of a combined region is computed.

regions, an interprocedural program analysis pass is implemented. The pass traverses all procedures, i.e., all nodes in the program call multi-graph (each call site is represented by an edge) in reverse topological order. The current implementation assumes acyclic call graphs, i.e., assumes no recursion. For each visited procedure, the annotated abstract syntax tree (AST), embedded in the SUIF2 intermediate format, is traversed in a bottom-up fashion, using the instrumented regions as leaf nodes.

To improve the efficiency, only AST nodes representing `if` statements, explicit loop structures, call sites, and procedures reachable from the main routine are annotated. The run-time behavior of combined regions defined as sequences of regions is computed "on the fly" in the selection phase. Currently, only regions in a *forward* sequencing manner are taken into account. In other words, for program construct

$$\textbf{loop}()\ \textbf{sequence}(R_1, \ldots, R_n)$$

the regions composed of $R_i \rightarrow R_{i+1} \rightarrow \ldots \rightarrow R_j$ for $j \geq i$ are considered, but not the "wrap-around" regions, $R_i \rightarrow \ldots \rightarrow R_n \rightarrow R_1 \rightarrow \ldots \rightarrow R_j$ for $j < i$. To evaluate "wrap-around" regions, loops induced by `goto`'s need to identified, which is still under development.

The selection phase enumerates all possible candidate regions, and tries to locate the one region that minimizes the energy model. The location of the selected region, along with its slowdown factor, is then passed on to the final code generation phase. Our implementation inserts speed-setting instructions at the entry and exit of the selected region. The speed at entry is set according to the slowdown factor, and at exit the speed is resumed to the original, i.e., non-scaled frequency.

4. EXPERIMENTS

In the experiments, we used the same setting as in [12] in order to be able to compare our two approaches. Specifically, `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 voltage scaling overheads were modeled. The training data sets (`train.in`) provided with the `SPECfp95` benchmarks distribution were used during the profiling phase of our compiler. As consistent with [12], the user-specified relative performance penalty r was set to 1%. This is a rather ambitious goal, since it eliminates the possibility of trading off performance for energy savings. To reduce the simulation time, we used the *reduced* reference data sets, a.k.a, the standard data sets (`std.in`) developed by Burger [4], instead of the original reference data inputs. A simple analytical energy model was used to estimate the energy consumption of a program.

The standard data sets were designed to have as few executed instructions as possible while retaining the behavior of the reference data sets. In most cases, the number of loop iterations in the reference input files was reduced so that the initialization only accounts for no more than 10% of all executed instructions. There are a few exceptions where even at 10% the program running time is still too long. In these cases, 10% is relaxed to be 20% instead. The benchmarks in this category include `tomcatv`, `su2cor`, `apsi`, and `wave5`. Burger claimed that benchmark `mgrid` has similar behavior using either test input or reference input, and replaced the reference input file with the test input file. Finally, benchmark `fpppp` was taken off from the experiments since it is extremely CPU bound, leaving no room for CPU slowdown. Table 2 gives the definitions of `std.in` for the `SPECfp95` benchmark programs.

Table 2: Different execution characteristics of data sets for `SPECfp95` benchmark programs. The reference data sets (`ref.in`) and the training data sets (`train.in`) are part of official `SPECfp95` benchmark distribution. The standard data sets (`std.in`) were designed by D. Burger as the "reduced" reference data sets. The leftmost two columns compare `ref.in` and `std.in` according to the type of data inputs used and the number of loop iterations specified in the input files. The rightmost two columns are extracted from Burger's Ph.D. thesis to compare the number of executed instructions for `train.in` and `std.in`.

benchmark	`ref.in`	`std.in`	`train.in`	`std.in`
tomcatv	ref750it	ref62it	17660.7	10651.7
swim	ref900it	ref45it	849.9	2846.2
su2cor	ref40it	ref5it	19851.1	11548.7
hydro2d	ref200it	ref6it	7583.0	2443.1
mgrid	ref40it	test4it	14292.1	480.3
applu	ref300it	ref5it	531.9	1748.1
turb3d	ref111it	ref2it	17120.6	2836.8
apsi	ref960it	ref6it	2350.0	318.2
wave5	ref40it	ref10it	3132.8	13072.9

All simulations were done through the SimpleScalar tool set [5], with memory hierarchy extensions and our DVS extensions. 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. While the original version supports a multi-level non-blocking memory subsystem and captures limited memory bandwidth, the extensions model the limitedness of non-blocking caches through finite miss status holding registers (MSHRs) [17]. Bus contention and arbitration at all levels are also taken into account. What is not considered are multi-bank memory organization, page hits versus misses, precharging overhead, and refresh cycles. Figure 5 gives the simulation parameters used in the experiments.

The DVS extensions introduce a new speed-setting instruction in SimpleScalar's ISA. 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, 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).

Due to the 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

Simulation parameters	Value
frequency	1 GHz
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
TLBs	128-entry, 4096-byte page
compiler	`gcc 2.7.2.3 -O3 -funroll-loops`

Figure 5: System simulation parameters.

is slowed down by δ, the total CPU energy E is defined as:

$$\begin{aligned} E &= E_1 + E_2 \\ E_1 &= (W^c + W^b) - (1 - 1/\delta^2) \cdot (W_R^c + W_R^b) \\ E_2 &= \rho \cdot W^m \end{aligned}$$

where E_1 models the CPU energy consumed by active cycles, and E_2 models the CPU energy consumed by idle cycles. 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 [14].

Finally, 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 values of W_R^c, W_R^b, W_R^m, and v_R for the associated marker value. At any given cycle, only one marker value is alive. In addition, a simple formula was used to compute the desired CPU frequency (f) from a slowdown factor δ:

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

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 [11]. This simple formula guarantees that the selected frequency is a multiple of memory latency. In practice, the DVS system provides a set of discrete well-engineered CPU frequency/voltage levels that the compiler or the operating system can choose from. In our experiments, f_{peak} was set to be 1000 and l_{mem} was set to be 100. In other words, we considered a DVS system whose CPU frequency ranges from 1 MHz to 1 GHz with discrete frequency/voltage levels.

4.1 Experimental Results

In this section the experimental results of the profile-driven single-region compiler implementation are presented. We first compare the regions it selects with those selected by the previous greedy algorithm. Due to space limitation, only the results for four benchmarks are shown here, in Figure 6. The four benchmarks are the top four programs that have the best energy saving from the results of the greedy algorithm. In this way, the compiler implementation is "validated" so that it will not miss the opportunity as observed by the greedy algorithm. Furthermore, it is interesting to know whether there is other opportunity missed by the greedy algorithm but captured by our implementation.

For benchmarks `swim` and `applu`, both strategies selected the same region. However, the algorithm shown in this paper determines smaller δ values. This is because the algorithm takes into account the voltage scaling overheads, while the results in [12] does not. The regions selected by both strategies for benchmark `tomcatv` are almost identical. The results for benchmark `hydro2d` is interesting. The algorithm in [12] selected a smaller region with a higher slowdown factor. In contrast, our single-region algorithm picked up a larger region with a smaller δ value.

The simulation results of `SPECfp95` benchmarks are listed in Table 3. They are given in term of *relative* performance and energy usage with respect to the *different* baselines. The leftmost two columns are the results of the greedy algorithm, using `f2c`-mangled un-DVS'ed versions as the baseline. The rightmost two columns are the results of our single-region compiler implementation, using `f2c`-mangled and SUIF2-mangled un-DVS'ed versions as the baseline.

The baseline for the results of our compiler implementation has to take SUIF2 system into account for two reasons:

- The SUIF2 system automatically performs certain *inter-file* optimizations for an application consisting of multiple files, and, as a result, it may improve the resulting program performance.
- The SUIF2 system automatically transforms certain program constructs into low level representations that may prevent `gcc` from better optimizing the code, and, as a result, it may degrade the resulting program performance.

For example, benchmark `wave5` consists of multiple files. The SUIF2 system requires to link all the files together to perform inter-procedural program analysis if procedures reside in different files. Our implementation then generates a big *single* C file as the output, which is fed into the `gcc` compiler. This process resulted in the speed-up of 1.12, comparing SUIF-mangled version with the original version. Since, in general, energy reduction is correlated positively with performance improvement, the reported benefit is boasted in the sense that it cannot be attributed to the dynamic voltage scaling alone. For benchmark `wave5`, comparing with the baseline for the greedy algorithm, the energy consumption is reduced to 78.31%. It is much higher than 94.14% as reported in Table 3, and can be largely attributed to the single file structure imposed by the SUIF2 system. On the other hand, the execution time of benchmark `applu` increases to 112.15% due to SUIF2-mangleness. As a result, we use SUIF2-mangled programs as the baseline to evaluate the effectiveness of our compiler technique in program energy reduction.

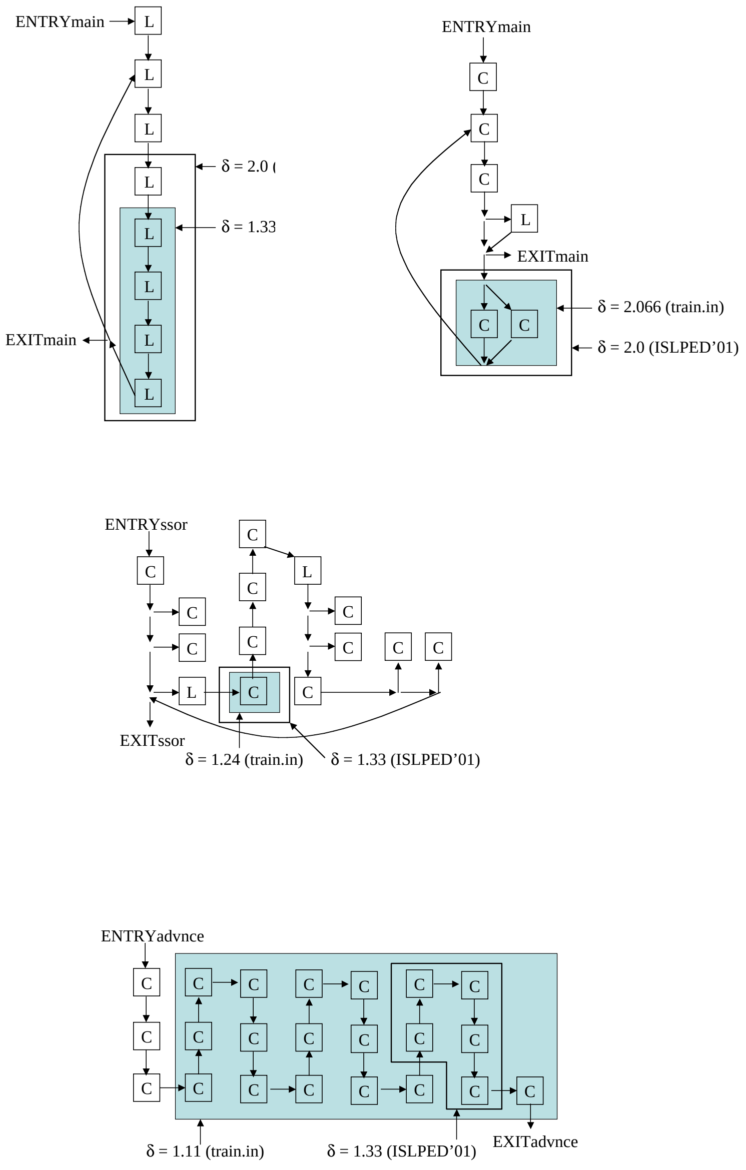

Figure 6: Comparison between hand-optimized and compiled versions of codes (from top left to bottom) `tomcatv`, `swim`, `applu`, and `hydro2d`. The shaded regions were selected by the compiler. The graph for each benchmark is a region-based control flow graph, where $\boxed{\text{C}}$ represents a call site and $\boxed{\text{L}}$ represents a loop nest.

Table 3: Experimental results for `SPECfp95`. All the results are represented in terms of the relative performance and energy consumption with respect to different baselines. The leftmost two columns are the results from the greedy algorithm, using the original `f2c`-translated versions as the baseline. The rightmost two columns are the results of our single-region compiler implementation, using `f2c`-mangled and SUIF2-mangled versions as the baseline.

strategy	[12] `std.in`		SUIF'ed `train.in`	
benchmark	T	E	T	E
tomcatv	101.99	76.25	100.48	83.49
swim	101.68	76.79	102.67	75.70
hydro2d	101.47	84.61	101.69	83.42
applu	101.82	90.43	101.22	93.94
su2cor	100.77	92.06	100.46	95.83
turb3d	101.52	92.83	101.65	94.93
mgrid	101.61	93.43	100.92	95.52
apsi	102.53	95.26	100.50	97.99
wave5	101.15	96.03	101.50	94.14

It can be seen from Table 3 that for most of the benchmarks the results of the two algorithms are very similar, except for `tomcatv`. The energy consumption introduced by our region-based algorithm only reduces to 86.24%. Comparing with 76.25% from the greedy algorithm, it is a large gap. The reason for the gap is because the instrumentation skewed the values of W_R^c, W_R^m, and W_R^b. Specifically, for the region selected in [12], its run-time behavior is $W_R^c = 1.76\%$, $W_R^m = 81.40\%$, and $W_R^b = 16.84\%$. In contrast, the behavior derived through the profile by our compiler implementation becomes $W_R^c = 5.44\%$, $W_R^m = 77.86\%$, and $W_R^b = 16.70\%$. While both strategies determined that region R takes about 40% of total execution time, the distributions in the region are different. More significantly, the W_R^c value derived by the implementation is much higher. As a result, it restricts the largest slowdown factor that the compiler implementation can choose. In comparison, 8 program locations were instrumented in [12], while 32 locations were instrumented by our compiler instrumentation pass. It suggests that our current implementation needs to be improved in such a way that fewer instrumentation codes are inserted so that the run-time characteristics is not significantly skewed.

4.2 The Impact of Different Training Inputs

Table 3 compares the greedy algorithm in [12] with the standard data sets (`std.in`) and the single-region compiler implementation with the training data sets (`train.in`). Since `train.in` may have different program behavior from `std.in`, in this section we evaluate how much impact the different inputs would make on our profile-driven implementation. Table 4 lists the relative performance and energy consumption of `SPECfp95` benchmarks using `train.in` and `std.in`. It can be seen that in most benchmarks, different training inputs do not make significant differences in impacting our compiler strategy. In two case, benchmarks `applu` and `su2cor`, `std.in` does guide more energy savings at the cost of higher performance penalty.

Table 4: The comparison of different training inputs to our profile-driven compiler implementation.

strategy	SUIF'ed `train.in`		SUIF'ed `std.in`	
benchmark	T	E	T	E
tomcatv	100.48	83.49	100.82	81.21
swim	102.67	75.70	101.72	77.52
hydro2d	101.69	83.42	101.88	81.70
applu	101.22	93.94	104.09	88.01
su2cor	100.46	95.83	102.00	89.48
turb3d	101.65	94.93	101.73	94.81
mgrid	100.92	95.52	101.39	94.93
apsi	100.50	97.99	101.32	93.30
wave5	101.50	94.14	101.26	94.74

Table 5: The comparison of single-region vs. multiple-region approaches in terms of E.

strategy	single region		multiple regions	
benchmark	T	E	T	E
tomcatv	100.48	83.49	101.02	84.04
swim	102.67	75.70	102.17	75.09
hydro2d	101.69	83.42	101.61	83.11
applu	101.22	93.94	100.85	95.44
turb3d	101.65	94.93	374.22	90.36
mgrid	100.92	95.52	101.22	94.67
apsi	100.50	97.99	197.01	96.75

5. EXTENSION TO MULTIPLE REGIONS

The algorithm can be extended to allow multiple regions to be slowed down. In other words, we are trying to solve the following problem:

$$\text{minimize } E = \frac{1}{W} \cdot \sum_i W_i/\delta_i^2$$

subject to

$$\textstyle\sum_i T(\delta_i, R_i) + s \cdot \sum_{i,j} v_{ij} \cdot z_{ij} \le (1+r) \cdot W \text{ and } 1 \le \delta$$

where

$$z_{ij} \overset{\text{def}}{=} \begin{cases} 0 & \text{if } \delta_i = \delta_j \\ 1 & \text{otherwise} \end{cases}$$

The newly introduced variables z_{ij} capture the possible transitions between regions. For example, the real execution of `swim` on training input indicates the following transitions for Figure 2.

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

$$R_5 \xrightarrow{1} R_2,\ R_6 \xrightarrow{8} R_2.$$

The weight of each edge is v_{ij}, the number of visits. If the total transition cost $\sum_{i,j} v_{ij} \cdot z_{ij}$ is dropped out from the problem formulation, the lower bound *with respect to the current set of basic regions* can be derived. For the `swim` example, the optimal solution is $E = 0.773$ with the following δ assignment.

$$\delta_1 = 1,\ \delta_2 = 1.0262,\ \delta_3 = 1.0262,\ \delta_4 = 1,\ \delta_5 = 4.5796, \text{ and } \delta_6 = 1.7467.$$

Comparing with $E = 0.7894$ derived from the single-region strategy, multiple-region strategy does not seem to improve much in this case.

There are a couple of issues involved in the design of multiple region approach, for example,

1. How to estimate the transitions z_{ij}, and what is a sufficient level of accuracy for the algorithm ?

2. How to model the δ's of regions that behave like the wildcard? Our formulation suggests that each region will have a specific δ value. Alternatively, regions can be left "open" without any specific δ assignment.

3. How do we 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.

4. How can we get the optimal solution and compute it fast?

We have an initial implementation that tries to address all the above question. To be brief, a region-oriented reaching definition analysis was performed to estimate the transitions between regions. For the third issue, we can reformulate the problem by introducing control variable for each case. However, it is not implemented yet. Solving the above mixed integer nonlinear programming (MINLP) problem is not an easy task. For `SPECfp95` benchmarks, the number of regions derived by our implementation ranges from 9 to 98, and the number of transitions estimated by our implementation is between 10 to 741. Most of MINLP solvers on the web fail to find the optimal solution when the size gets to over 50. As a result, we implemented a program analysis that intends to reduce the number of regions and the number of transitions, if necessary. Table 5 lists our initial results. Only benchmarks `tomcatv`, `swim`, and `mgrid` were able to get the optimal solutions. The results do not show that multiple-region algorithm has a much better savings in energy than the single-region algorithm. We are currently investigating whether this insignificance is due to the implementation or the inherent characteristics of the programs.

6. RELATED WORK

Microprocessors that support dynamic voltage scaling have been available, such as Transmeta's Crusoe, Intel's Xscale, AMD's K6-IIIE+, and academic prototypes [3, 26]. Current implementations have 200-800 MHz CPU core and take about 75-520 microseconds for a single transition. With such high transition cost, the frequency of apply DVS to the program need to be taken extreme care.

There have been many proposals on how to effectively use a DVS microprocessor to reduce energy consumption. Most of them are implemented at operating system level, and can be classified as either *task*-based or *task*-based. An interval-based algorithm divides time into fixed-length intervals, predicts the workload of the current interval using past information, and sets the CPU speed accordingly. Related work in this category include [35, 8, 25, 31, 19]. In contrast to interval-based approach, a task-based algorithm makes the performance-setting decisions on a per-task basis. It may also evaluate dynamically the decisions made in the past to adjust the current decision. Algorithms in this category include [37, 15, 9, 24, 20, 30, 33].

Some of the task-based algorithms formulated the DVS scheduling as a (mixed-)integer linear/nonlinear programming problem. For example, Ishihara and Yasuura in [15] gave an ILP formulation for a set of tasks and a set of discrete voltage levels. A task may be assigned different voltage/frequency in different cycles. On the contrary, the paper by Manzak and Chakrabarti [20] assumed continuous voltages and a single voltage/frequency for a task. Raje and Sarrafzadeh [27] formulated the problem, from the context of circuit design, for an acyclic task graph and discrete voltage levels. None of the above took into account the transition costs. Recently, Swaminathan and Chakrabarty in [33] incorporated transition costs into the problem formulation. They assumed a single dual-speed CPU executes a set of periodic non-preemptive real-time tasks.

There is also a growing interest in adapting hardware resources to the current workload requirement so as to save energy. Dynamic voltage scaling can be considered as part of this strategy. Examples in this category include [7, 21, 6, 13]. For example, Ghiasi et al. in [7] and Childers et al. in [6] suggested using desired IPC rate to direct the DVS application. Marculescu in [21] proposed to use cache misses as the scaling points.

Compiler is also involved in recognizing, and possibly eliminating, CPU slacks. Some of the algorithms exploit the slacks between real program behavior and the estimated one at the compile time (e.g. worst case scenario). These "checkpoints", piece of codes inserted into the original program, encode the voltage scheduling decisions and are executed with the run-time information. Work in this categories include [18, 22, 29, 2]. One issue is where to put these checkpoints. Lee and Sakurai in [18] evenly distributed these points in a task, while Shin et al. in [29] put them on the control flow edges. Recently, the transition costs are taken into account when inserting these points [1].

Our work is similar to [28] in spirit. In [28], Seth et al. described algorithms that can make efficient use of powerdown instructions to shut off processor units not required during portions of program execution. The algorithms are based on program static analysis and a combinatorial optimization formulation of the problem. The problem is to insert the powerdown instructions at different places such that energy saving is maximized and the execution time of the resulting program is not increased beyond a user-specified value.

Finally, we want to point out that the impact of classical compiler optimizations to the program energy usage has been evaluated in a few studies [16, 34, 36]. In general, energy reduction is correlated positively with performance improvement. However, in a recent study [10], it is showed that in some cases more aggressive optimizations introduce significant CPU workload with marginal performance improvement. As a result, it not only may increase the entire system energy usage, but also may prohibit DVS techniques to be applied effectively.

7. CONCLUSION AND FUTURE WORK

Frequency and voltage scaling is an effective technique to reduce CPU power dissipation. We have discussed a novel compiler approach to identify regions in the program that can be slowed down without significant performance penalties. Our algorithm picks a single region to be executed at a lower CPU frequency and voltage. A prototype implementation based on the SUIF2 compiler infrastructure was used to validate the approach on the `SPECfp95` benchmark. Cycle accurate simulation using the SimpleScalar tool set showed significant energy savings of up to 24% with performance penalties of less than 2.7%. To the best of our knowledge, our compiler is the first power-aware compiler to perform dynamic voltage and frequency scaling. An extension of our compilation strategy to multiple regions is currently underway.

Acknowledgements

The authors wish to thank Professor Doug Burger from the University of Texas at Austin for providing the SimpleScalar tool set with his memory extensions.

8. REFERENCES

[1] N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Porceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, 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 *Proceeding of Design, Automation and Test in Europe Conference (DATE)*, March 2002.

[3] 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.

[4] D. Burger. *Hardware Techniques to Improve the Performance of the Processor/Memory Interface.* PhD thesis, Computer Sciences 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, June 1997.

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

[7] 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.

[8] 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.

[9] 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.

[10] C.-H. Hsu and U. Kremer. Dynamic voltage and frequency scaling for scientific applications. In *Proceedings of the 14th annual workshop on Languages and Compilers for Parallel Computing (LCPC 2001)*, August 2001.

[11] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems (PACS)*, November 2000.

[12] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'01)*, August 2001.

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

[14] M. Irwin. Low power design: From soup to nuts, June 2000.

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

[16] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference (DAC)*, June 2000.

[17] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA-81)*, pages 81–87, May 1981.

[18] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation (DAC'00)*, pages 806–809, June 2000.

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

[20] 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.

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

[22] 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.

[23] T. Mudge. Power: A first calss design constraint for future architectures. In *Proceedings of International Conference on High Performance Computing*, December 2000.

[24] 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.

[25] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of 2000 International Symposium on Low Power Electronics and Design (ISLPED'00)*, pages 96–101, July 2000.

[26] 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.

[27] S. Raje and M. Sarrafzadeh. Variable voltage scheduling. In *International Symposium on Low Power Electronics and Design (ISLPED-95)*, pages 9–14, August 1995.

[28] A. Seth, R. Keskar, and R. Venugopal. Algorithms for energy optimization using processor instructions. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, November 2001.

[29] 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.

[30] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'00)*, pages 365–368, November 2000.

[31] A. Sinha and A. Chandrakasan. Dynamic voltage scheduling using adpative filtering of workload traces. In *Proceedings of the 14th International Conference on VLSI Design*, January 2001.

[32] T. SUIF. Stanford university intermediate format.

[33] 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.

[34] M. Valluri and L. John. Is compiling for performance == compiling for power? In *The 5th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-5)*, January 2001.

[35] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *the 1st Symposium on Operating Systems Design and Implementation (OSDI-94)*, pages 13–23, November 1994.

[36] H. Yang, G. Gao, A. Marquez, G. Cai, and Z. Hu. Power and energy impact of loop transformations. In *Porceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, September 2001.

[37] 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.