Compilers for Leakage Power Reduction

YI-PING YOU, CHINGREN LEE, and JENQ KUEN LEE
National Tsing Hua University

Power leakage constitutes an increasing fraction of the total power consumption in modern semiconductor technologies. Recent research efforts indicate that architectures, compilers, and software can be optimized so as to reduce the switching power (also known as dynamic power) in microprocessors. This has lead to interest in using architecture and compiler optimization to reduce leakage power (also known as static power) in microprocessors. In this article, we investigate compiler-analysis techniques that are related to reducing leakage power. The architecture model in our design is a system with an instruction set to support the control of power gating at the component level. Our compiler provides an analysis framework for utilizing instructions to reduce the leakage power. We present a framework for analyzing data flow for estimating the component activities at fixed points of programs whilst considering pipeline architectures. We also provide equations that can be used by the compiler to determine whether employing power-gating instructions in given program blocks will reduce the total energy requirements. As the duration of power gating on components when executing given program routines is related to the number and complexity of program branches, we propose a set of scheduling policies and evaluate their effectiveness. We performed experiments by incorporating our compiler analysis and scheduling policies into SUIF compiler tools and by simulating the energy consumptions on Wattch toolkits. The experimental results demonstrate that our mechanisms are effective in reducing leakage power in microprocessors.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Algorithms, Experimentation Languages

Additional Key Words and Phrases: Compilers for low power, leakage-power reduction, power-gating mechanisms

1. INTRODUCTION

The demands of power-constrained mobile and embedded computing applications are increasing rapidly, which makes the reduction of power consumption

This work was supported in part by the National Science Council (under grant nos. NSC-93-2213-E-007-025, NSC-93-2220-E-077-020, NSC-93-2220-E-007-019, and NSC-93-2752-E-007-004-PAE), Ministry of Economic Affairs (under grant nos. 93-ED-17-A-03-S1-0002 and 94-EC-17-A-01-S1-034), and ITRI (under an ITRI/NTHU research grant).

Authors' address: Department of Computer Science, National Tsing Hua University, 101, Section 2 Kuang Fu Road, Hsinchu 30013, Taiwan; email: {ypyou,crlee}@pllab.cs.nthu.edu.tw; jklee@cs.nthu.edu.tw.

a crucial challenge for software and hardware developers. The continuing size reductions and increasing speeds of transistors increases the importance of leakage-power dissipation in the absence of any switching activities. Recent theoretical analyses have attempted to characterize engineering equations and cost models for analyzing static powers [Thompson et al. 1998; De and Borkar 1999; Doyle et al. 2002]. One such analysis produced the following relationship: $P_{static} = V_{CC} \cdot N \cdot k_{design} \cdot \hat{I}_{leak}$, where V_{CC} is the supply voltage, N is the number of transistors in the design, k_{design} is the characteristic of an average device, and $\hat{I}_{leak}$ is a technology parameter describing the per-device subthreshold leakage [Butts and Sohi 2000].

In this article, we discuss compiler analysis techniques used to reduce the number of devices, N, in the static power equation above to ease the problem of leakage power. The architecture model in our design is a system with an instruction set that supports the control of power gating at the component level. We attempt to reduce the number of devices by turning devices off when they not being used. Our work provides compiler solutions for the analysis and scheduling of the power-gating control at the component level. A data-flow analysis framework is given that estimates the component activities at fixed points in programs whilst considering pipeline architectures. We also provide equations that can be used by the compiler to determine whether employing power-gating instructions in given program blocks will reduce the total energy requirements. As the duration of power gating on components in given program routines is related to the number and complexity of program branches, we propose a set of scheduling policies (*Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched*) and evaluate their effectiveness. Our proposed framework are effective for machines with in-order executions. Additional cares have to be taken when one deals with out-of-order issues. For out-of-order issues, we suggest power-gating operations on a function unit should be considered dependent to normal operations on this unit. Our experiments are performed by incorporating our compiler analysis and scheduling policy into SUIF compiler tools [Smith 1998; Stanford Compiler Group 1995] and by simulating the energy consumptions on Wattch [Brooks et al. 2000] toolkits. We also revise Wattch/SimpleScalar to adopt our proposed schemes to deal with out-of-order issues. The experimental results demonstrate that our mechanisms are very effective in reducing leakage power in microprocessors. In summary, the key contributions of our work include the presentations of data flow analysis framework for component activities, the scheduling policies for power-gating instructions going beyond basic blocks, and the suggestions of hardware refinements for out-of-order issues to work with our proposed methods.

The remainder of this article is organized as follows: Section 2 presents our machine architectures with power-gating controls. Section 3 presents our data-flow analysis framework for component activities. Next, Section 4 provides scheduling policies for leakage power reductions by utilizing gathered component information. Experimental results will then be presented in Section 5. Finally, Section 6 describes related work and Section 7 concludes this article.

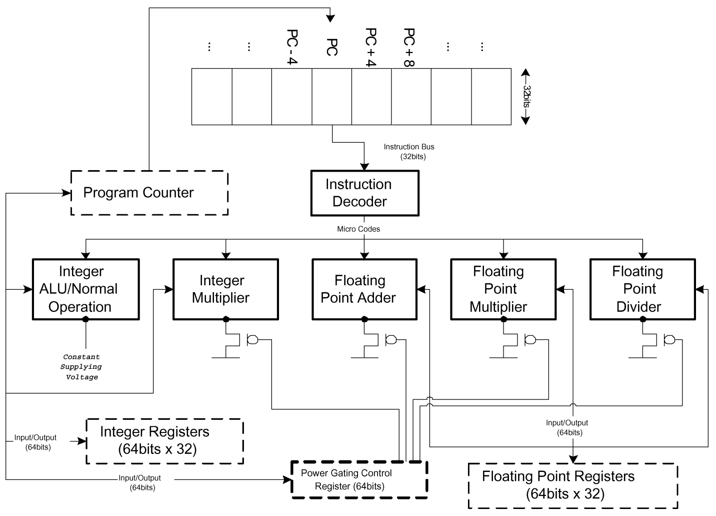

PC - 4
PC
PC + 4
PC + 8
32bits
Instruction Bus (32bits)
Program Counter
Instruction Decoder
Micro Codes
Integer ALU/Normal Operation
Integer Multiplier
Floating Point Adder
Floating Point Multiplier
Floating Point Divider
Constant Supplying Voltage
Input/Output (64bits)
Integer Registers (64bits x 32)
Power Gating Control Register (64bits)
Floating Point Registers (64bits x 32)

Fig. 1. Machine architecture model with power-gating control.

2. MACHINE ARCHITECTURE

The architecture model in our design is a system with an instruction set that supports the control of power gating at the component level. Figure 1 shows an example of our target machine architecture on which our optimization is based. We focus on the reduction of the power consumption of the certain function units by invoking the "power-gating" technology. Power gating is analogous to clock gating—power gating turns off devices by switching off their supply voltage rather than switching off the clock. This can be achieved by forcing transistors to turn off or using multithreshold voltage CMOS technology (MTCMOS) to increase the threshold voltage [Butts and Sohi 2000; Kao and Chandrakasan 2000; Roy 1998].

We built the experimental architecture within the Wattch simulation environment [Brooks et al. 2000]. In this simulation environment we can measure the power consumption of every microprocessor component throughout the experimental program. This architecture is essentially compatible with the DEC Alpha 21264 processor [Compaq Computer Corporation 1999]; the major difference between these two architectures is the additional power-gating design and the static pipeline scheduling in our experimental architecture. The compiler approach proposed in this article is basically for in-order issue processors, but we also propose a solution to make our methodology feasible for out-of-order issue processors shown later in Section 5.3. We implemented the proposed mechanism into SimpleScalar and evaluated our approach with out-of-order issue processors.

The power-gated function units in our experimental architecture are Integer Multiplier, Floating-Point Adder, Floating-Point Multiplier, and Floating-Point Divider. The power gating of each function unit can be controlled by the "power-gating control register" (PGCR). The PGCR is a 64-bit integer register.

In this case, only the lowest four bits of this register can affect the power-gating status. The 0th bit of the lowest four bits of the PGCR controls the power gating of the Integer Multiplier: setting this bit will cause the Integer Multiplier on, and clearing it will turn off the corresponding function unit in the next clock cycle. The 1st, 2nd, 3rd bits of these four bits are used for the Floating-Point Adder, Floating-Point Multiplier, and Floating-Point Divider, respectively. It is worth mentioning that the integer ALU unit within the architecture is also involved in general program execution, since it also performs data movements to the PGCR. This means that the integer ALU is always required, and so this function unit is always on. In addition, we invoke a new instruction in the simulation environment to specify the access direction of PGCR. This instruction can operate those four power-gated function units at once by moving the appropriate value from a general-purpose register to the PGCR.

3. COMPONENT-ACTIVITY DATA-FLOW ANALYSIS

In this section, we investigate the compiler analysis techniques used to reduce the leakage power. We present a data-flow analysis framework for a compiler to analyze the state of components in a microprocessor. The process collects the information of the utilization of components at various points in a program. We first construct basic blocks and control flow graphs of given programs, and then develop a data-flow equation for the summary of component usages at given program points. To gather the data-flow information, we define *comp_gen*[*B*], *comp_kill*[*B*], *comp_in*[*B*], and *comp_out*[*B*] for each block *B*.

We say that a component-activity *c* is generated at a block *B* if a component is required for this execution, symbolized as *comp_gen*[*B*], and that it is killed if the component is released by the last request, symbolized as *comp_kill*[*B*]. We then create the two groups of equations shown below. The first group of equations follows from the observation that *comp_in*[*B*] is the union of activities arriving from all the predecessors of *B*. The second group is the activities at the end of a block that are either generated within the block, or those entering at the beginning but not killed as control flows through the block. The data-flow equation for these two groups is as follows:

$$comp\_in[B] = \bigcup_{\substack{P \text{ a predessor} \\ \text{of } B}} comp\_out[P]$$

$$comp\_out[B] = comp\_gen[B] \cup (comp\_in[B] - comp\_kill[B]).$$

We use an iterative approach to compute the desired results of *comp_in* and *comp_out* after *comp_gen* has been computed for each block. The algorithm is sketched in Figure 2. This is an iterative algorithm for data-flow equations [Aho et al. 1986] with the addition of resource management structures. A two-dimension array, called *RemainingCycle*, is used to maintain the number of cycles that are required to fulfill requests for each component and block. In addition, a resource-utilization table is adopted to give the resource requirement for each instruction of the given microprocessor. The resource-utilization table can be used to give the initial values of *RemainingCycle*. The remaining cycles

```
Input   A control flow graph in which each block $B$ contains only one instruction;
        a resource utilization table.
Output  $comp\_in[B]$ and $comp\_out[B]$ for each block $B$.

Begin
    for each block $B$ do begin
        /* computation of $comp\_gen$ */
        for each component $C$ that will be used by $B$ do begin
            $RemainingCycle[B][C] := N$,
                where $N$ is the number of cycles needed for $C$ by $B$;
            $comp\_gen[B] := comp\_gen[B] \cup C$;
        end

        $comp\_in[B] := comp\_kill[B] := \emptyset$;
        $comp\_out[B] := comp\_gen[B]$;
    end

    /* iterative analysis */
    while changes to any $comp\_out$ occur do begin
        for each block $B$ do begin
            /* computation of $comp\_kill$ */
            for each component $C$ do begin
                $RemainingCycle[B][C] := \mathbf{MAX}(RemainingCycle[P][C]) - 1$,
                    where $P$ is a predecessor of $B$;
                if $RemainingCycle[B][C] = 0$ then $comp\_kill[B] := comp\_kill[B] \cup C$;
            end

            /* computation of $comp\_in$ */
            $comp\_in[B] := \bigcup comp\_out[P]$, where $P$ is a predecessor of $B$;

            /* computation of $comp\_out$ */
            $comp\_out[B] := comp\_gen[B] \cup (comp\_in[B] - comp\_kill[B])$;
        end
    end
End
```

Fig. 2. Data-flow analysis algorithm for component activities.

of a component decrease by one for each propagation. Initially, both *comp_in* and *com_kill* are set to be empty. The iteration continues until *comp_in* (and hence *comp_out*) converges. As *comp_out*$[B]$ never decreases in size for any B, the algorithm will eventually halt when all *comp_out* are in the steady state. Intuitively, the algorithm propagates activities of components as far as they will go by simulating all possible execution paths of the program. This algorithm provides the state of utilization of components for each point of a program.

4. LEAKAGE-POWER REDUCTION

In this section, we present a cost model for the compiler to determine whether power-gating control should be applied, and a set of scheduling policies to place power-gating instructions within given programs.

4.1 Cost Model

With the utilization of components obtained from Section 3, we can insert power-gating instructions into programs at the appropriate points (i.e., the beginning and of an inactive block) to turn off and on unused components so as to reduce the leakage power. However, both shut-down and wake-up procedures are associated with an additional penalty, especially the latter due to peak voltage requirements. The following equation represents our cost model for deciding if the insertion of power-gating instructions will provide energy-consumptions benefits:

$$E_{turn\_off}(C) + E_{turn\_on}(C) \leq \mathbf{BreakEven_C} \times P_{leak\_saving}(C),$$

where $E_{turn\_off}(C)$ is the energy penalty for shutting down component C, E_{turn_on} is the energy penalty for waking up component C, $\mathbf{BreakEven_C}$ is the break-even cycle for component C, and $P_{leak\_saving}(C)$ is the leakage-power saving of component C per cycle when power-gating controls are employed. The left-hand side of the equation shows the energy consumed by shut-down and wake-up procedures, and the right-hand side equals the leakage energy consumed over a certain number of cycles. Power-gating control will only save power if the amount of power required to shut down and wake up is less than the leakage energy consumed during the same intervening period in the absence of these procedures.

The latency associated with turning a component on should also be considered when employing power gating. Due to the high capacitance of microprocessor circuits, a component will typically need several clock cycles to reach its normal operating state. Butts and Sohi [2000] also illustrated that at 1 GHz it takes about 7.5 cycles to charge 5 nF to 1.5 V with 1 A (which are typical values in microprocessor circuits). With this consideration, we enforce power gating on a component only when the size of its inactive block (i.e., the idle region) is larger than its break-even cycle and its latency to recovery. Our cost model after incorporating latency becomes the following:

$$\mathbf{Threshold_C} = \mathbf{MAX}(\mathbf{BreakEven_C}, \mathbf{Latency_C}),$$

where $\mathbf{Latency_C}$ is the power-gating latency of component C. In addition, we attempt to insert the wake-up operations of power-gating control ahead of the time at which the corresponding components are required, in order to avoid program stalling whilst waiting for the wake-up latency.

4.2 Scheduling Policies for Power Gating

The component activity information gathered and the cost model for deciding if the power-gating instructions should be employed now allow us to consider the scheduling mechanisms when inserting the power-gating instructions into given programs. As the duration of power-gating control on components is influenced to conditional branches in programs, we propose a set of scheduling policies (*Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched*) with power-gating instructions. The details are given below.

A naive mechanism to control the power-gating instructions will set the on and off instructions at each basic block according to the component activities gathered by the data-flow equation in Section 3. We call this scheme *Basic_Blk_Sched*.

An additional complication is that the inactive period of a component may span more than two adjacent basic blocks. We therefore use a *depth-first-traveling* algorithm to traverse all possible execution paths. In general, an inactive block will be turned off when the criteria discussed in Section 4.1 are reached. Another case to consider in power gating is that of an inactive block containing conditional branches, since the length of the two inactive blocks—which follow the branch targets—may be different. For example, only one of the branchings may benefit from power gating, in which case taking power-gating control in that branch when the other branch is instead taken may not reduce the power requirements. In other words, the path lengths of the taken and not-taken paths of a branch may not be equal and therefore one path may satisfy the cost model in Section 4.1 and the other path may not. Hence, we propose a *MIN_Path_Sched* policy to ensure that power-gating control is activated only when the inactive lengths of both branching paths exceed the power-gating threshold; that is, the minimum length of those paths reaches the criteria for power gating.

Figure 3 presents the details for the *MIN_Path_Sched* algorithm proposed. The algorithm is adopted from depth-first-traveling algorithm, where recursion is incorporated in order to guarantee that all paths of the inputted control flow graph (CFG)—which is annotated with component utilization—are traversed. The arguments (C, B, *Branched*, *Edge*, and *Count*) represent the type of the component in analysis for power-gating control, the node ID of the CFG, a Boolean variable that shows whether the current traverse comes through a branch, the type of the outgoing edge, and the accumulated inactive length so far, respectively. The algorithm starts traversing from the root of the CFG with a *Count* of zero, schedules power-gating instructions at the beginning and end of inactive blocks if necessary, and halts when all execution paths are traversed. The algorithm is divided into four parts to handle conditions when encountering or not encountering a conditional branch while the analyzing component is active or inactive, respectively:

(1) *A conditional branch is reached and the component is inactive.* Under this condition, the algorithm increases the *Count* and makes two recursive calls that returns the inactive length of its right and left branch, respectively. A judgment on power gating is then made, and it returns the minimum inactive length of two branchings. Note that *comp_out*[B] represents the set of the component activities of block B; therefore, the condition $C \notin$ *comp_out*[B] indicates that component C is not in the active set of *comp_out*[B].
(2) *A conditional branch is reached and the component is active.* Under this condition, the algorithm takes control of power gating if necessary, starts two recursive calls for both branches, and finally returns the current inactive length.
(3) *Any statement except for a conditional branch is reached and the component is inactive.* Under this condition, the algorithm continues the traverse; that is, it only increases *Count* and then returns.
(4) *Any statement except for a conditional branch is reached and the component is active.* Like condition 2, the algorithm takes control of power gating if necessary and starts a new traveling for its successor. And finally, it returns *Count*.

```
Input   A control flow graph (CFG) annotated with component utilization.
Output  A scheduling for power-gating instructions.

MIN_Path_Sched($C, B, Branched, Edge, Count$)
Begin
    if block $B$ is the end of $CFG$ or $Count >$ MAX_COUNT then return $Count$;
    if block $B$ has two children then do

        /* condition 1; conditional branch, inactive */
        if $C \notin comp\_out[B]$ then do
            $Count := Count + 1$;
            if left edge is a forward edge then
                $l\_Count := MIN\_Path\_Sched(C$, left child of $B$, TRUE, FWD, $Count)$;
            else
                $l\_Count := MIN\_Path\_Sched(C$, left child of $B$, TRUE, BWD, $Count)$;
            if right edge is a forward edge then
                $r\_Count := MIN\_Path\_Sched(C$, right child of $B$, TRUE, FWD, $Count)$;
            else
                $r\_Count := MIN\_Path\_Sched(C$, right child of $B$, TRUE, BWD, $Count)$;
            if MIN($l\_Count, r\_Count$) > $\mathbf{Threshold_C}$ and !$Branched$ then
                schedule power-gating instructions at the beginning and end of inactive blocks;
            return MIN($l\_Count$, $r\_Count$);

        /* condition 2; conditional branch, active */
        else
            if $Count >$ $\mathbf{Threshold_C}$ and !$Branched$ then
                schedule power-gating instructions at the beginning and end of inactive blocks;
            if $Edge =$ FWD then
                if right edge is a forward edge then
                    $MIN\_Path\_Sched(C$, left child of $B$, FALSE, FWD, $Count)$;
                else
                    $MIN\_Path\_Sched(C$, left child of $B$, FALSE, BWD, $Count)$;
                if left edge is a forward edge then
                    $MIN\_Path\_Sched(C$, right child of $B$, FALSE, FWD, $Count)$;
                else
                    $MIN\_Path\_Sched(C$, right child of $B$, FALSE, BWD, $Count)$;
            end
            return $Count$;
        end;
    else

        /* condition 3; statements except conditional branches, inactive */
        if $C \notin comp\_out[B]$ then do
            $Count := Count + 1$;
            if edge is a forward edge then
                return $MIN\_Path\_Sched(C$, child of $B$, $Branched$, FWD, $Count)$;
            else
                return $MIN\_Path\_Sched(C$, child of $B$, $Branched$, BWD, $Count)$;

        /* condition 4; statements except conditional branches, active */
        else
            if $Count >$ $\mathbf{Threshold_C}$ and !$Branched$ then
                schedule power-gating instructions at beginning and end of inactive blocks;
            if $Edge =$ FWD then
                if the edge pointing to child of $B$ is a forward edge then
                    $MIN\_Path\_Sched(C$, child of $B$, FALSE, FWD, $Count)$;
                else
                    $MIN\_Path\_Sched(C$, child of $B$, FALSE, BWD, $Count)$;
            end
            return $Count$;
        end
    end
End
```

Fig. 3. *MIN_Path_Sched* algorithm based on depth-first-traveling for power gating.

Note that care must be taken for recursive boundaries to reach the backward edges for a loop. As a depth-first search algorithm can find the loop, cycling can occur in our algorithm. In a cyclic situation, if none of the instructions used in the cycle of a program fragment use the component in the search, we will assume the loop cycle is executed once with the minimum-path scheduling policy. If some instructions in the backward edge of a program fragment do use the component in the search, the backward edge extending to that instruction will be accounted for in the program path. In addition, since our proposed algorithm is based on depth-first-traveling, the complexity of our approach is $O(N)$ where N is the number of nodes in a control flow graph.

Next, since the behavior of program branches depends on the structure and the input data of programs, some branches may be followed rarely or even never. To accommodate this, we propose an eclectic policy, called *AVG_Path_Sched*, to schedule power-gating instructions. The only difference between *AVG_Path_Sched* and *MIN_Path_Sched* is the judgments made in condition 1 above: *AVG_Path_Sched* returns the average length of two branchings instead of the minimum. This scheme will take advantage of power reduction if an infrequently taken branch returns a small value of *Count* which causes inactivation of power-gating mechanism. The *AVG_Path_Sched* mechanism can be approximately implemented by assuming the probabilities of all branches are 50%, by assigning branch probabilities at compilation time by programmers or compilers, or by incorporating path-profiling schemes to examine the probabilities of all branches.

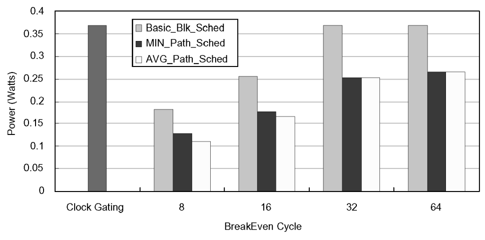
5. EXPERIMENTS AND DISCUSSIONS

5.1 Platform

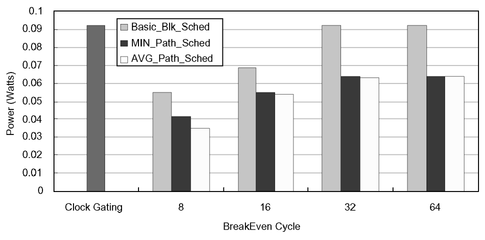
We use a DEC-Alpha-compatible architecture with power-gating control and instruction sets described in Figure 1 as the target architecture for our experiments. The proposed data-flow analysis and scheduling policies are incorporated into the compiler tool with SUIF [Stanford Compiler Group 1995] and MachSUIF [Smith 1998], and evaluated by the Wattch simulator [Brooks et al. 2000]. Table I summaries the baseline configuration of the simulator in our experiment. By default the simulator performs out-of-order execution. We use "-issue:inorder" option in the configuration so that instructions would be executed in order for ensuring the correctness of execution; our approach might be harmed in an out-of-order architecture if no additional support is provided. We discuss the problem and propose solutions with hardware supports to the limitation in Section 5.3. Furthermore, several assumptions are made for completeness as follows: (1) As Wattch does not model leakage at the component level per se, we assume that leakage power contributes 10% of total power consumption. Though ten percent might be underestimated according to De and Borkar [1999] and Thompson et al. [1998], larger percentage of leakage power result in more power reduction for our approach. (2) We assume that wake-up operations of power-gating control take 20-cycle latency, although 7.5 cycles are introduced in Butts and Sohi [2000]. We show our scheme is still

Table I. Baseline Processor Configuration

Parameter	Configuration
Clock	600 MHz
Process parameters	0.10 μm, 1.9 V
Issue	In-order
Decode width	4
Issue width	4
Commit width	4
RUU size	8
LSQ size	8
Function units	4 integer ALU 1 integer multiply/divide unit 4 FP ALU 1 FP multiply/divide unit
Register file	32 64-bit integer registers 32 64-bit FP registers 1 power-gating control register



(a)



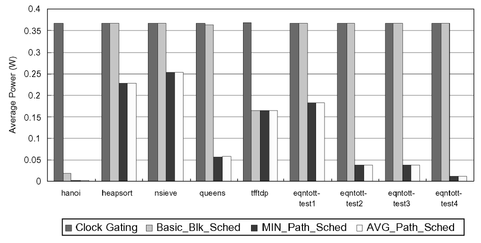
(b)

Fig. 4. (a) Results of Floating-Point Adder for *nsieve* (b) Results of Floating-Point Multiplier for *nsieve*.

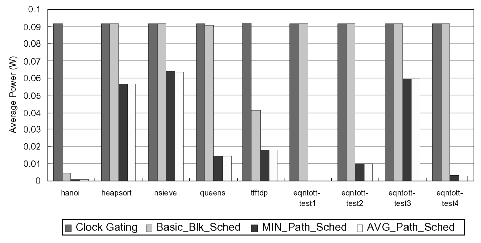
with good benefits despite of overestimated latency. (3) To let the power-gating instructions—which are generated by the optimized compiler—be recognized by the Alpha assembler and linker, power-gating instructions are replaced by a set of instructions "*stl \$24, negative_offset(\$31)*", where *negative_offset* is a negative integer and is used for indicating which function unit to be powered on or off. The instruction stores the value of register \$24 into the memory address below zero, which is an invalid memory address—\$31 is a constant zero register—and should never be generated by standard compilers. To avoid processors from accessing the invalid memory addresses, we made a small modification in SimpleScalar: when the instruction decoder decodes such instructions, it extracts the power-gating information and converts it to a NOP instruction. The test suites used in our experiment are benchmarks listed in AbuFAQ of comp.benchmarks [Aburto et al. 1997].

5.2 Simulation Results

Figure 4(a) and 4(b) illustrate the power-consumption results for the simulations of power-gating control over Floating-Point Adder and Floating-Point Multiplier for the *nsieve* application, respectively. In these figures, the X-axis represents the break-even cycle for our scheduling criteria, and the Y-axis



(a)

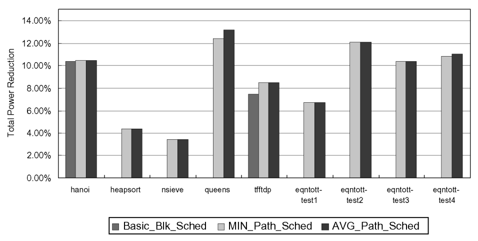


(b)

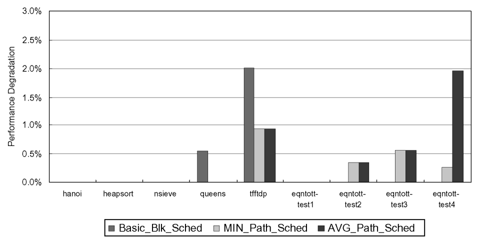
Fig. 5. Power gating on (a) Floating-Point Adder and (b) Floating-Point Multiplier for miscellaneous benchmarks (**BreakEven$_C$** = 32).

represents the power consumption. The leftmost bar shows the power dissipated by function units when no power-gating control is employed, which represents the result of standard clock-gating mechanism provided by the Wattch power estimator. We use this as the baseline version for comparison. The clock-gating mechanism gates the clocks of those unused resources in multiported hardware to reduce the dynamic power. However, static power is still leaked. The remaining bars in the figures show the power-gating results for the proposed scheduling policies with different break-even cycles. The results show that the power-gating mechanism reduces the leakage power by a large amount even when the penalty of power-gating control is high (i.e., a large break-even cycle). Note that we have incorporated the energy penalty associated with inserting power-gating instructions into the Wattch power simulator. Our experimental data also indicate that the *MIN_Path_Sched* and *AVG_Path_Sched* scheduling algorithms always provide better results than the *Basic_Blk_Sched* algorithm. This is because the *Basic_Blk_Sched* algorithm schedules power-gating instructions within basic blocks while the other two schedule those beyond branches. The possible inactive durations of components are extended when *MIN_Path_Sched* or *AVG_Path_Sched* is employed. A more accurate model for the *AVG_Path_Sched* mechanism would incorporate path profiling schemes (replacing our assumption of 50% probabilities in all branches), which would further improve the results. The power consumed by the Floating-Point Adder is reduced from 0% to 50.3%, from 27.5% to 65.5%, and from 27.5% to 70.2% for the *Basic_Blk_Sched*, *MIN_Path_Sched* and *AVG_Path_Sched* policies, respectively. The corresponding reductions for the Floating-Point Multiplier are from 0% to 39.8%, 30.3% to 55.4%, and 30.3% to 62.0%, respectively.

Figure 5(a) and Figure 5(b) give the power consumption of the Floating-Point Adder and the Floating-Point Multiplier for various benchmarks while employing the power-gating mechanism with a break-even cycle of 32. It is again evident that the *AVG_Path_Sched* policy provides the greatest power reduction, with the *MIN_Path_Sched* and *Basic_Blk_Sched* coming second and third, respectively. However, all three produce better results than the one without power gating (i.e., which only employs clock gating). Figure 5(a) shows that the *Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched* policies produce average reductions of 16.77%, 70.41%, 70.43% for the Floating-Point Adder in



(a) Total energy reduction



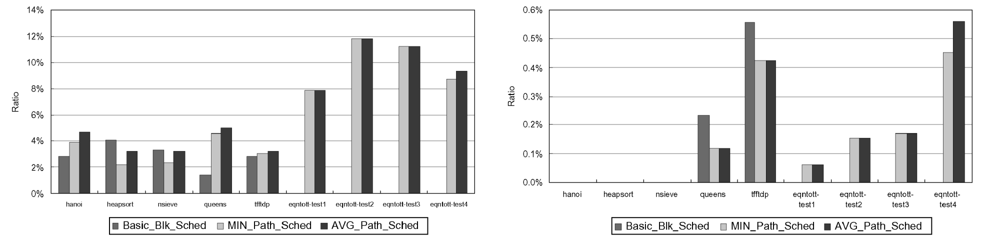
(b) Performance degradation

Fig. 6. Simulation results when using the power-gating control for integer and floating-point units ($\textbf{BreakEven}_{\textbf{C}} = 32$).

all benchmarks, respectively. In the case of the *hanoi* benchmark, which is an integer program, the reduction is 96.7% of power for *Basic_Blk_Sched* and 99.5% for *MIN_Path_Sched* and *AVG_Path_Sched*. Similar results are summarized in Figure 5(b).

Our experiments demonstrate a clear reduction in the energy consumption of each component, but it is also of interests to determine the reduction in energy as a percentage of the total energy consumption. The information from our experimental data is given in Figure 6(a). For a break-even cycle set to be 32, using the power-gating control for integer and floating-point units with the *Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched* policies reduces the total power by average 1.98%, 8.78%, and 8.89%, respectively. With regard to the impact on performance, the cycle counts of execution provided by the Wattch (i.e., SimpleScalar) show that our approach has a light impact (less than 2%) on performance. Figure 6(b) shows the performance degradation in terms of different scheduling policies. Note that, as mentioned earlier, the latency of power-on operations is assumed to be 20 cycles, which is overestimated for strict evaluation. Note that the performance degradation numbers of hanoi, heapsort, nsieve, and eqntott-test1 program are too small (less than 0.01%) to be illustrated in the figure. The reason why the numbers are small is that the execution time of these programs is so large that it amortizes the performance impact caused by power-gating operations.

We also compiled the experiment statistics and found that the ratios of power-gating instructions to total instructions in the program code and simulated code are small. Figure 7 illustrates the details. Figure 7(a) shows the ratios of power-gating instructions in the program code when Integer ALU, Floating-Point Adder, and Floating-Point Multiplier are considered for power gating, and Figure 7(b) shows those in the simulated code (code ratio in runtime execution) when Integer ALU, Floating-Point Adder, and Floating-Point Multiplier are considered for power gating. It is found that the ratios in tfftdp and eqntott-test4 program are much higher. This is because the lengths of these programs are small so that the proportion of power-gating instructions looks much larger. However, it is also found that power-gating instructions, in fact, are issued with a very small number of counts, which turns out that the inserted power-gating instructions would barely affect the program execution.



(a) Program code (static version) (b) Simulated code (dynamic version)

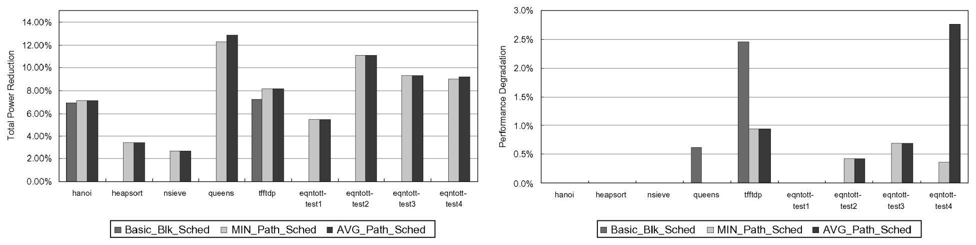
Fig. 7. Ratio of power-gating instructions to total instructions when using the power-gating control for integer and floating-point units ($\mathbf{BreakEven_C} = 32$).

Finally, in the issue to increase the off-times of the units by scheduling instructions, it's certainly important to be able to do that. As compiler technologies nowadays are now done one phase after another phase. We consider the increase of the off-time phase can be done as a separate phase before our instruction issue phase. Our work can work with the increase off-time phase if available.

5.3 Simulation Results for Out-of-Order Issue Processors

The proposed framework is applicable to general superscalar machines that execute instructions in order. Our approach can be applied to out-of-order issue machines as well if additional hardware supports are employed. We present a solution below. Superscalar machines use dynamic pipeline scheduling, which dynamically reorder instructions to avoid hazards (such as structure and data hazards), to utilize the resources and then result in out-of-order execution. To ensure that power-gating instructions are executed at the correct timing with respect to instructions, called consumer instructions, that use the power-gated function units, power-gating instructions on a function unit are considered dependent to consumer instructions on this unit. That is, consumer instructions cannot be advanced before power-on operations and postponed after power-off operations during the dynamic pipeline scheduling; moreover, power-on and power-off operations are not interchangeable. In this regard, the situation that an instruction finds its function unit turned off can be avoided, which turns out that our approach can be applied to out-of-order machines.

We implemented the above idea into SimpleScalar by checking the status of reservation stations to maintain the dependencies between power-gating instructions and consumer instructions. (Remember that, in this article, we had proposed a compiler technique to insert power-gating instructions at appropriate positions, i.e., the compiler automatically inserts a power-on instruction before consumer instructions and a power-off instruction after the consumer instruction.) These dependencies can be categorized into two types: the dependencies between power-on instructions and consumer instructions that use the unit to be powered on and the dependencies between power-off instructions and consumer instructions that use the unit to be powered off. In fact, the former



(a) Total energy reduction (b) Performance degradation

Fig. 8. Simulation results (out-of-order issue) when using the power-gating control for integer and floating-point units ($\mathbf{BreakEven_C} = 32$).

type of dependencies is unnecessary and needs no adaptation in SimpleScalar since a power-on instruction should be issued in advance of consumer instructions due to the in-order fetch model in SimpleScalar, and thus a unit will be powered on before it is used. However, the dependencies between a power-off instruction and consumer instructions are required—power-off instructions might be issued in advanced of consumer instructions—in order to avoid executing instructions on a unit that is turned off. We modified SimpleScalar to ensure that a power-off instruction is stalled until the reservation station of the unit to be powered off is empty, that is, the power-off instruction is issued after all of the consumer instructions are executed. Moreover, to ensure the execution order of power-on and power-off instructions, we enforce a power-gating instruction be stalled until an another power-gating instruction prior to the power-gating instruction are issued. In fact, if this happens, these two power-gating instructions, a power-on instruction and a power-off instruction, can be flushed since a successive execution of the power-on and power-off operation is unnecessary. It would be meaningless if we power on a unit right after powering off the unit and vice-versa. With the above adaptation, the proposed compiler framework for power-gating control can be applied to out-of-order issue processors.

Admittedly, the above mechanism will have performance impacts—a power-off instruction is stalled and occupied a slot in the instruction window until the dependencies between the power-off instruction and consumer instructions are resolved, but along with the results of Figure 7(b), the occurrence of issuing a power-gating instruction is so rare that the performance impact would be negligible. In fact, the performance impacts can even be eliminated with a minor modification in the implementation: adding a power management controller to handle power-gating instructions. Once the instruction decoder decodes a power-gating instruction, the instruction dispatcher dispatches the instruction to the power management controller and removes the instruction from the instruction window and the power management controller performs power-gating control based on the behavior described in the last paragraph. In this case, the performance impact is eliminated.

We used the same processor configuration in Table I, but the instruction issue is changed to be out-of-order to evaluate the effect on out-of-order processors. Figure 8(a) shows the total energy reduction in terms of the entire microprocessor when only integer and floating-point units are under the power-gating control. The results are quite similar to Figure 6(a) which is configured with in-order issue. The major differences in simulation result between in-order and out-of-order processors are the number of the simulation cycle and the total energy consumption. Out-of-order processors are in average 29.3% faster and consume less 17.5% of power than in-order processors. The performance degradation due to power-gating instructions is shown in Figure 8(b) and is almost the same as those shown in Figure 6(b).

6. RELATED WORK

Minimization of power dissipation can be considered at algorithmic, architectural, logic, and circuit levels [Chandrakasan et al. 1992]. Studies on low-power design are abundant in the literature [Alidina et al. 1994; Benini and Micheli 1995; Hachtel et al. 1994; Hong et al. 1999; Prasad and Roy 1993; Roy and Prasad 1992; Tsui et al. 1993], and these have proposed various techniques for synthesizing designs with low transitional activities. Static power dissipation (or the leakage current in the absence of any switching activities) has increased in importance as transistors have become smaller and faster.

The reduction in power consumption has been addressed by architecture designs and software arrangements at the instruction level [Bellas et al. 2000; Chang and Pedram 1995; Horowitz et al. 1994; Lee et al. 2003, 1997; Su and Despain 1995; Tiwari et al. 1997, 1998]. The efforts to reduce dynamic power include software rearrangements to optimize the value locality of registers [Chang and Pedram 1995], the swapping of operands for the Booth multiplier [Lee et al. 1997], the scheduling of VLIW instructions to reduce the power consumption on the instruction bus [Lee et al. 2003], gating the clock to reduce workloads [Horowitz et al. 1994; Tiwari et al. 1997, 1998], cache subbanking mechanism [Su and Despain 1995], and the utilization of the instruction cache [Bellas et al. 2000].

Several research groups have recently proposed and developed hardware techniques to reduce dynamic and static power dissipation. The work of Powell et al. [2000] combines circuit and architectural techniques to reduce the power consumption in a processor's cache. The cache miss rate is used to determine the working-set size of the application relative to that of the cache. Power is then removed from the unused portions of the cache using V_{dd}-gated transistors. Kaxiras et al. [2001] also addressed static power dissipation in the cache, by considering policies and implementations for reducing cache leakage by turning off cache lines when they hold data that is unlikely to be reused. Our approach considers compiler optimizations for static power reduction, and forms a part of our efforts in the Design Technology Center of our university to develop compiler toolkits [Lee et al. 2003; You et al. 2001, 2002; Chen et al. 2004; Hwang et al. 1998, 2003; Chang et al. 1998, 2001] for reducing the power consumption of advanced microprocessors. The work done by Rele et al. [2002] is a concurrent work to ours by using compiler technique and microarchitecture support to guide power-gating controls. We brought up the idea quite early as well as the essential mechanism of this work was applied for a patent in Taiwan, June 2001 (with the issue number 172459). Rele's work is based on profiling approach to identify hot blocks and cold blocks with the execution frequencies of those blocks. Our work provides data-flow analysis framework for component activities. In addition, we present schemes to schedule power-gating instructions go beyond basic blocks when branches are encountered.

7. CONCLUSIONS

In the study described in this article, we investigated compiler analysis techniques aimed at reducing microprocessor leakage power. The architecture model in our design is a system with an instruction set that supports the control of power gating at the component level. Here we presented a data-flow analysis framework for estimating the component activities at fixed points of programs whilst considering pipeline architectures. A set of scheduling policies comprising *Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched* mechanisms was proposed and evaluated. The experimental results demonstrate that our mechanisms are effective in reducing leakage power in microprocessors. Future research directions include investigating the effects of using *AVG_Path_Sched* mechanism with path profiling and edge profiling schemes in experiments.

REFERENCES

ABURTO, A., SILL, D., AND THOMPSON, D. 1997. *comp.benchmarks FAQ*. Computer Sciences Department, University of Wisconsin, http://www.cs.wisc.edu/~thomas/comp.benchmarks.FAQ.html.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

ALIDINA, M., MONTEIRO, J., DEVADAS, S., GHOSH, A., AND PAPAEFTHYMIOU, M. 1994. Precomputation-based sequential logic optimization for low power. *IEEE Trans. VLSI Syst. 2*, 4 (Dec.), 426–436.

BELLAS, N., HAJJ, I. N., AND POLYCHRONOPOULOS, C. D. 2000. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Trans. VLSI Syst. 8*, 3 (June), 317–326.

BENINI, L. AND MICHELI, G. D. 1995. State assignment for low power dissipation. *IEEE J. Solid State Circ. 30*, 3 (Mar.), 258–268.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture* (Vancouver, B. C. Canada). 83–94.

BUTTS, J. A. AND SOHI, G. S. 2000. A static power model for architects. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture* (Monterey, CA). ACM, New York, 191–201.

CHANDRAKASAN, A., SHENG, S., AND BRODERSEN, R. 1992. Low-power CMOS digital design. *IEEE J. Solid-State Circ. 27*, 4 (Apr.), 473–484.

CHANG, J.-M. AND PEDRAM, M. 1995. Register allocation and binding for low power. In *Proceedings of the Design Automaton Conference* (San Francisco, CA). 29–35.

CHANG, R.-G., CHUANG, T.-R., AND LEE, J.-K. 1998. Efficient support of parallel sparse computation for array intrinsic functions of Fortran 90. In *Proceedings of the ACM International Conference on Supercomputing* (Melbourne, Australia). ACM, New York, 13–17.

CHANG, R.-G., LI, J.-S., CHUANG, T.-R., AND LEE, J. K. 2001. Probabilistic inference schemes for sparsity structures of Fortran 90 array intrinsics. In *Proceedings of the International Conference on Parallel Processing* (Valencia, Spain). 61–68.

CHEN, P.-S., HWANG, Y.-S., JU, R. D.-C., AND LEE, J. K. 2004. Interprocedural probabilistic pointer analysis. *IEEE Trans. Paral. Distrib. Syst. 15*, 10 (Oct.), 893–907.

COMPAQ COMPUTER CORPORATION. 1999. *Alpha 21264 Microprocessor Hardware Reference Manual*.

DE, V. AND BORKAR, S. 1999. Technology and design challenges for low power and high performance. In *Proceedings of the International Symposium on Low Power Electronics and Design* (San Diego, CA). 163–168.

DOYLE, B., ARGHAVANI, R., BARLAGE, D., DATTA, S., DOCZY, M., KAVALIEROS, J., MURTHY, A., AND CHAU, R. 2002. Transistor elements for 30nm physical gate lengths and beyond. *Intel Tech. J. 6*, 2 (May), 42–54.

HACHTEL, G., HERMIDA, M., A. PARDO, M. P., AND SOMENZI, F. 1994. Re-encoding sequential circuits to reduce power dissipation. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, CA). 70–73.

HONG, I., KIROVSKI, D., QU, G., POTKONJAK, M., AND SRIVASTAVA, M. B. 1999. Power optimization of variable voltage core-based systems. *IEEE Trans. Computer-Aided Des. Integ. Circ. Syst. 18*, 12 (Dec.), 1702–1714.

HOROWITZ, M., INDERMAUR, T., AND GONZALEZ, R. 1994. Low-power digital design. In *Proceedings of the IEEE Symposium on Low Power Electronics*. IEEE Computer Society Press, Los Alamitos, CA, 8–11.

HWANG, G.-H., LEE, J. K., AND JU, R. D.-C. 1998. A function-composition approach to synthesize Fortran 90 array operations. *J. Paral. Distr. Comput. 54*, 1 (Oct.), 1–47.

HWANG, Y.-S., CHEN, P.-S., LEE, J.-K., AND JU, R. 2003. Probabilistic points-to analysis. Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing (LCPC). vol. 2624. Springer-Verlag, New York, 290–305.

KAO, J. T. AND CHANDRAKASAN, A. P. 2000. Dual-threshold voltage techniques for low-power digital circuits. *IEEE J. Solid-State Circ. 35*, 7 (July), 1009–1018.

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the International Symposium on Computer Architecture* (Gothenburg, Sweden). 240–251.

LEE, C., LEE, J. K., HWANG, T.-T., AND TSAI, S.-C. 2003. Compiler optimization on VLIW instruction scheduling for low power. *ACM Trans. Des. Automat. Electron. Syst. 8*, 2 (April), 252–268.

LEE, M. T.-C., TIWARI, V., MALIK, S., AND FUJITA, M. 1997. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans. VLSI Syst. 5*, 1 (Mar.), 123–133.

POWELL, M., YANG, S.-H., FALSA, B., ROY, K., AND VIJAYKUMAR, T. 2000. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design* (Rapallo, Italy). ACM, New York, 90–95.

PRASAD, S. C. AND ROY, K. 1993. Circuit activity driven multilevel logic optimization for low power reliable operation. In *Proceedings of the EDAC'93 EURO-ASIC* (Paris, France). 368–372.

RELE, S., PANDE, S., ONDER, S., AND GUPTA, R. 2002. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction*. 261–275.

ROY, K. 1998. Leakage power reduction in low-voltage CMOS designs. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems* (Lisbon, Portugal). IEEE Computer Society Press, Los Alamitos, CA, 167–173.

ROY, K. AND PRASAD, S. C. 1992. SYCLOP: Synthesis of CMOS logic for low power applications. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA). IEEE Computer Society Press, Los Alamitos, CA, 464–467.

SMITH, M. D. 1998. *The SUIF Machine Library*. Division of of Engineering and Applied Science, Harvard University.

STANFORD COMPILER GROUP. 1995. *The SUIF Library*. Stanford Compiler Group, Stanford Univ. Stanford, CA.

SU, C.-L. AND DESPAIN, A. M. 1995. Cache designs for energy efficiency. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences* (Los Angeles, CA). 306–315.

THOMPSON, S., PACKAN, P., AND BOHR, M. 1998. MOS scaling: Transistor challenges for the 21st century. *Intel Tech. J. Q3*.

TIWARI, V., DONNELLY, R., MALIK, S., AND GONZALEZ, R. 1997. Dynamic power management for microprocessors: A case study. In *Proceedings of the International Conference on VLSI Design* (Hyderabad, India). 185–192.

TIWARI, V., SINGH, D., RAJGOPAL, S., MEHTA, G., PATEL, R., AND BAEZ, F. 1998. Reducing power in high-performance microprocessors. In *Proceedings of the Design Automaton Conference* (San Francisco, CA). 732–737.

TSUI, C., PEDRAM, M., AND DESPAIN, A. 1993. Technology decomposition and mapping targeting low power dissipation. In *Proceedings of the Design Automation Conference* (Dallas, TX). 68–73.

YOU, Y.-P., LEE, C.-R., AND LEE, J.-K. 2002. Compiler optimization for low power on power gating. Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing (LCPC), vol. 2481. Springer-Verlag, New York, 45–60.

YOU, Y.-P., LEE, C.-R., LEE, J.-K., AND SHIH, W.-K. 2001. Real-time task scheduling for dynamically variable voltage processors. In *Proceedings of the IEEE Workshop on Power Management for Real-Time and Embedded Systems* (Taipei, Taiwan). IEEE Computer Society Press, Los Alamitos, CA, 5–10.

Received January 2003; revised May 2004 and December 2004; accepted April 2005