Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors *

Yi-Ping You, Chingren Lee, and Jenq Kuen Lee

Department of Computer Science
National Tsing Hua University
Hsinchu 300, Taiwan
{ypyou, crlee}@pllab.cs.nthu.edu.tw and jklee@cs.nthu.edu.tw

Abstract. Power leakage constitutes an increasing fraction of the total power consumption in modern semiconductor technologies. Recent research efforts also indicate architecture, compiler, and software participations can help reduce the switching activities (also known as dynamic power) on microprocessors. This raises interests on the issues to employ architecture and compiler efforts to reduce leakage power (also known as static power) on microprocessors. In this paper, we investigate the compiler analysis techniques 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 in the component levels. Our compiler gives an analysis framework to utilize the instruction to reduce the leakage power. We present a data flow analysis framework to estimate the component activities at fixed points of programs with the consideration of pipelines of architectures. We also give the equation for the compiler to decide if the employment of the power gating instructions on given program blocks will benefit the total energy reductions. As the duration of power gating on components on given program routines is related to program branches, we propose a set of scheduling policy include *Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched* mechanisms and evaluate the effectiveness of those schemes. Our experiment is done by incorporating our compiler analysis and scheduling policy into SUIF compiler tools [31] and by simulating the energy consumptions on Wattch toolkits [6]. Experimental results show our mechanisms are effective in reducing leakage powers on microprocessors.

1 Introduction

The demands of power-constrained mobile and embedded computing applications increase rapidly. Reducing power consumption hence becomes a crucial

* This paper is submitted to *the 15th Workshop on Languages and Compilers for Parallel Computing*. The correspondence author of this work is Jenq Kuen Lee. His postal address is Prof. Jenq Kuen Lee, Department of Computer Science, National Tsing Hua Univ., Hsinchu, Taiwan. His e-mail address is jklee@cs.nthu.edu.tw, FAX number is 886-3-5723694, and phone number is 886-3-5715131 EXT. 3519.

challenge for today's software and hardware developers. While maximization of battery life is an obvious goal, the reduction of heat dissipations is important as well. The reduction of power consumptions is also with the similar objective as the reduction of heat dissipations. Minimization of power dissipation can be considered at algorithmic, architectural, logic and circuit levels [11]. Studies on low power design are abundant in the literature [3, 4, 15, 19, 27, 28, 36] in which various techniques were proposed to synthesize designs with low transitional activities.

Recently, new research directions in reducing power consumptions have begun to address the issues on the aspect of architecture designs and on software arrangements at instruction-level to help reduce power consumptions [5, 12, 20, 24, 25, 32–34]. The architecture and software efforts to reduce energy consumptions in the recent attempt have been primarily on the dynamic component of power dissipation (also known as dynamic power). The energy, E, consumed by a program, is given by $E = P \times T$, where T is the number of execution cycles of the program [25] and P the average power. The average power P is given by $P = \frac{1}{2} \cdot C \cdot Vdd^2 \cdot f \cdot E$, where C is the load capacitance, Vdd the supply voltage, f the clock frequency, and E the transition count. In order to reduce the dynamic power, several research works have been proposed to reduce the dissipations. For example, software re-arrangements to utilize the value locality of registers [12], the swapping of operands for booth multiplier [25], the scheduling of VLIW instructions to reduce the power consumption on the instruction bus [24], gating clock to reduce workloads [20, 33, 34], cache sub-banking mechanism [32], the utilization of instruction cache [5], etc.

As transistors become smaller and faster, another mode of power dissipation has become important. This is static power dissipation or the leakage current in the absence of any switching activities. For example, consider two Intel's Pentium III processors manufactured on 0.18μm process, the Pentium III 1.0 GHz and the Pentium III 1.13 GHz [21]. The datasheet lists the 1.0 GHz processor has a total power dissipation of 33.0 Watts and a deep sleep (i.e., static) power of 3.74 Watts while the maximum power dissipation at 41.4 Watts and the static power at 5.40 Watts for the 1.13 GHz one. The static power is up by 44% and comprises 13% of the total power dissipation while the total power is increased by only 25%. Figure 1 and Figure 2 show the growing ratio of static power among the total power [14, 35]. Figure 1 gives the growing trend of static power. It's growing in a fast paste. Figure 2 again shows the the growing trend of static power in terms of temperatures in the hardware devices. This raises the importance of reducing static power dissipations. Recently, academic results have tried to characterize the engineering equation and cost model for analyzing static powers [14, 35]. This is important, as the architecture designers and system developers can then deploy the architecture and software designs to reduce the static power according to the cost model. Previously, the availability of the cost equation for dynamic powers have prompted fruitful research results in the efforts to reduce dynamic power. The recent result in characterizing static power has the following equation. $P_{static} = V_{CC} \cdot N \cdot k_{design} \cdot \hat{I}_{leak}$, where V_{CC} is the supply

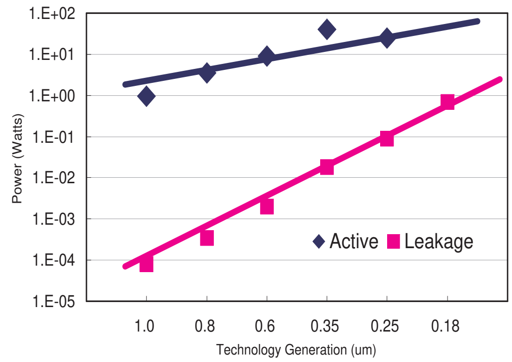

Fig. 1. Trends in Active and Leakage Power Dissipation (From Thomopson et. al.)

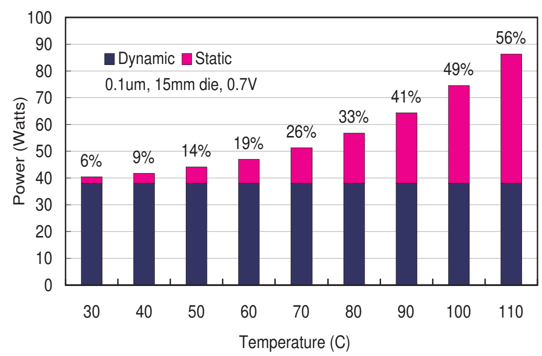

Fig. 2. Leakage Power Trend in Temperature (From De et. al.)

voltage, N is the number of transistors in design, k_{design} is the characteristic of an average device, $\hat{I}_{leak}$ is the technology parameter describing the per device subthreshold leakage [8].

In this paper, we investigate the compiler analysis techniques to reduce the number of devices, N, in the static power equation above to try to ease the problem of leakage powers. The architecture model in our design is a system with an instruction set to support the control of power gating in the component levels. From the viewpoints of engineering equation for static power, we attempt to reduce the number of devices by turning them off when they are unused. Our work provides compiler solutions in giving analysis and scheduling for the power gating control at component levels. Our compiler gives an analysis framework to utilize the instruction to reduce the leakage power. A data-flow analysis framework is given to estimate the component activities at fixed points of programs with the consideration of pipelines of architectures. We also give the equation for the compiler to decide if the employment of the power gating instructions on given program blocks will benefit the total energy reductions. As the duration of power gating on components on given program routines is related to program branches, we propose a set of scheduling policy including *Basic_Blk_sched, MIN_Path_Sched, AVG_Path_Sched* mechanisms and evaluate the effectiveness of those schemes. *Basic_Blk_Sched* mechanism schedules the power gating instructions according to the component activities in a given basic block. *MIN_Path_Sched* mechanism schedules the power gating instructions by assuming the minimum length among plausible program paths. *AVG_Path_Sched* schedules the power gating instructions by assuming the average length among plausible program paths. *MIN_Path_Sched* and *AVG_Path_Sched* mechanisms proposed in our work are based-on a depth-first traversal scheme to look up the interval in inserting power gating intsructions for components to reduce static power. Our experiment is done by incorporating our compiler analysis and scheduling policy into SUIF compiler tools [30, 31] and by simulating the energy consumptions on Wattch [6] toolkits. Experimental results show our mechanisms are very effective in reducing leakage powers on microprocessors. This work is also a part of our efforts in DTC (design technology center) of our university to develop compiler toolkits[24, 17, 38, 18, 37, 10, 9] for high-performance and low-power microprocessors and SoC designs.

2 Machine Architecture

The architecture model in our design is a system with an instruction set to support the control of power gating in the component levels. Figure 3 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 the clock. It can be done by forcing transistors to be off or using multithreshold voltage CMOS technology (MTCMOS) to increase threshold voltage [8, 22, 29].

We build the experimental architecture within the Wattch simulation environment [6]. In the simulation environment, we can measure every CPU components's power consumption of the whole experimental program. Basically, this architecture is compatible with the DEC Alpha 21264 processor [13]. The length of an instruction in our experimental architecture is 32 bits. Memory addressing is byte address. As Alpha is a 64-bit processor and uses 64-bit data bus, our experimental architecture has 32 integer registers (R0 through R31), and each is 64 bits wide. The major difference of these two architectures is the additional "power gating" design in our experimental architecture.

Those 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" for short). The PGCR is a 64-bit integer register. In this case, the only lowest 4 bits of this register

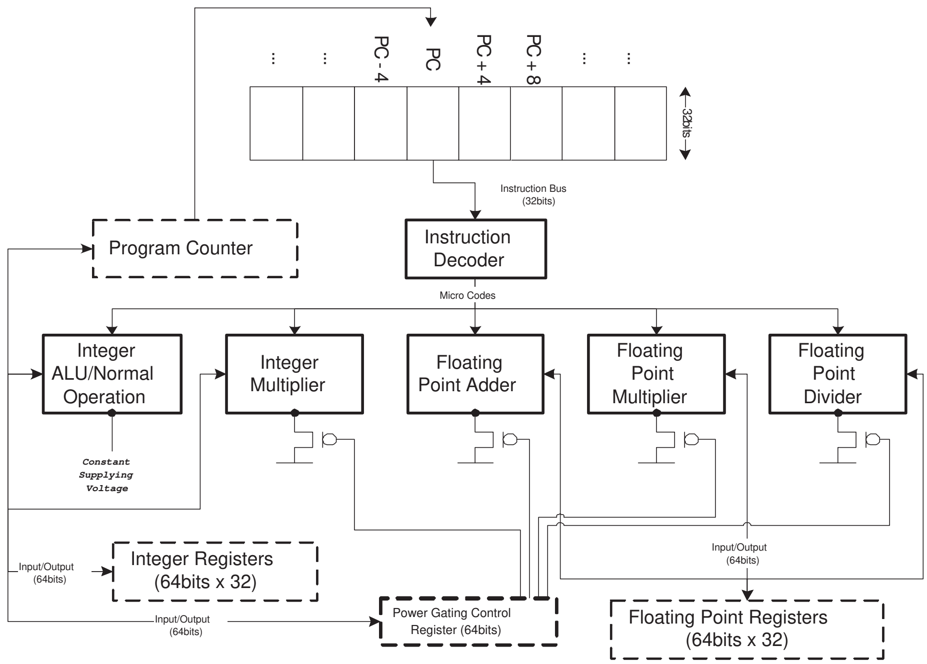

Fig. 3. Machine Architecture Model with Power Gating Control

can affect the power gating status. The 0th bit of the lowest 4 bits of the PGCR controls the power gating of the Integer Multiplier. Setting of the bit will cause the Integer Multiplier to be turned on. Clearing of the bit will turn off the corresponding function unit in the immediately following clock. The 1st bit of the 4 bits is for Floating Point Adder, the 2nd bit is for Floating Point Multiplier, and the 3rd bit of the 4 bits is for the Floating Point Divider. Worth to mention, the Integer ALU unit within architecture also takes response to execute general operation. And, it performs the data movement to the PGCR, too. As a result of the Integer ALU is always required, this function unit is always turned on. In addition, we invoke a new instruction in the simulation environment to specify the access direction of PGCR. This instruction can operate those 4 power gated function units at once by move a proper value from a general purpose register to the PGCR.

Figure 3 is also the architecture model on which we carry out our experiments later in Section 5.

3 Component-Activity Data-Flow Analysis

In this section, we investigate the compiler analysis techniques to ease the problem of leakage powers. We present a data-flow analysis framework [2] for a compiler to analyze the inactive states of components on a microprocessor. The pro-

```
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
        for each component C that will be used by B do begin  /* computation of comp_gen */
            RemainingCycle[B][C] := N,
                where N is the number of cycles needed for C by B;
            comp_gen[B] := comp_gen[B] ∪ C;
        end
        comp_in[B] := comp_kill[B] := ∅;
        comp_out[B] := comp_gen[B];
    end
    while changes to any comp_out occur do begin              /* iterative analysis */
        for each block B do begin
            for each component C do begin                     /* computation of comp_kill */
                RemainingCycle[B][C] := MAX(RemainingCycle[P][C]) − 1,
                    where P is a predecessor of B;
                if RemainingCycle[B][C] = 0 then comp_kill[B] := comp_kill[B] ∪ C;
            end
            /* computation of comp_in */
            comp_in[B] := ⋃ comp_out[P], where P is a predecessor of B;
            /* computation of comp_out */
            comp_out[B] := comp_gen[B] ∪ (comp_in[B] − comp_kill[B]);
        end
    end
End
```

Fig. 4. Data-Flow Analysis Algorithm for Component Activities

cess 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. We then try to 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 a component-activity c is generated at a block B if a component is required for this execution, symbolized as $comp\_gen[B]$, and it is killed if the component is released by the last request, symbolized as $comp\_kill[B]$. We then create 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 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. We have the data flow equation for these two groups below,

$$comp\_in[B] = \bigcup_{\substack{P\ a\ pred- \\ essor\ of\ B}} comp\_out[P]$$

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

We use an iterative approach to compute the desired results of $comp\_in$ and

Instruction	Component	Latency
Inst_1	ALU	3
Inst_2	Multiplier	4
Inst_3	Divider	2
Inst_4	Data Bus	1
Inst_5	ALU	2
	Data Bus	2
Inst_6	others	-

Table 1. Resource Utilization Table (Instruction Latency in Cycles)

comp_out after *comp_gen* have been computed for each block. The algorithm is sketched in Figure 4. This is an iterative algorithm for data flow equations [2] with the additions of resource management strcutures. A two-dimension array, called $RemainingCycle$, is used to maintain the number of required cycles to achieve requests for each component and block. In addtion, a resource utilization table is adopted to give the resource requirement for each instruction of given processors. The resource utilization table can be used to give the initial values of $RemainingCycle$. The remaining cycles of a component will be decreased by one for each propagation. Initially, both *comp_in* and *com_kill* are set to be empty. The iteration goes until the *comp_in* (and hence the *comp_out*) converges. As *comp_out*$[B]$ never decreases in size for any B, the algorithm will eventually halt while all *comp_out* are steady. Intuitively, the algorithm propagates activities of components as far as they will go by simulating all possible executing paths of the program. This algorithm gives the state of utilization of components for each point of programs.

In the following, we give a running example to illustrate the idea of our algorithm. Figure 5 shows a program fragment represented in the form of a control flow graph. We will use this graph an exmaple to illustrate how the data flow analysis was done to gather component activitiy information. To give initial values to the set in the data flow equation, we need to first establish a resource utilization table. This can be done by looking up hardware reference manuals or retrieving information from architecture description languages [16]. Table 1 gives the resource utilization table needed for our example. In the resource utilization table, it gives each type of instructions the resources needed and the number of cycles for the executions. The number of cyles will be used for the initial values of remaining cycles in the algorithm. We then can give values with *comp_gen*$[B]$ for each program block B. Table 2 shows the resource requirements and *comp_gen* for each block. For example, B_2 generates a component-activity of ALU, represented by a bit vector, 1000, and B_3 generates that of data bus, represented by a bit vector 0001.

Once the *comp_gen*$[B]$ is calculated, the initial values of *comp_in*$[B]$, *comp_kill*$[B]$, and *comp_out*$[B]$ for each block B can be set. The initial information is listed in the initial phase column in Table 3. Then we move on with

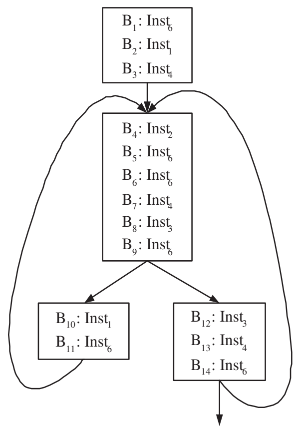

Fig. 5. Control Flow Graph

Block	*RemainingCycle*				*comp_gen*	
	ALU	Mult	Div	DBus		
B_1	0	0	0	0	∅	0000
B_2	3	0	0	0	{ALU}	1000
B_3	0	0	0	1	{DataBus}	0001
B_4	0	4	0	0	{Multiplier}	0100
B_5	0	0	0	0	∅	0000
B_6	0	0	0	0	∅	0000
B_7	0	0	0	1	{DataBus}	0001
B_8	0	0	2	0	{Divider}	0010
B_9	0	0	0	0	∅	0000
B_{10}	3	0	0	0	{ALU}	1000
B_{11}	0	0	0	0	∅	0000
B_{12}	0	0	2	0	{Divider}	0010
B_{13}	0	0	0	1	{DataBus}	0001
B_{14}	0	0	0	0	∅	0000

Table 2. Computation of *comp_gen*

Block	Initial			Pass 1			Pass 2		
	comp_in	*comp_kill*	*comp_out*	*comp_in*	*comp_kill*	*comp_out*	*comp_in*	*comp_killt*	*comp_out*
B_1	0000	0000	0000	0000	0000	0000	0000	0000	0000
B_2	0000	0000	1000	0000	0000	1000	0000	0000	1000
B_3	0000	0000	0001	1000	0000	1001	1000	0000	1001
B_4	0000	0000	0100	1001	0001	1100	1001	0001	1100
B_5	0000	0000	0000	1100	1000	0100	1100	1000	0100
B_6	0000	0000	0000	0100	0000	0100	0100	0000	0100
B_7	0000	0000	0001	0100	0000	0101	0100	0000	0101
B_8	0000	0000	0010	0101	0101	0010	0101	0101	0010
B_9	0000	0000	0000	0010	0000	0010	0010	0000	0010
B_{10}	0000	0000	1000	0010	0010	1000	0010	0010	1000
B_{11}	0000	0000	0000	1000	0000	1000	1000	0000	1000
B_{12}	0000	0000	0010	0010	0010	0010	0010	0010	0010
B_{13}	0000	0000	0001	0010	0000	0011	0010	0000	0011
B_{14}	0000	0000	0000	0011	0011	0000	0011	0011	0000

Table 3. Computation of *comp_in*, *comp_kill*, and *comp_out*

iteration phases. For B_1 block, there are no predecessors so that $comp\_in[B_1]$ remains the empty set, represented by 0000, and $comp\_out[B_1]$ remains to be $comp\_gen[B_1]$. Next, for B_2 block, $comp\_in[B_2]$ is equal to the union of the *comp_out* of its predecessors. In this case, they are $comp\_out[B_1]$, and $comp\_out[B_2]$. The component-activities are then propagated until they are killed. For instance, the component-activity of ALU is killed at B_4 since *RemainingCycle* is decreased to zero. The *comp_in* and *comp_out* of B_4 are computed by

$$
\begin{aligned}
comp\_in[B_4] &= com\_out[B_3] \cup comp\_out[B_{11}] \cup comp\_out[B_{14}] \\
&= 1001 + 0000 + 0000 = 1001
\end{aligned}
$$

$$
\begin{aligned}
comp\_out[B_4] &= com\_gen[B_4] \cup (comp\_in[B_4] - comp\_kill[B_4]) \\
&= 0100 + (1001 \text{ - } 0001) = 1100
\end{aligned}
$$

The computation is summarized in Table 3. After the second pass, there are no changes in any of the *comp_out* sets, so the algorithm converges. This gives the the information of the utilization of components at fixed points of a program.

4 Leakage Power Reduction

In this section, we present a cost model for decisions if power gating control should be applied and a set of scheduling policies to place power-gating instruction sets for given programs.

4.1 Cost Model

With the utilization of components obtained from last section, we can insert power gating instructions into programs at proper points, the head/tail of an inactive block, to turn off/on useless components. This can reduce the leakage power. However, both shutdown and wakeup procedures take additional penalty, especially for wakeup process due to peak voltage. The following gives our cost model for deciding if the insertions of energy instructions will profit in energy consumptions.

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

where $E_{turn\_off}(C)$ is the penalty of energy for shutting down component C, E_{turn_on} is the penalty of energy 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 by employing power gating controls. The left-hand side of the equation shows the energy consumed by shutdown and wakeup procedures, and the right-hand side shows the leakage energy consumed for a certain cycles. It will save power for power gating control only if the amount of power of shutdown and wakeup is less than the one at RHS.

While employing power gating, there is another thing we should note. It's the latency to turn a component on. Due to the high capacitance on the circuits, several clock cycles will be needed to bring the component back to the normal operating state. Butts et al. also illustrated that it takes about 7.5 cycles at 1 GHz to charge 5 nF to 1.5 V with 1A [8]. 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 recover. Our cost model after incorporating latency issue is now as follows.

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

where $\mathbf{Latency_C}$ is the power gating latency of component C. In addition, we will try to insert the on operations of the power-gating control to be ahead of the time for the use of the corresponding components to avoid the delay in the programs due to wakeup latency.

4.2 Scheduling Policies for Power Gating

With the component activity information gathered and the cost model for deciding if the power-gating instructions should be employed, we now give the scheduling mechanisms to place the power gating instructions for given programs. As the duration of power gating on components is related to conditional branches in programs, we propose a set of scheduling policy including *Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched* to schedule power gating instructions. The details are given below.

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

Next, an inactive block of a component may cross over more than two adjacent basic blocks. We use a *depth-first-traveling* algorithm to traverse all possible executing paths. In general, an inactive block will be turned off while the criteria reached. In the case of conditional branches occurred in the inactive block, there should be an another consideration to take action of power gating. This is because the size of the two inactive blocks, which are targets that the branch instruction points to, may be different. There may be a situation that one of the branchings benefits for power gating while the other doesn't. It will be against power reductions if we take control of power gating considering only one branch but the other branch is taken. Hence, we propose a *MIN_Path_Sched* policy to ensure that power gating control would be activated only if 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 6 presents the details for *MIN_Path_Sched* algorithm. Given a control flow graph annotated with component utilizations, we define a integer variable, *Count*, to maintain the inactive length so far. It is passed around from parent blocks to successors and increased for each passing. The algorithm is recursive to guarantee the accuracy of *Count* and to ensure all paths being traversed. The algorithm is divided into four parts to handle conditions when

```
Input   A control flow graph annotated with component utilizations.
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 ∉ 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) > Threshold_C and !Branched then
                schedule power gating instructions at the head and tail of inactive blocks;
            return MIN(l_Count, r_Count);

        /* condition 2; conditional branch, active */
        else
            if Count > Threshold_C and !Branched then
                schedule power gating instructions at the head and tail 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 ∉ 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 > Threshold_C and !Branched then
                schedule power gating instructions at the head and tail 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. 6. *MIN_Path_Sched* Algorithm Based on Depth-First-Traveling for Power Gating

encountering/non-encountering a conditional branch while the analyzing component is active/inactive.

1) A conditional branch reaches and the component is inactive: under this condition, current traveling will halt until both two branches having done their travelings. And then, it makes a judgement on power gating when no branch encountered before and return the minimum inactive length of two branchings.

2) A conditional branch reaches and the component is active: under this condition, it takes control of power gating if necessary, starts two new travelings for both branchings, and finally returns the current inactive length.

3) Any statement except conditional branches reaches and the component is inactive: under this condition, it only continues the current traveling, that is, it only increases $Count$ for passing and returns.

4) Any statement except conditional branches reaches and the component is active: like condition 2, it takes control of power gating if necessary and starts a new traveling for its successor. And finally, it returns $Count$.

Note that cares have to be taken for recursive boundaries to reach the backward edges for a loop. As a depth-first search algorithm can find out the loop, the cycle situation can be known in our algorithm. In a cycle situation, if the the whole instructions used in the cycle of a program fragment does not use the component in the search, we will assume the loop cycle is executed for once with the minimum path scheduling policy. If some instructions in the backward edge of a program fragment does use the component in the search, the the backward edge extend to that instruction will be counted for the program path. In addition, since our proposed algorithm is based on the depth-first-traveling, the complexity of our approach is $O(N)$ where N is the size of nodes in a control flow graph.

Next, as the behavior of program branches depends on the structure and the input data of programs, some branches may be taken rarely or even not taken. To accomodate this issue, 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 judgements made in condition 1 above. AVG_Path_Sched returns the average length of two branchings instead of the minimums. With this scheme, it will take advantage of power reduction if a unusual-taken branch returns a small value of $Count$ which causes power gating mechanism inactivated. AVG_Path_Sched mechanism can be approximately done by assuming the probabilities of all branches are 50%, by assinging branch probabilities at compiler time by programmers or compilers or by incorporating path profiling schemes to examine the probabilities of all branches.

5 Experimental Results

5.1 Platform

We use an Alpha-compatible architecture with power gating control and instruction sets described in Figure 3 of Section 2 as the target architecture for our experiments. The proposed data-flow analysis and scheduling policies are incorporated into the compiler tool with SUIF [31] and MachSUIF Library [30] and evaluated by Wattch simulator [6]. Figure 7 shows the structure of the framework including compilation and simulation parts. In the compilation part, we use the SUIF library to perform compiler optimization for performances and the MachSUIF Library to perform machine-dependent optimizations for Alpha processors. After those optimizations having been done, we then analyze component activities with our proposed data-flow equation and schedule power gating instructions to reduce leakage dissipation. Finally, the compiler generates the Alpha assembly code with power gating instructions. To be recognized by Alpha assembler and linker, power gating instructions are replaced by an instruction sequence within the Alpha instruction set with annotation information to simulators. We then use the Wattch power estimator, which is based on the SimpleScalar [7] architectural simulator, to simulate power dissipation and to evaluate our approach. The SimpleScalar is a simulator that provides execution-driven and cycle-accurate simulations for various instruction set architectures (include our target architecture, Alpha ISA). Both SimpleScalar and Wattch are now widely used for simulations to evaluate performance and power dissipation [6]. We also do refinements on the Wattch estimator to catch the instruction sequences for power gating control.

5.2 Results

The test suits in our experiment are the common benchmarks listed in FAQ of comp.benchmarks [1]. Figure 8 and Figure 9 illustrate the power results for the simulations of power gating control over Floating Point Adder and Floating Point Multiplier for *nsieve* application, respectively. In these figures, the X-axis represents the break-even cycle for our scheduling criteria and the Y-axis represents the power consumption. The leftest bar shows the power dissipated by function units while no power gating control being employed. This is the results of traditional clock gating mechanism provided by the Wattch power estimator. This is the version we use as the base version for comparison. The clock gating mechanism gates the clocks of those unused resources in multi-ported hardware to reduce the dynamic power. However, there is still static power leaked. Wattch assumes that clock-gated units dissipate 10% of their maximum power, rather than drawing zero power. For example, the clock gating mechanism reduces about 30% of total power consumption against the one without clock gating for several SPECint95 and SPECfp95 benchmarks [6]. The rest bars of the figures give the power gating results for the proposed scheduling policies with different break-even cycle. The results show that the power gating mechanism reduces a large

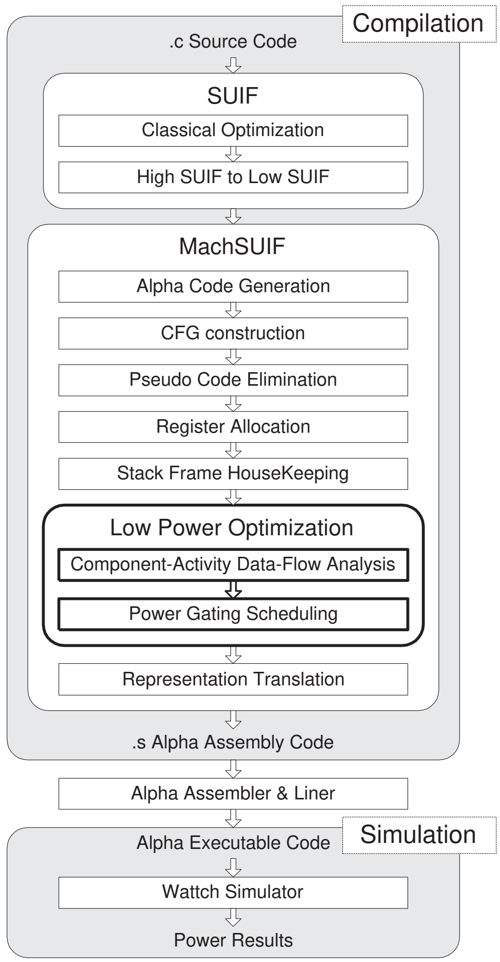

Fig. 7. Our Experimental Framework

amount of leakage power even if the penalty of power gating control is high (i.e., large break-even cycle). Note that we have incorporated the penalty of inserting power gating instructions into our power simulator, Wattch. In our experimental data, it also indicates the MIN_Path_Sched and the AVG_Path_Sched scheduling algorithms always perform better results than the $Basic\_Blk\_Sched$. This is because the $Basic\_Blk\_Sched$ algorithm schedules power gating instructions within basic blocks while the other two schedule those beyond branches. It will extend the possible inactive lengths for components while the MIN_Path_Sched or the AVG_Path_Sched is employed. The AVG_Path_Sched mechanism used in our implementation is an approximation by assuming the probabilities of all branches are 50%. We think a more accurate model by incorporating path profiling schemes can further improve the results. The reduction of the power

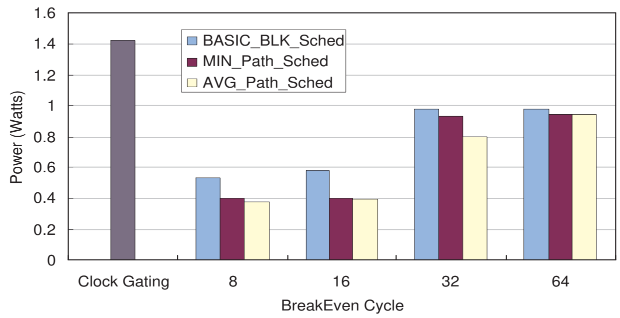

Fig. 8. Results of Floating Point Adder for *nsieve*

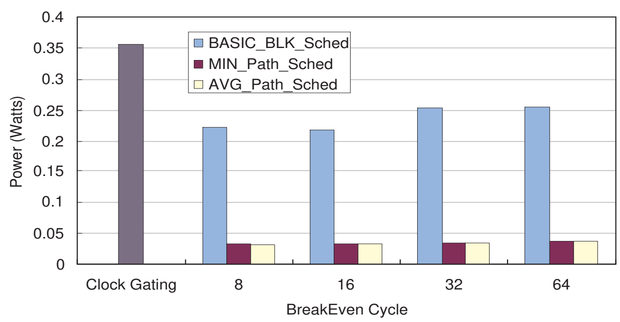

Fig. 9. Results of Floating Multiplier for *nsieve*

consumed by the Floating Point Adder is from 30.11% to 70.50%, 30.36% to 77.01% and 31.36% to 77.68% for the $Basic\_Blk\_Sched$, MIN_Path_Sched and AVG_Path_Sched, respectively. And that of the Floating Point Multiplier is from 28.28% to 39.58%, 89.67% to 91.41% and 89.67% to 91.25%, respectively.

Figure 10 and Figure 11 give the power consumption of the Floating Point Adder and the Floating Point Multiplier for various benchmarks while employing the power gating mechanism with break-even cycle 32. Once again, it is observed that the AVG_Path_Sched benefited the most power reduction while the MIN_Path_Sched came second and the $Basic\_Blk\_Sched$ finished third. However, they always have better results than the one without power gating (and hence only clock gating employed). Figure 10 shows that the $Basic\_Blk\_Sched$ policy has an average 23.05% reduction for all benchmarks while the MIN_Path_Sched and the AVG_Path_Sched have 76.93% and 82.84%, respectively. In the case of *hanoi* benchmark, which is an integer program, it even reduces 99.03% of power for the $Basic\_Blk_Sched$ and 99.69% for the MIN_Path_Sched and AVG_Path_Sched. Similar results can also be summarized in Figure 11.

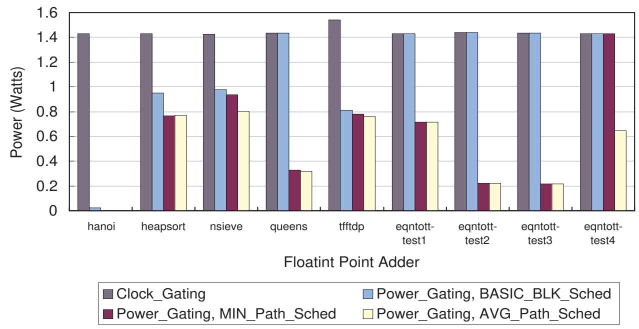

Fig. 10. Power Gating on Floating Point Adder for miscellaneous benchmarks ($BreakEven = 32$)

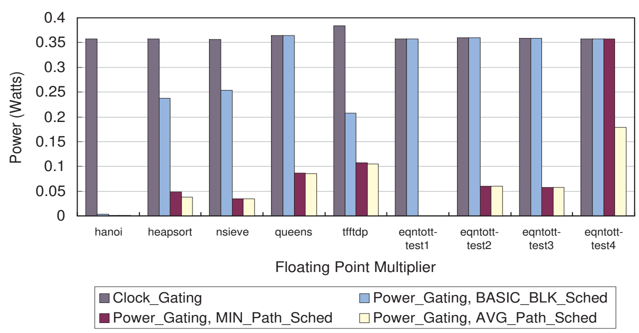

Fig. 11. Power Gating on Floating Point Multiplier for miscellaneous benchmarks ($BreakEven = 32$)

With the significant reduction on the energy consumption for each component in the experiment, one might be interested in seeing the energy reduction percentage in terms of total energy consumption. We now give some reference points of our experimental data below. With the break-even cycle set to be 32 and the power gating control on Integer ALU and Floating Point ALU, our experiment shows that it will reduce 13.19% of total power for *queens* and 14.58% for *hanoi*. Figure 12 gives additional information for the total energy reduction for *nsieve*. It shows the total power consumption of *nsieve* with different size of break-even cycles. The reduction of the leakage power in terms of total energy consumption for *nsieve* is from 4.01% to 10.71% while employing the $Basic\_Blk\_Sched$ algorithm. With the AVG_Path_Sched, it improves the reduction from 4.17% to 13.38%.

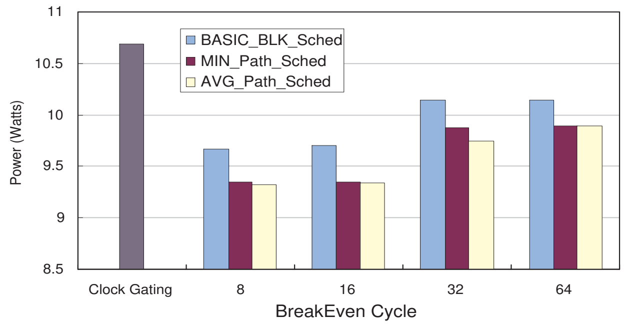

Fig. 12. Total Power Results of *nsieve* (While Integer ALU, Floating Point Adder, and Floating Point Multiplier are considered for power gating)

6 Related Work

Several research groups have proposed and developed hardware techniques to reduce dynamic and static power dissipation in recent. Recent work by Powell et al. architectural and circuit-level techniques to reduce the power consumption in instruction caches [26]. The cache miss rate is used to determine the working set size of the application relative to that of the cache. Leakage power is then removed from the unused SRAM cells using gated-V_{dd} transistors. Kaxiras et al. also attacks leakage power in cache memories. Policies and implementations for reducing cache leakage by invalidating and turning off cache lines when they enter a dead period are4 discussed [23]. This leads to power savings in the cache. Zhang et al. presented a compiler approach which exploits schedule slacks in VLIW architectures to optimize leakage and dynamic energy consumption [39]. However, this approach has limitations on what it can achieve. It lacks a data flow analysis framework to exploit large slacks which may cross more than one basic block.

7 Conclusions

In this paper, we investigated the compiler analysis techniques 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 in the component levels. We presented a data flow analysis framework to estimate the component activities at fixed points of programs with the consideration of pipelines of architectures. A set of scheduling policy including *Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched* mechanisms were proposed and evaluated. Experimental results show our mechanisms are effective in reducing leakage powers on microprocessors.

References

1. Al Aburto, *collections of common benchmarks of FAQ of comp.benchmarks USENET newsgroup*, ftp site: ftp.nosc.mail/pub/aburto.
2. A. Aho, R. Sethi, J. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1985.
3. M. Alidina, J. Monteiro, S. Devadas, A. Ghosh and M. Papaefthymiou, "Precomputation-Based Sequential Logic Optimization for Low Power," *Proc. of ICCAD-94*, pp. 74-81, 1994.
4. Luca Benini and G. De Micheli, "State Assignment for Low Power Dissipation," *IEEE Journal of Solid State Circuits*, Vol. 30, No. 3, pp. 258-268, March 1995.
5. Nikolaos Bellas, Ibrahim N. Hajj, and Constantine D. Polychronopoulos, "Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 317-326, June 2000.
6. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th. International Symposium on Computer Architecture*, pp. 83-94, June 2000.
7. D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Architecture News,* pp. 13-25, June 1997.
8. J. Adam Butts and Gurindar S. Sohi, "A Static Power Model for Architects," *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 191-201, December 2000.
9. R. G. Chang, T. R. Chuang, Jenq-Kuen Lee. "Efficient Support of Parallel Sparse Computation for Array Intrinsic Functions of Fortran 90," ACM International Conference on Supercomputing, Melbourne, Australia, July 13-17, 1998.
10. Rong-Guey Chang, Jia-Shing Li, Tyng-Ruey Chuang, Jenq Kuen Lee. "Probabilistic inference schemes for sparsity structures of Fortran 90 array intrinsics," International Conference on Parallel Processing, Spain, Sep. 2001.
11. A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low-Power CMOS Digital Design," *IEEE Journal of Solid-State Circuits*, Vol. 27, No.4, pp. 473-484, April 1992.
12. Jui-Ming Chang, Massoud Pedram, "Register Allocation and Binding for Low Power," *Proceedings of Design Automaton Conference*, San Francisco, USA, June 1995.
13. Compaq Computer Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, EC-RJRZA-TE, (July 1999).
14. V. De and S. Borkar, "Technology and design challenges for low power and high performance," *Proc. of Int. Symp. Low Power Electronics and Design*, pp. 163-168, 1999.
15. G. Hachtel, M. Hermida, A. Pardo, M. Poncino and F. Somenzi, "Re-Encoding Sequential Circuits to Reduce Power Dissipation," *Proc. of ICCAD' 94*, pp. 70-73, 1994.
16. G. Hadjiyiannis, S. Hanono and S. Devadas. "ISDL: An Instruction Set Description Language for Retargetability," *Design Automation Conference*, June 1997
17. Yuan-Shin Hwang, Peng-Sheng Chen, Jenq-Kuen Lee, Roy Ju. "Probabilistic Points-to Analysis," *LCPC '2001*, Aug. 2001, USA.
18. Gwan-Hwan Hwang, Jenq Kuen Lee, Roy Dz-Ching Ju. "A Function-Composition Approach to Synthesize Fortran 90 Array Operations," *Journal of Parallel and Distributed Computing*, 54, 1-47, 1998.

19. Inki Hong, Darko Dirovski, et.al., "Power Optimization of Variable Voltage Core-Based Systems," *Proc. of 35th DAC*, pp. 176-181, 1998.
20. M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-Power Digital Design," *Proceedings of the 1994 IEEE Symposium on Low Power Electronics*, pp. 8-11.
21. Intel corporation, "Pentium III Processor for the SC242 at 450 MHz to 1.13 GHz Datasheet," pp. 26-30.
22. J. T. Kao and A. P. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *IEEEJournal of Solid-state circuits*, 35(7):1009-1018, July 2000.
23. S. Kaxiras, Z.Hu and M.Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *Proc. of the Int'l Symposium on Computer Architecture*, pp.240-251, 2001.
24. Chingren Lee, Jenq Kuen Lee, TingTing Hwang, and Shi-Chun Tsai, "Compiler Optimization on Instruction Scheduling for Low Power," *Proceedings of the 13th International Symposium on Systems Synthesis*, pp. 55 - 60, September 2000.
25. Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, Masahiro Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Transactions on VLSI Systems*, Vol. 5, no. 1, pp. 123-133, March 1997.
26. M.D. Powell, S-H. Yang, B. Falsa, K. Roy, and T.N. Vijaykumar, "Gated-Vdd: a Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories," *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2000.
27. S. C. Prasad and K. Roy, "Circuit Activity Driven Multilevel Logic Optimization for Low Power Reliable Operation," *Proceedings of the EDAC'93 EURO-ASIC* , pp. 368-372, Feb., 1993.
28. K. Roy and S. C. Prasad, "SYCLOP: Synthesis of CMOS Logic for Low Power Applications," *Proceedings of the ICCD*, pp. 464-467, 1992.
29. K. Roy, "Leakage Power reduction in Low-Voltage CMOS Designs," *IEEE International Conference on Circuits and Systems*, Vol. 2, pp. 167-173, 1998.
30. Michael D. Smith, "The SUIF Machine Library", *Division of of Engineering and Applied Science, Harvard University*, March 1998.
31. Stanford Compiler Group, "The SUIF Library", *Stanford Compiler Group*, Stanford, March 1995.
32. Ching-Long Su and Alvin M. Despain, "Cache Designs for Energy Efficiency," *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pp. 306 -315, 1995.
33. V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez, "Dynamic Power Management for Microprocessors: A Case Study," *Proceedings of the 10th International Conference on VLSI Design*, pp. 185-192, 1997.
34. V. Tiwari, D.Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing Power in High-Performance Microprocessors," *Proceedings of the Design Automaton Conference*, pp. 732-737, 1998.
35. Scott Thompson, Paul Packan, and Mark Bohr, "MOS Scaling: Transistor Challenges for the 21st Century," Portland Technology Development, Intel Corp. *Intel Technology Journal*, Q3 1998.
36. C.Y. Tsui, M. Pedram, and A.M. Despain, "Technology Decomposition and Mapping Targeting Low Power Dissipation," *Proc. of 30th Design Automaton Conf.*, pp.68-73, June 1993.
37. J. Z. Wu, Jenq-Kuen Lee. "A bytecode optimizer to engineer bytecodes for performances," *LCPC 00*, Aug. 2000, USA (Also in LNCS 2017).

38. Yi-Ping You, Ching-Ren Lee, Jenq-Kuen Lee, Wei-Kuan Shih. "Rea-Time Task Scheduling for Dynamically Variable Voltage Processors," *IEEE workshop on Power Management for Real-Time and Embedded Systems*, May 2001.
39. W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y. Tsai. "Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction," *Proceedings of the Thirty-Fourth Annual International Symposium on Microarchitecture (MICRO-34)*. pp. 102-113. Austin, TX. December 2001.