A Sink-N-Hoist Framework for Leakage Power Reduction

Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 30013, Taiwan
{ypyou, cwhuang}@pllab.cs.nthu.edu.tw, jklee@cs.nthu.edu.tw

ABSTRACT

Power leakage constitutes an increasing fraction of the total power consumption in modern semiconductor technologies. Recent research efforts have tried to integrate architecture and compiler solutions to employ power-gating mechanisms to reduce leakage power. This approach is to have compilers perform data-flow analysis and insert instructions at programs to shut down and wake up components whenever appropriate for power reductions. While this approach has been shown to be effective in early studies, there are concerns for the amount of power-control instructions being added to programs with the increasing amount of components equipped with power-gating control in a SoC design platform. In this paper, we present a *Sink-N-Hoist* framework in the compiler solution to generate balanced scheduling of power-gating instructions. Our solution will attempt to merge power-gating instructions as one compound instruction. Therefore, it will reduce the amount of power-gating instructions issued. We perform 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 the amount of power-gating instructions while further in reducing leakage power compared to previous methods.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

Algorithms, Experimentation, Languages

Keywords

Compilers for low power, data-flow analysis, leakage power reduction, balanced scheduling, power-gating mechanisms

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

1. INTRODUCTION

Minimization of power dissipation can be considered at algorithmic, architectural, logic, and circuit levels [4]. Studies on low power design are abundant in the literature 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. [1, 5, 8, 11, 12, 17, 19, 20]. In order to reduce the dynamic power, several research work have been proposed to reduce the dissipation. For example, software rearrangements to utilize the value locality of registers [5], the swapping of operands for booth multiplier [12], the scheduling of VLIW instructions to reduce the power consumption on the instruction bus [11], gating clock to reduce workloads [8, 19, 20], cache sub-banking mechanism [17], the utilization of instruction cache [1], etc.

As semiconductor technology continues to scale down, the leakage power gains more significance in the total power dissipation. It is predicted that the leakage power will become comparable to the dynamic power in only a few generations [18]. Therefore, power gating in addition to clock gating should be used to reduce both leakage power and dynamic power, as clock gating is only able to reduce the dynamic power [3, 10]. Recent research efforts have tried to integrate architecture and compiler solutions to employ power-gating mechanisms to reduce leakage power [6, 13, 22–25]. This approach is to have compilers perform data-flow analysis and insert instructions at programs to shut down and wake up components whenever appropriate for power reductions. While this approach has been shown to be effective in early studies, there are concerns for the amount of power-control instructions being added to programs with the increasing amount of components equipped with power-gating control in a SoC design platform for embedded systems. Note that architecture designers can custom the processor with unique operation functions [7, 9, 21]. Examples of these modules are abundant. For example, one may have extensible instructions for crypto modules, 3D graphic modules, motion estimation modules, a variety of wireless communication modules, etc.

In this paper, we present a Sink-N-Hoist framework in the compiler solution to generate balanced scheduling of power-gating instructions. Our solution will attempt to merge power-gating instructions as one compound instruction. Therefore, it will reduce the amount of power-gating instructions issued. Note that power-gating instructions

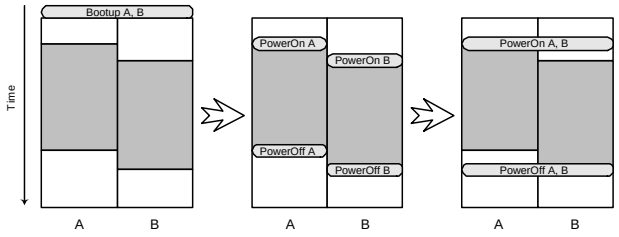

Figure 1: Scenario of power-gating control (the shadowed area indicates that the component is in use).

can significantly reduce leakage power, but produce recovery penalties, increase the execution time of programs, and increase code sizes of programs. Figure 1 illustrates an example of power-gating controls. In the LHS of the figure, it shows two different components in use. Next, the current practice will attempt to issue power-on and power-off instructions at programs for these two hardware components separately. The one in the RHS of Figure 1 shows our scheme to try to merge these instructions. In our research work, we will provide cost model and a software foundation to guide this process. Our solution includes a set of data-flow equations for code motions of power-gating instructions. Our work gives a theoretical foundation and step-by-step framework to group power-gating instructions, together. We perform experiments by incorporating our compiler analysis and scheduling policies into SUIF compiler tools and by simulating the energy consumptions on a platform integrating Wattch toolkits [2]. The experimental results done with DSP-stone benchmark demonstrate that our mechanisms are effective in reducing the amount of power-gating instructions as well as producing power reductions over previous methods. It results in average 31.2% of reduction in the amount of power-gating instructions over the scheme without incorporating our *Sink-N-Hoist* framework for merging power-gating instructions. In fact, we further reduce the energy consumption in our framework. This is due to that the effect of a block version of power-gating instructions gives better power and performance effects than the pointwise version of power-gating instructions.

The remainder of this paper is organized as follows. Section 2 describes a machine architecture for the target platform. Section 3 overviews the leakage-power reduction framework. Section 4 presents our analysis and merging techniques for reducing the amount of power-gating instructions. Section 5 gives the experimental results of our work. Finally, Section 6 concludes this work.

2. MACHINE ARCHITECTURE

The architecture model in our design is a modern system with an instruction set to support the control of power gating in the component level. We focus on reducing the power consumption of the certain components by invoking the power gating technology. Power gating is analogous to clock gating; power gating powers off devices by switching off their supply voltage rather than the clock. It can be done by forcing transistors to be off or using multi-threshold voltage CMOS technology (MTCMOS) to increase threshold voltage [3, 10, 14].

Figure 2 illustrates an example of our target machine architecture based on Alpha 21264 processor having the integer function unit (Execution Box) and the floating point function unit (Floating-point Box). In the adapted ALPHA 21264 architecture model, the E box and F box were added the power-gated functions. The power gating of each unit can be controlled by the "Power Gating Control Register" ("PGCR" for short). The PGCR is a 64-bit integer register. In this case, there are one bit used for Integer Multiplier and 3 bits for Floating Point Function Units. Setting the power gating bit true will cause the corresponding module to be powered on. Clearing the bit to zero will power off the corresponding module immediately in the following clock. A new instruction was implemented to control units with the power gated function by move a proper value from a general purpose register to the PGCR. The Integer ALU unit is always powered on, due to that it takes response to performs the data movement to the PGCR.

3. LEAKAGE-POWER REDUCTION FRAMEWORK

In this section, we present a compiler framework for employing power-gating mechanisms to reduce leakage power dissipation. In our earlier work, we have presented a data-flow analysis framework, called *Component-Activity Data-Flow Analysis*, to estimate the component activities on a microprocessor within a given program [23, 24]. The analysis collects the information of the utilization of components at each point in the program. After that, a power-gating instruction scheduling is performed to determine when, where, and whether power-gating control should be employed with the concern of power reduction. Finally, power-gating instructions are inserted into the program accordingly. In this research work, we present a *Sink-N-Hoist* framework, which is applied in the phase right before power-gating instructions are inserted, to generate balanced scheduling of power-gating instructions. Our solution will attempt to merge power-gating instructions as one compound instruction. Figure 3 presents our compiler flow of the leakage power reduction framework. Step (i), (ii), and (iii) are the steps in conventional methods [23, 24] and step (iv) and (v) are the steps proposed in this paper to perform mergings of power gating instructions. Among the stages of design flow, step (i) and (ii) gives component activity flow analysis, step (iii) decides where and if power gating instruction should be inserted. Next, step (iv) attempts to merge the power gating instructions with our proposed *Sink-N-Hoist* framework. Finally, step (v) performs code emits for the group case. A motivating example of power-gating control over floating-point units (a floating-point ALU, a floating-point multiplier, and a floating-point divider) with this framework is illustrated in Figure 4, where each plot shows the status of a component in timeline and the shadowed plot represents that it is in use. Three scenarios are given as follows: the leftmost figure shows the case without power-gating control, the middle one shows the case when (i), (ii), (iii), and (v) in the framework are applied, and the rightmost one shows the case when all phases in the framework are applied. The number of power-gating instructions inserted can be decreased from six to two when the *Sink-N-Hoist Analysis* is applied.

In the following, we first describe the the methods in step (ii) and (iii) and then present step (iv) and step (v) with

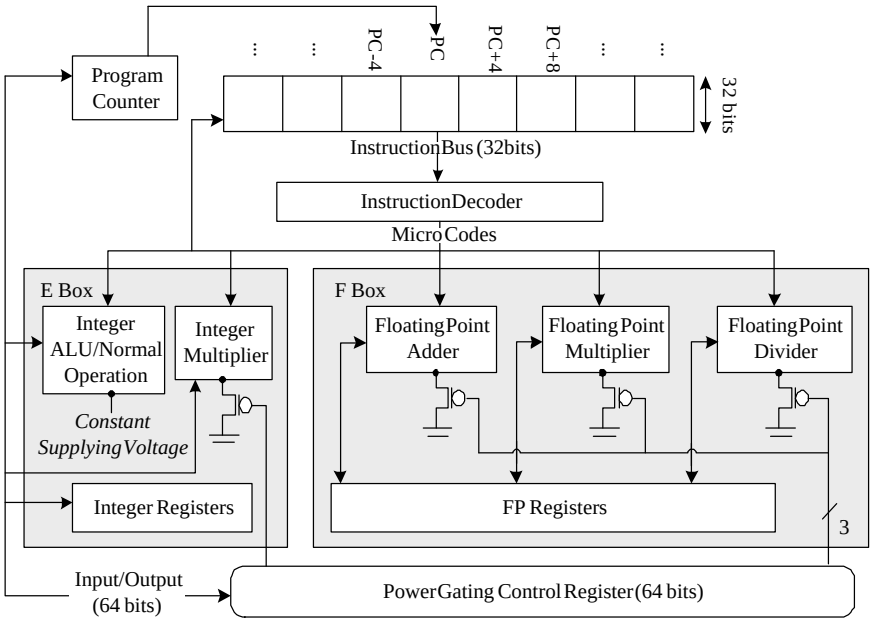

Figure 2: Alpha 21264 architecture with power gating support.

Leakage-Power Reduction Framework

Input: A source program
Output: The program with power-gating control

i. Construct the interprocedural control-flow graph of the program.

ii. Perform *Component-Activity Data-Flow Analysis*.

iii. Perform power-gating instruction scheduling.

iv. Perform *Sink-N-Hoist Analysis*.

v. Emit power-gating instructions for groups.

Figure 3: The leakage-power reduction framework.

the *Sink-N-Hoist Analysis* for code motion of power-gating instructions in Section 4.

3.1 Component-Activity Data-Flow Analysis

The goal of the *Component-Activity Data-Flow Analysis* is to collect the information of the utilization of components at each point in a program. A set of data-flow equations is proposed to compute such information. We say a *component-activity c* is generated at a block b if a component is required for the execution, symbolized as $\mathsf{COMPONENT}_{loc}(b)$, and it is killed if the component is released by the last request, symbolized as $\mathsf{COMPONENT}_{blk}(b)$. The predicates of the data-flow equations for collecting component-activity information are given as follows:

- $\mathsf{COMPONENT}_{loc}(b)$ is a set of components which are required for the first cycle of the execution.
- $\mathsf{COMPONENT}_{blk}(b)$ is a set of components which are released by the execution.
- $\mathsf{COMPONENT}_{in}(b)$ is a set of components which are required for the execution in the beginning of block b. It can be computed by

$$\mathsf{COMPONENT}_{in}(b) = \bigcup_{p \in Pred(b)} \mathsf{COMPONENT}_{out}(p),$$

where $Pred(b)$ is the set of predecessor program blocks of p.

- $\mathsf{COMPONENT}_{out}(b)$ is a set of components which are required for the execution in the end of block b. It can be computed by

$$\mathsf{COMPONENT}_{out}(b) = \mathsf{COMPONENT}_{loc}(b) \cup (\mathsf{COMPONENT}_{in}(b) - \mathsf{COMPONENT}_{blk}(b))$$

and can be read as, "the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement."

- $\mathsf{INACTIVITY}(b)$ is a set of components which are not active at block b. In fact, $\mathsf{INACTIVITY}(b)$ is the complementary set to $\mathsf{COMPONENT}_{out}(b)$, i.e.,

$$\mathsf{INACTIVITY}(b) = \Omega - \mathsf{COMPONENT}_{out}(b),$$

where Ω is the universal set.

3.2 Power-Gating Instruction Scheduling

With the utilization information of components, we can insert power-gating instructions into programs at the appropriate points (i.e. the beginning and the end of an inactive block) to power 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 a cost model for deciding if the insertion of power-gating instructions will provide energy-consumptions benefits:

$$\mathbb{P}_{leak}(C) \cdot \mathsf{ITVL}^{idle} > \mathbb{E}_{off}(C) + \mathbb{E}_{on}(C) + \mathbb{P}_{rleak}(C) \cdot \mathsf{ITVL}^{idle},$$

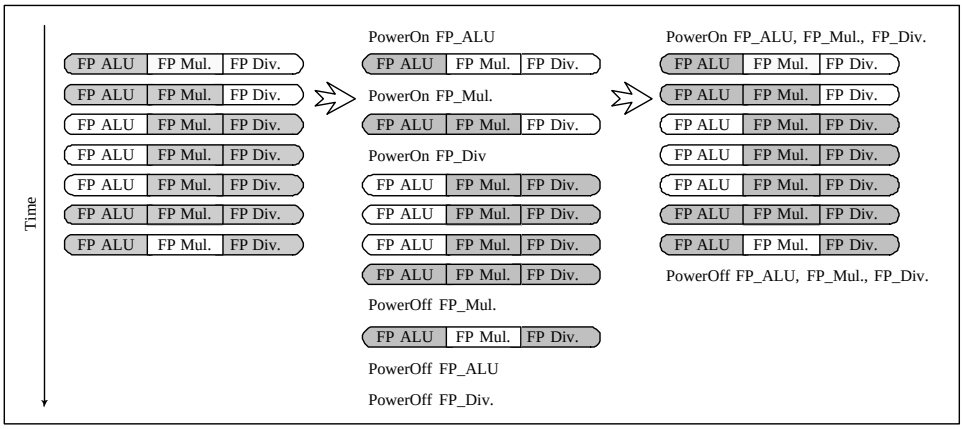

Figure 4: An example of power-gating control over floating-point units. (The shadowed components represent that they are in use.)

where functions $\mathbb{E}$ and $\mathbb{P}$ return the value of energy and power consumption, respectively; $\mathbb{E}_{off}(C)$ represents the energy consumption of issuing a power-off instruction for component C and $\mathbb{E}_{on}(C)$ represents the energy consumption of issuing a power-on instruction for component C; $\mathbb{P}_{leak}(C)$ represents the leakage power consumption of component C in a cycle; $\mathbb{P}_{rleak}(C)$ represents the leakage power consumption of component C in a reduced level in a cycle[1] ; and ITVL^{idle} is the length of the idle interval. Accordingly, we have a break-even length of idle intervals for each component C, called $\mathsf{BE\text{-}ITVL}_C^{idle}$, that sustains the above inequality and it is given by

$$\mathsf{BE\text{-}ITVL}_C^{idle} = \frac{\mathbb{E}_{off}(C) + \mathbb{E}_{on}(C)}{\mathbb{P}_{leak}(C) - \mathbb{P}_{rleak}(C)} \quad .$$

Hence, the compiler must be aware that power-gating control of a certain component C is employed only when there exists a continuous idle interval whose length is greater than $\mathsf{BE\text{-}ITVL}_C^{idle}$ on the component. Moreover, the latency associated with powering a component on should also be considered.

The component activity information gathered and the cost model for deciding if the power-gating instructions should be employed now 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. We call this scheme *Basic_Blk_Sched*. 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 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. Finally, 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. *AVG_Path_Sched* returns the average length of two branchings instead of the minimum.

4. SINK-N-HOIST ANALYSIS

The main idea of the *Sink-N-Hoist Analysis* is to abate the problem of too many instructions being added with code motion techniques. The approach attempts to merge several power-gating instructions into one compound instruction by 'sinking' power-off instructions and 'hoisting' power-on instructions, i.e., postponing the issue of power-off instructions late and advancing the issue of power-on instructions early. This will result in profits mainly for code size, but also in performance and energy via grouping effects. For instance, a power-off instruction can be postponed for some cycles to be merged with other adjacent power-off instructions. Nevertheless, there should be a limitation on the number of cycles to be sank or hoisted since sinking or hoisting a power-gating instruction will cause more leakage dissipation. A cost model is given below to determine the feasibility. For a component C, we have

$$\mathbb{E}_{off}(C) + \mathbb{P}_{rleak}(C) \cdot \mathsf{SINK\text{-}SLK} > \mathbb{P}_{leak}(C) \cdot \mathsf{SINK\text{-}SLK} + \mathbb{E}_{fet\text{-}dec\text{-}off}(C)/N + \mathbb{E}_{exe\text{-}off}(C),$$

where $\mathsf{SINK\text{-}SLK}$ is the number of cycles a power-off statement (or instruction[2]) is sank, i.e., the power-off statement

[1]An effective way to reduce leakage power is to power off a component with power-gating mechanisms, which shut down the component and make $\mathbb{P}_{rleak}(C)$ be zero, while others may increase threshold voltage to cause smaller $\mathbb{P}_{rleak}(C)$. We model this factor as a variable rather than treat it as zero.

[2]In the following context, 'statement' and 'instruction' are interchangeably used since a statement in the assembly level means an instruction.

Sinkable-N-Hoist Algorithm

Input: INACTIVITY(b) for each block b and positions for power-gating instructions.
Output: Appropriate positions for power-gating instructions.

1. Perform the *Sinkable Analysis* and *Hoistable Analysis*. (Equation (3)–(6))
2. Perform the *Grouping-Off Analysis* and *Grouping-On Analysis*. (Equation (7)–(10))
3. Perform the *Power-Gating Instruction Placement*.

Figure 5: Sink-N-Hoist algorithm.

is delayed for SINK-SLK cycles, $\mathbb{E}_{fet\text{-}dec\text{-}off}(C)$ returns the energy consumption of fetching and decoding a power-off instruction, $\mathbb{E}_{exe\text{-}off}(C)$ returns the energy consumption of executing a power-off instruction, and N is the number of power-gated components. Note that the sum of $\mathbb{E}_{fet\text{-}dec\text{-}off}(C)$ and $\mathbb{E}_{exe\text{-}off}(C)$ is equal to $\mathbb{E}_{off}(C)$. The right-hand side of the inequality represents the energy consumed when the power-off statement is delayed for SINK-SLK cycles and merged with other $(N-1)$ power-off statements while the left-hand side stands for the energy consumed when the power-off statement is called right after the end of the active interval. In consequence, we have a maximum sinkable slack for each component C, called MAX-SINK-SLK$_C$, that sustains the above inequality and it is given by

$$\text{MAX-SINK-SLK}_C = \frac{(N-1)\cdot \mathbb{E}_{fet\text{-}dec\text{-}off}(C)}{N\cdot(\mathbb{P}_{leak}(C)-\mathbb{P}_{rleak}(C))} .$$

Similarly, we have a maximum hoistable slack for each component and it is given by

$$\text{MAX-HOIST-SLK}_C = \frac{(N-1)\cdot \mathbb{E}_{fet\text{-}dec\text{-}on}(C)}{N\cdot(\mathbb{P}_{leak}(C)-\mathbb{P}_{rleak}(C))} .$$

With such cost constraint as the basis, we now present a set of data-flow equations to collect the information for code motion of power-gating instructions. Figure 5 shows the algorithm of the *Sink-N-Hoist Analysis*. The whole set of equations used are presented in Figure 6. The *Sink-N-Hoist Analysis* mainly consists of three phases: 1) the *Sinkable* and *Hoistable Analysis*, which compute the information of possible positions for each power-gating instruction, 2) the *Grouping-Off* and *Grouping-On Analysis*, which group together the power-gating instructions that can be merged, and 3) the determination of the appropriate positions for power-gating instructions. The details are discussed as follows.

4.1 Sinkable and Grouping-Off Analysis

The predicates for collecting SINKABLE and GROUP-OFF information are given as follows: The SINKABLE gives the data flow equation to collect how far the turn-off instructions of component activities can be sank. In addition, GROUP-OFF gives the data flow equation to partition the turn-off instructions into groups, and we can then use this information to group them by selecting emitting instructions.

- SINKABLE$_{loc}(b)$ is a set of power-off statements which occur within block b and can be safely moved to the end of the block. Each statement is associated with a number, named SINK-SLK^b_C, which keeps an integer value of slack time for component C for indicating how many cycles the power-off statement can be sank at the current position. The initial value of SINK-SLK^b_C is set as MAX-SINK-SLK$_C$.
- SINKABLE$_{blk}(b)$ is a set of power-off statements which cannot be safely moved from the start to the end of bock b, i.e., a set of power-off statements whose value of the associated SINK-SLK^b_C is zero.
- SINKABLE$_{in}(b)$ is a set of power-off statements which can be safely moved to the beginning of block b. The SINKABLE$_{in}(b)$ is computed as follows:
$$\text{SINKABLE}_{in}(b) = \bigcap_{p\in Pred(b)} \text{SINKABLE}_{out}(p).$$
Meanwhile, the value of SINK-SLK^b_C would be the minimum one among the predecessors of b if the value of SINK-SLK^p_C is inconsistent with each other, where p is a predecessor of block b. It means that the sinkable-slack from one predecessor would be shrunk if other predecessors have a smaller sinkable-slack. This is for the consideration that a power-off statement should not be sank far away to the position that may cause a reverse effect. Moreover, the value of each SINK-SLK^b_C is decreased by one to be in accordance with the definition. In brief, the value of SINK-SLK^b_C is given by
$$\text{SINK-SLK}^b_C = \mathbb{MIN}_{p\in Pred(b)}(\text{SINK-SLK}^p_C) - 1,$$
where $\mathbb{MIN}$ is a function that returns the minimum value among its parameters.
- SINKABLE$_{out}(b)$ is a set of power-off statements which can be safely moved to the end of block b. The SINKABLE$_{out}(b)$ is computed as follows:
$$\text{SINKABLE}_{out}(b) = \text{SINKABLE}_{loc}(b) \cup (\text{SINKABLE}_{in}(b) - \text{SINKABLE}_{blk}(b)).$$
Meanwhile, the value of SINK-SLK^b_C is given from the value of the associated SINK-SLK^b_C in SINKABLE$_{loc}(b)$ if there exists a power-off-C statement in SINKABLE$_{loc}(b)$; otherwise, it is given from the one in SINKABLE$_{in}(b)$.

We now gives the data flow equation for GROUP-OFF. It will partition the turn-off instructions into groups, and we can then use this information to group them by selecting emitting instructions.

- GROUP-OFF$_{loc}(b)$ is a set with at most one element, i.e., a singleton or an empty set, in which the element (if it exists) is an integer representing a group number and never appears in other sets of GROUP-OFF$_{loc}$. The block b belongs to the group it numbered and is the beginning block of a set of successive blocks if the GROUP-OFF$_{loc}(b)$ is not empty. The GROUP-OFF$_{loc}(b)$ set is not empty only when
$$\text{SINKABLE}_{out}(b) \neq \emptyset \text{ and } \bigcup_{p\in Pred(b)} \text{SINKABLE}_{out}(p) = \emptyset.$$

$$\mathsf{COMPONENT}_{in}(b) = \bigcup_{p \in Pred(b)} \mathsf{COMPONENT}_{out}(p) \tag{1}$$

$$\mathsf{COMPONENT}_{out}(b) = \mathsf{COMPONENT}_{loc}(b) \cup (\mathsf{COMPONENT}_{in}(b) - \mathsf{COMPONENT}_{blk}(b)) \tag{2}$$

$$\mathsf{SINKABLE}_{in}(b) = \bigcap_{p \in Pred(b)} \mathsf{SINKABLE}_{out}(p) \tag{3}$$

$$\mathsf{SINKABLE}_{out}(b) = \mathsf{SINKABLE}_{loc}(b) \cup (\mathsf{SINKABLE}_{in}(b) - \mathsf{SINKABLE}_{blk}(b)) \tag{4}$$

$$\mathsf{HOISTABLE}_{out}(b) = \bigcap_{s \in Succ(b)} \mathsf{HOISTABLE}_{in}(s) \tag{5}$$

$$\mathsf{HOISTABLE}_{in}(b) = \mathsf{HOISTABLE}_{loc}(b) \cup (\mathsf{HOISTABLE}_{out}(b) - \mathsf{HOISTABLE}_{blk}(b)) \tag{6}$$

$$\mathsf{GROUP\text{-}OFF}_{in}(b) = \begin{cases} \{\mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}OFF}_{out}(p)))\} \\ \emptyset, \quad \text{if } \mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}OFF}_{out}(p))) = \infty \end{cases} \tag{7}$$

$$\mathsf{GROUP\text{-}OFF}_{out}(b) = \mathsf{GROUP\text{-}OFF}_{loc}(b) \cup (\mathsf{GROUP\text{-}OFF}_{in}(b) - \mathsf{GROUP\text{-}OFF}_{blk}(b)) \tag{8}$$

$$\mathsf{GROUP\text{-}ON}_{in}(b) = \begin{cases} \{\mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}ON}_{out}(p)))\} \\ \emptyset, \quad \text{if } \mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}ON}_{out}(p))) = \infty \end{cases} \tag{9}$$

$$\mathsf{GROUP\text{-}ON}_{out}(b) = \mathsf{GROUP\text{-}ON}_{loc}(b) \cup (\mathsf{GROUP\text{-}ON}_{in}(b) - \mathsf{GROUP\text{-}ON}_{blk}(b)) \tag{10}$$

Figure 6: Component-Activity and Sink-N-Hoist data-flow equations.

A simple way to ensure that all the numbers in the sets of $\mathsf{GROUP\text{-}OFF}_{loc}$ of all blocks are unique is using a integer counter to assign each element with the value of the counter. Once an element is assigned, the counter increases.

- $\mathsf{GROUP\text{-}OFF}_{blk}(b)$ is a universal set of integers, namely Ω, or an empty set. The set is not empty (set to be a set with an Ω value) only when

  $$\mathsf{SINKABLE}_{out}(b) = \emptyset \text{ and } \bigcup_{s \in Succ(b)} \mathsf{SINKABLE}_{out}(s) \neq \emptyset.$$

  In all other cases, it will be an empty set.

- $\mathsf{GROUP\text{-}OFF}_{in}(b)$ is an integer singleton, a group number, which can be assigned to the start of block b or an empty set. The $\mathsf{GROUP\text{-}OFF}_{in}(b)$ is computed by

  $$\mathsf{GROUP\text{-}OFF}_{in}(b) = \begin{cases} \{\mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}OFF}_{out}(p)))\} \\ \emptyset, \quad \text{if } \mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}OFF}_{out}(p))) = \infty \end{cases}$$

  where Φ returns the value of the element of its parameter and returns infinity if the parameter is an empty set. In addition, all the $\mathsf{GROUP\text{-}OFF}_{out}$ set in the same group of its predecessors can be replaced by the $\mathsf{GROUP\text{-}OFF}_{in}(b)$ if the $\mathsf{GROUP\text{-}OFF}_{out}$ set of the predecessor of b is not empty. This will allow opportunity for further grouping effects.

- $\mathsf{GROUP\text{-}OFF}_{out}(b)$ is an integer singleton, a group number, which can be assigned to the end of block b or an empty set. In fact, the element in $\mathsf{GROUP\text{-}OFF}_{out}(b)$ gives the group number that block b belongs to. The $\mathsf{GROUP\text{-}OFF}_{out}(b)$ is computed by

  $$\mathsf{GROUP\text{-}OFF}_{out}(b) = \mathsf{GROUP\text{-}OFF}_{loc}(b) \cup (\mathsf{GROUP\text{-}OFF}_{in}(b) - \mathsf{GROUP\text{-}OFF}_{blk}(b)).$$

In the following, we give a running example to illustrate how the analysis works. Suppose that two components, namely **A** and **B**, are considered for analyses. Given a control-flow graph as shown in Figure 7(a), where each block in the graph contains only a statement, we can determine where power-gating statements should be located by performing step (i), (ii), (iii), and (v) in Figure 3. It includes the *Component-Activity Data-Flow Analysis* and power-gating instruction scheduling. In this example, it is found that component **A** and **B** should be powered off at B_{m+2} and B_{n+2} and at B_{m+5} and B_{n+5}, respectively. The shadowed blocks represent that components are in use (the left half is for component **A** and the right half is for component **B**). To reduce the amount of power-gating instructions issued, we then apply the *Sinkable Analysis*. By the definition of the $\mathsf{SINKABLE}_{loc}(b)$, a set of power-off statements which occur within block b, we have $\mathsf{SINKABLE}_{loc}(B_{m+2}) = \{\text{PowerOff } \mathbf{A}(4)\}$, $\mathsf{SINKABLE}_{loc}(B_{m+5}) = \{\text{PowerOff } \mathbf{B}(2)\}$, $\mathsf{SINKABLE}_{loc}(B_{n+2}) = \{\text{PowerOff } \mathbf{A}(4)\}$, and $\mathsf{SINKABLE}_{loc}(B_{n+5}) = \{\text{PowerOff } \mathbf{B}(2)\}$, where the numbers in parentheses indicate the value of the associated $\mathsf{SINK\text{-}SLK}_C$ (in fact, the values come from the $\mathsf{MAX\text{-}SINK\text{-}SLK}_A$ and $\mathsf{MAX\text{-}SINK\text{-}SLK}_B$), and the $\mathsf{SINKABLE}_{loc}$ of the other blocks are empty sets. To simplify the representation, the word 'PowerOff' is removed and the value of the associated $\mathsf{SINK\text{-}SLK}_C$ is superscripted, e.g., $\mathsf{SINKABLE}_{loc}(B_{m+2}) = \{\mathbf{A}^4\}$. Table 1 shows the computation results of the $\mathsf{SINKABLE}_{blk}(b)$, $\mathsf{SINKABLE}_{in}(b)$, and $\mathsf{SINKABLE}_{out}(b)$ for each block. Actually, the $\mathsf{SINKABLE}_{out}(b)$ indicates the set of power-off statements that can be issued at block b without energy penalties if the statements could be merged with other statements. In other

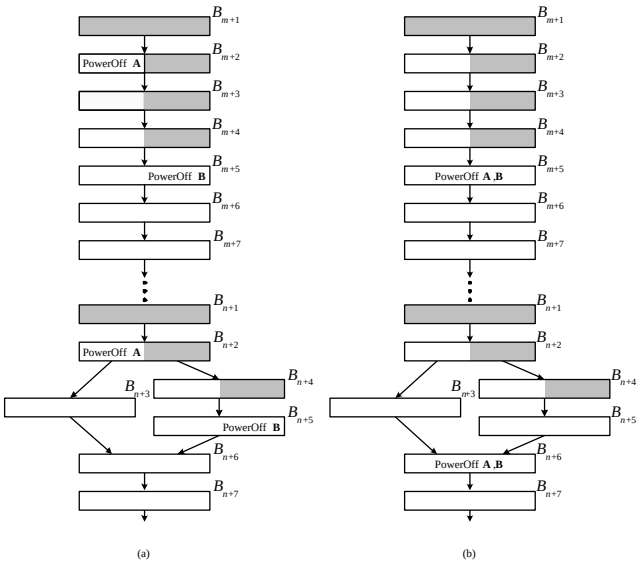

Figure 7: An example of sinking power-off statements. (The shadowed blocks represent that components are in use.)

words, the power-off statements of component **A** can be issued at B_{m+2} to B_{m+5} and B_{n+2} to B_{n+6}. Next, we compute the GROUP-OFF$_{loc}(b)$, GROUP-OFF$_{blk}(b)$, GROUP-OFF$_{in}(b)$, and GROUP-OFF$_{out}(b)$ for each block to group the blocks in which the power-off statements of the component that appear in this group should be issued only and exactly once. Table 2 gives the grouping results: B_{m+2} to B_{m+6} belong to group number one and B_{n+2} to B_{n+6} belong to group number two.

4.2 Hoistable and Grouping-On Analysis

The *Hoistable Analysis* and *Grouping-On Analysis* are similar to the *Sinkable Analysis* and *Grouping-Off Analysis*, but the *Hoistable Analysis* is a backward data-flow analysis. Similarly, we can define a set of predicates for collecting HOISTABLE and GROUP-ON information as follows.

- HOISTABLE$_{loc}(b)$ is a set of power-on statements which occur within block b and can be safely moved to the start of the block. Each statement is associated with a number, named HOIST-SLK^b_C, which keeps an integer value of slack time for component C for indicating how many cycles the power-on statement can be hoisted at the current position. The initial value of HOIST-SLK^b_C is set as MAX-HOIST-SLK$_C$.
- HOISTABLE$_{blk}(b)$ is a set of power-on statements which cannot be safely moved from the end to the start of bock b, i.e., the set of power-on statements whose value of the associated HOIST-SLK^b_C is zero.
- HOISTABLE$_{out}(b)$ is a set of power-on statements which can be safely moved to the end of block b. The HOISTABLE$_{out}(b)$ is computed as follows:

$$\text{HOISTABLE}_{out}(b) = \bigcap_{s \in Succ(b)} \text{HOISTABLE}_{in}(s).$$

Meanwhile, the value of HOIST-SLK^b_C would be the minimum one among the successors of b if the value of HOIST-SLK^s_C is inconsistent with each other, where s is a successor of block b. It means that the hoistable-slack from one successor would be shrunk if other successors have a smaller hoistable-slack. This is for the consideration that a power-on statement should not be hoisted far away to the position that may cause a reverse effect. Moreover, the value of each HOIST-SLK^b_C is decreased by one to be in accordance with the definition. In brief, the value of HOIST-SLK^b_C is given by

$$\text{HOIST-SLK}^b_C = \mathbb{MIN}_{s \in Succ(b)}(\text{HOIST-SLK}^s_C) - 1.$$

- HOISTABLE$_{in}(b)$ is a set of power-on statements which can be safely moved to the start of block b. The HOISTABLE$_{in}(b)$ is computed as follows:

$$\text{HOISTABLE}_{in}(b) = \text{HOISTABLE}_{loc}(b) \cup (\text{HOISTABLE}_{out}(b) - \text{HOISTABLE}_{blk}(b)).$$

Meanwhile, the value of HOIST-SLK^b_C is given from the value of the associated HOIST-SLK^b_C in HOISTABLE$_{loc}(b)$ if there exists a power-on-C statement in HOISTABLE$_{loc}(b)$; otherwise, it is given from the one in HOISTABLE$_{out}(b)$.

Block	SINKABLE$_{blk}(b)$	SINKABLE$_{in}(b)$	SINKABLE$_{out}(b)$
B_{m+1}			
B_{m+2}			$\{\mathbf{A}^4\}$
B_{m+3}		$\{\mathbf{A}^3\}$	$\{\mathbf{A}^3\}$
B_{m+4}		$\{\mathbf{A}^2\}$	$\{\mathbf{A}^2\}$
B_{m+5}		$\{\mathbf{A}^1\}$	$\{\mathbf{A}^1, \mathbf{B}^2\}$
B_{m+6}	$\{\mathbf{A}\}$	$\{\mathbf{A}^0, \mathbf{B}^1\}$	$\{\mathbf{B}^1\}$
B_{m+7}	$\{\mathbf{B}\}$	$\{\mathbf{B}^0\}$	
...			
B_{n+1}			
B_{n+2}			$\{\mathbf{A}^4\}$
B_{n+3}		$\{\mathbf{A}^3\}$	$\{\mathbf{A}^3\}$
B_{n+4}		$\{\mathbf{A}^3\}$	$\{\mathbf{A}^3\}$
B_{n+5}		$\{\mathbf{A}^2\}$	$\{\mathbf{A}^2, \mathbf{B}^2\}$
B_{n+6}		$\{\mathbf{A}^1, \mathbf{B}^1\}$	$\{\mathbf{A}^1, \mathbf{B}^1\}$
B_{n+7}	$\{\mathbf{A},\mathbf{B}\}$	$\{\mathbf{A}^0, \mathbf{B}^0\}$	

Table 1: SINKABLE predicates. (The superscript represents the value of the associated SINK-SLK^b_C.)

Block	GROUP-OFF$_{blk}(b)$	GROUP-OFF$_{in}(b)$	GROUP-OFF$_{out}(b)$
B_{m+1}			
B_{m+2}			{1}
B_{m+3}		{1}	{1}
B_{m+4}		{1}	{1}
B_{m+5}		{1}	{1}
B_{m+6}		{1}	{1}
B_{m+7}	Ω	{1}	
...			
B_{n+1}			
B_{n+2}			{2}
B_{n+3}		{2}	{2}
B_{n+4}		{2}	{2}
B_{n+5}		{2}	{2}
B_{n+6}		{2}	{2}
B_{n+7}	Ω	{2}	

Table 2: GROUP-OFF predicates.

- $\mathsf{GROUP\text{-}ON}_{loc}(b)$ is a set with at most one element, i.e., a singleton or an empty set, in which the element (if it exists) is an integer representing a group number and never appears in other sets of $\mathsf{GROUP\text{-}ON}_{loc}$. The block b belongs to the group it numbered and is the beginning block of a set of successive blocks if the $\mathsf{GROUP\text{-}ON}_{loc}(b)$ is not empty. The $\mathsf{GROUP\text{-}ON}_{loc}(b)$ set is not empty only when

  $$\mathsf{HOISTABLE}_{in}(b) \neq \emptyset \text{ and } \bigcup_{p \in Pred(b)} \mathsf{HOISTABLE}_{in}(p) = \emptyset.$$

  A simple way to ensure that all the numbers in the sets of $\mathsf{GROUP\text{-}ON}_{loc}$ of all blocks are unique is using a integer counter to assign each element with the value of the counter. Once an element is assigned, the counter increases.

- $\mathsf{GROUP\text{-}ON}_{blk}(b)$ is a universal set of integers, namely Ω, or an empty set. The block b is one, or the only one, of the end blocks of a set of successive blocks if the $\mathsf{GROUP\text{-}ON}_{blk}(b)$ is not empty. The set is not empty only when

  $$\mathsf{HOISTABLE}_{in}(b) = \emptyset \text{ and } \bigcup_{s \in Succ(b)} \mathsf{HOISTABLE}_{in}(s) \neq \emptyset.$$

- $\mathsf{GROUP\text{-}ON}_{in}(b)$ is an integer singleton, a group number, which can be assigned to the start of block b or an empty set. The $\mathsf{GROUP\text{-}ON}_{in}(b)$ is computed by

  $$\mathsf{GROUP\text{-}ON}_{in}(b) = \begin{cases} \{\mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}ON}_{out}(p)))\} \\ \emptyset, \quad \text{if } \mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{-}ON}_{out}(p))) = \infty \end{cases}$$

  where Φ returns the value of the element of its parameter and returns infinity if the parameter is an empty set. In addition, we can also replace all the $\mathsf{GROUP\text{-}ON}_{out}$ set of its predecessors by the $\mathsf{GROUP\text{-}ON}_{in}(b)$, i.e. if $\mathsf{GROUP\text{-}ON}_{out}$ set of the predecessor of b is not empty. Note that this gives further opportunity for grouping effects.

- $\mathsf{GROUP\text{-}ON}_{out}(b)$ is an integer singleton, a group number, which can be assigned to the end of block b or an empty set. In fact, the element in $\mathsf{GROUP\text{-}ON}_{out}(b)$ gives the group number that block b belongs to. The $\mathsf{GROUP\text{-}ON}_{out}(b)$ is computed by

  $$\mathsf{GROUP\text{-}ON}_{out}(b) = \mathsf{GROUP\text{-}ON}_{loc}(b) \cup (\mathsf{GROUP\text{-}ON}_{in}(b) - \mathsf{GROUP\text{-}ON}_{blk}(b)).$$

4.3 Power-Gating Instruction Placement

With the $\mathsf{SINKABLE}_{out}$, $\mathsf{HOISTABLE}_{in}$, and $\mathsf{GROUP\text{-}OFF/ON}_{out}$ collected in Section 4.1 and 4.2, we then use these information to determine how to place power-gating instructions, i.e., whether power-gating instructions should be combined together or issued separately. Figure 8 gives a brief algorithm of the power-gating instructions placement. The basic idea of the algorithm is to place power-gating instructions in a group-by-group manner. It first determines all possible policies for issuing power-gating instructions — a legal policy is that all power-gating instructions should be issued at the block b in which $\mathsf{SINKABLE}_{out}(b)$ or $\mathsf{HOISTABLE}_{in}(b)$ is not empty and each type of power-gating instructions appearing within a group must be issued only and exactly once. Next, it uses an energy-cost model, which describes the energy, such as the leakage energy, the energy of issuing power-off instructions, etc., to determine which policy has the best benefit in energy consumption aspects.

Power-Gating Instruction Placement Algorithm

Input: $\mathsf{SINKABLE}_{out}$, $\mathsf{GROUP\text{-}OFF}_{out}$, $\mathsf{HOISTABLE}_{in}$, and $\mathsf{GROUP\text{-}ON}_{out}$ information for each block.
Output: Appropriate positions for power-gating instructions.

```
placement() {
  for each group
    /* determine all possible policies for issuing
    power-gating instructions */
    policy_list = get_possible_policies(
       SINKABLE_out,GROUP-OFF_out,
       HOISTABLE_in,,GROUP-ON_out);

    /* determine which policy consumes least power*/
    best_policy = get_best_policy(policy_list);

    /* annotate the positions of power-gating
    instructions */
    make_annotation(best_policy);
  end
}
```

Figure 8: Power-Gating Instruction Placements.

In the following, we elaborate the idea by continuing the example in Section 4.1. With the information of $\mathsf{SINKABLE}_{out}$ and $\mathsf{GROUP\text{-}OFF}_{out}$, an energy-cost model is established and evaluated for each case of issuing-power-off-instruction policies under the guideline that power-off instructions must be issued at the block in which the $\mathsf{SINKABLE}_{out}$ is not empty and each type of power-gating instructions appearing within a group must be issued only and exactly once, e.g., the policy could be '*powering off* **A** *at* B_{m+2} *and powering off* **B** *at* B_{m+5}' or '*powering off* **A** *and* **B** *at* B_{m+2}' in group number one. The final decision of which policy to be taken depends on the energy cost evaluated by the model; certainly, the one with the minimum cost is chosen as it should be for low-power consideration. Finally, power-off instructions are inserted at appropriated points as shown in Figure 7(b): the power-off statements within each group are merged.

5. EXPERIMENTAL RESULTS

We use a DEC-Alpha-compatible architecture with power-gating control and instruction sets described in Figure 2 as the target architecture for our experiments. The proposed leakage-power reduction framework is incorporated into the compiler tool with SUIF [16] and MachSUIF [15], and evaluated by the Wattch simulator with .10μm process parameter and 1.9 V supply voltage [2]. Figure 9 illustrates the phases in the compilation and simulation framework. We incorporate the low-power optimization phase following MachSUIF phase. As Wattch does not model leakage at the component level per se, we assume that leakage power contributes 10% of total power consumption. Furthermore, we assume that wake-up operations of power-gating control take 20-cycle latency, although 7.5 cycles are introduced in [3], and it takes

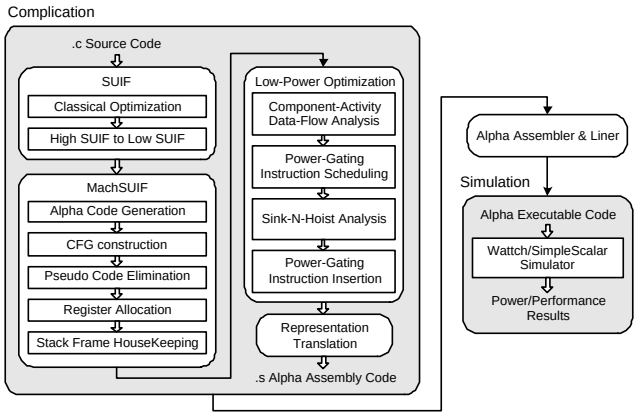

Figure 9: Compilation and simulation framework.

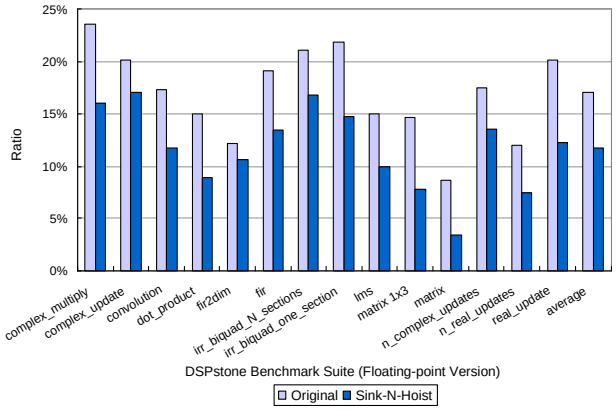

Figure 10: Ratio of power-gating instructions over total instructions in program codes.

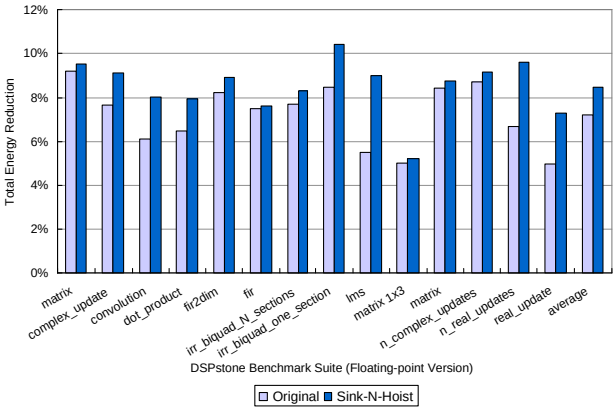

Figure 11: Total energy reduction.

two times and ten times of leakage energy per cycle to power off and power on a component, respectively. The energy consumption of fetching and decoding a power-gating instruction is assumed to be two times of leakage power. Also the baseline data is provided by Wattch's *cc3* clock-gating power estimation, which gates the clocks of those unused resources in multiported hardware to reduce the dynamic power; however, leakage power is still leaked. The benchmarks used in our experiment are from the floating-point version of DSP-stone benchmark suite [26].

Three versions are compared. The base version is the one without power gating mechanism. The original version is the one from a previous work [23,24] that only performs the step (i), (ii) and (iii) in Figure 3. The *Sink-N-Hoist Analysis* scheme is the one proposed in this work to perform all phases in Figure 3. In addition, three policies for power-gating instruction scheduling were proposed in step (iii) of Figure 3 to deal with conditional branches in programs. Without loss of generality, we use the Min_Path_Sched policy to schedule power-gating instructions in this experiment. Figure 10–12 give the compilation and simulation results of two approaches: the original one and the Sink-N-Hoist one when the integer ALU, floating-point adder, and floating-point multiplier are considered for power gating, and the comparison baseline in these figures is the one without power-gating controls. Figure 10 presents the ratio of power-gating instructions over total instructions in program codes. It shows that the *Sink-N-Hoist* approach has about 31.2% of improvement (from 17.0% to 11.8%) in the reduction of the amount of power-gating instructions generated comparing to the one without *Sink-N-Hoist* framework. Moreover, our scheme also further reduces the total energy consumption compared to the one without *Sink-N-Hoist* framework. This is due to that the effect of a block version of power-gating instructions gives better power and performance effects than the pointwise version of power-gating instructions. Figure 11 shows our scheme gives average 18.0% reduction (from 7.2% to 8.5% of total power) comparing to the base method. Note that the average reduction of total energy is less than 10%, but we should recall that only three types of functional units (the integer ALU, floating-point adder, and floating-point multiplier) are under power-gating controls in this experiment. In fact, the base method already achieved average 70.4% and 72.6% energy reduction for the floating-point adder and floating-point multiplier in combined dynamic and leakage power, respectively [23,24]. Figure 11 also shows our scheme holding edges over the original scheme in energy reduction. This is due to the effect of a block version of power-gating instructions gives better power effects than the pointwise version as illustrated in our cost model.

Finally, Figure 12 shows the detailed information of the performance impact caused by power-gating mechanisms, and it says that the performance degradation is reduced about 2.9% (from 2.01% to 1.95%) over the original method. Our method holds a small edge over the one without *Sink-N-Hoist* framework due to the reduction of the amount of power-gating instructions. Note that the performance penality is not as bad as the amount of instructions added due to most instructions were added outside the loop kernel. Nevertheless, the reduction of the amount of power-gating instructions still gives performance edges.

6. CONCLUSION

In this paper, we presented a *Sink-N-Hoist Analysis* for merging several power-gating instructions. In summary, our experiment shows that the *Sink-N-Hoist Analysis* framework results in benefits for code sizes as well as energy consump-

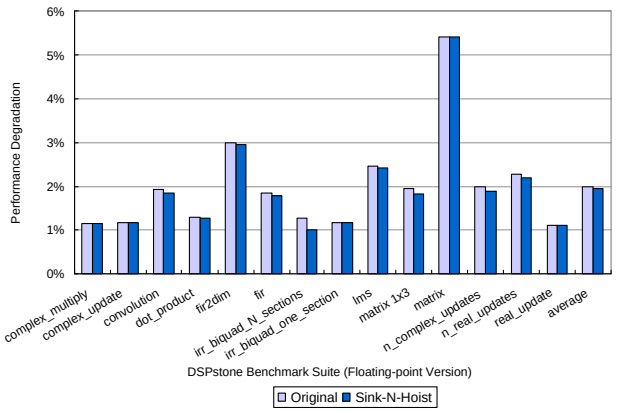

Figure 12: Performance degradation.

tion and performance. As the compiler phase is done one phase after another, our framework gives a sound theoretical foundation capable to work with other phases such as adding more slackness for low power with code motions. Finally, we are in the process of incorporating more components, such as crypto modules, into our architecture and simulator. We expect the effects of our scheme will be even more important as more extensible modules are equipped with power-gating control in this platform.

7. ACKNOWLEDGEMENTS

This work was supported in part by Ministry of Economic Affairs under grant no. 93-EC-17-A-03-S1-0002 and 94-EC-17-A-01-S1-034, by National Science Council under grant no. 93-2220-E-007-019, 93-2220-E-007-020, and 94-2752-E-007-004-PAE in Taiwan.

8. REFERENCES

[1] 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 Systems*, 8(3):317–326, June 2000.

[2] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)*, pages 83–94, June 2000.

[3] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'00)*, pages 191–201, December 2000.

[4] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[5] Jui-Ming Chang and Massoud Pedram. Register allocation and binding for low power. In *Proceedings of the Design Automaton Conference*, pages 29–35, June 1995.

[6] Steven Dropsho, Volkan Kursun, David H. Albonesi, Sandhya Dwarkadas, and Eby G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO'02)*, pages 321–332, November 2002.

[7] Ricardo E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[8] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Proceedings of the IEEE Symposium on Low Power Electronics*, pages 8–11, October 1994.

[9] Henry Ip, James Low, Peter Y. K. Cheung, George A. Constantinides, Wayne Luk, Shay P. Seng, and Paul Metzgen. Strassen's matrix multiplication for customisable processors. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT'02)*, pages 453–456, December 2002.

[10] J. T. Kao and A. P. Chandrakasan. Dual-threshold voltage techniques for low-power digital circuits. *IEEE Journal of Solid-State Circuits*, 35(7):1009–1018, 2000.

[11] Chingren Lee, Jenq Kuen Lee, Ting-Ting Hwang, and Shi-Chun Tsai. Compiler optimizations on VLIW instruction scheduling for low power. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):252–268, 2003.

[12] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power analysis and minimization techniques for embedded DSP software. *IEEE Transactions on Very Large Scale Integration Systems*, 5(1):123–133, March 1997.

[13] Siddharth Rele, Santosh Pande, Soner Onder, and Rajiv Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, pages 261–275, April 2002.

[14] K. Roy and S. C. Prasad. SYCLOP: Synthesis of CMOS logic for low power applications. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'92)*, pages 464–467, October 1992.

[15] Michael D. Smith. *The SUIF Machine Library*. Division of of Engineering and Applied Science, Harvard University, 1998.

[16] Stanford Compiler Group. *The SUIF Library*. Stanford Compiler Group, Stanford University, 1995.

[17] Ching-Long Su and Alvin M. Despain. Cache designs for energy efficiency. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 306–315, January 1995.

[18] Scott Thompson, Paul Packan, and Mark Bohr. Mos scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, 1998.

[19] V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez. Dynamic power management for microprocessors: A case study. In *Proceedings of the International Conference on VLSI Design*, pages 185–192, January 1997.

[20] Vivek Tiwari, Deo Singh, Suresh Rajgopal, Gaurav Mehta, Rakesh Patel, and Franklin Baez. Reducing power in high-performance microprocessors. In *Proceedings of the Design Automaton Conference (DAC'98)*, pages 732–737, June 1998.

[21] Hiroshi Tsutsui, Takahiko Masuzaki, Tomonori Izumi, Takao Onoye, and Yukihiro Nakamura. High speed JPEG2000 encoder by configurable processor. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS'02)*, pages 45–50, December 2002.

[22] Hongbo Yang, R. Govindarajan, Guang R. Gao, George Cai, and Ziang Hu. Exploiting schedule slacks for rate-optimal power-minimum software pipelining. In *Proceedings of the 3rd workshop on Compilers and Operating Systems for Low Power (COLP'02)*, September 2002.

[23] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. Compilers for leakage power reduction. *Accepted, ACM Transactions on Design Automation of Electronic Systems*.

[24] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. Compiler analysis and supports for leakage power reduction on microprocessors. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, pages 63–73, July 2002. Lecture Notes in Computer Science, Vol. 2481, Springer Verlag.

[25] W. Zhang, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and V. De. Compiler support for reducing leakage energy consumption. In *Proceedings of the 6th Design Automation and Test in Europe Conference (DATE'03)*, pages 1146–1147, March 2003.

[26] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, pages 715–720, October 1994.