Compilation for Compact Power-Gating Controls

YI-PING YOU, CHUNG-WEN HUANG, and JENQ KUEN LEE
National Tsing Hua University

Power leakage constitutes an increasing fraction of the total power consumption in modern semiconductor technologies due to the continuing size reductions and increasing speeds of transistors. Recent studies have attempted to reduce leakage power using integrated architecture and compiler power-gating mechanisms. This approach involves compilers inserting instructions into programs to shut down and wake up components, as appropriate. While early studies showed this approach to be effective, there are concerns about the large amount of power-control instructions being added to programs due to the increasing amount of components equipped with power-gating controls in SoC design platforms. In this article we present a *sink-n-hoist* framework for a compiler to generate balanced scheduling of power-gating instructions. Our solution attempts to merge several power-gating instructions into a single compound instruction, thereby reducing the amount of power-gating instructions issued. We performed experiments by incorporating our compiler analysis and scheduling policies into SUIF compiler tools and by simulating the energy consumption using Wattch toolkits. The experimental results demonstrate that our mechanisms are effective in reducing the amount of power-gating instructions while further reducing leakage power compared to previous methods.

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

General Terms: Algorithms, Experimentation, Languages

Additional Key Words and Phrases: Compilers for low power, data-flow analysis, leakage-power reduction, balanced scheduling, power-gating mechanisms

ACM Reference Format:

You, Y.-P, Huang, C.-W., and Lee, J. K. 2007. Compilation for compact power-gating controls. ACM Trans. Des. Automat. Electron. Syst. 12, 4, Article 51 (September 2007), 26 pages. DOI = 10.1145/1278349.1278364 http://doi.acm.org/10.1145/1278349.1278364

This work was supported in part by the National Science Council Grants NSC 95-2220-E-007-001 and NSC 95-2220-E-007-002, the Ministry of Economic Affairs Grants 95-EC-17-A-01-S1-034 and 96-EC-17-A-01-S1-034, and ITRI under an ITRI/NTHU research grant.

Authors' addresses: Y.-P. You, C.-W. Huang, J. K. Lee, (corresponding author), Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan; email: {ypyou, cwhuang}@pllab.cs.nthu.edu.tw; jklee@cs.nthu.edu.tw.

1. INTRODUCTION

Minimizing power dissipation can be considered at algorithmic, architectural, logic, and circuit levels [Chandrakasan et al. 1992]. Numerous studies in the literature on low-power design have proposed various techniques for synthesizing designs with reduced transitional activities. Recently, the prospect of combining architecture design and software arrangement at the instruction level has been addressed to help reduce power consumption [Bellas et al. 2000; Chang and Pedram 1995; Horowitz et al. 1994; Lee et al. 2003; 1997; Su and Despain 1995; Tiwari et al. 1998, 1997] For example, several types of software rearrangement have been used to reduce the dynamic power, such as utilizing the value locality of registers [Chang and Pedram 1995], swapping operands for Booth multipliers [Lee et al. 1997], scheduling VLIW instructions to reduce the power consumption on the instruction bus [Lee et al. 2003], gating the clock to reduce workloads [Horowitz et al. 1994; Tiwari et al. 1998, 1997], utilizing cache subbanking mechanisms [Su and Despain 1995], and an instruction cache for loops [Bellas et al. 2000].

Leakage power is coming to represent a greater proportion of total power dissipation as the feature size of semiconductor technology continues to reduce as shown in Figure 1. It is predicted that leakage power will become comparable to dynamic power within only a few generations [Doyle et al. 2002; Karnik et al. 2002; Kim et al. 2003; Semiconductor Industry 2004; Jones 2004]. Therefore, power gating to reduce leakage power should be used in addition to clock gating, which is only able to reduce the dynamic power [Kao and Chandrakasan 2000; Butts and Sohi 2000; Hu et al. 2004]. Recent studies have attempted to reduce leakage power using integrated architecture and compiler power-gating mechanisms [Dropsho et al. 2002; Yang et al. 2002; You et al. 2002, 2006; Rele et al. 2002; Zhang et al. 2003]. This approach involves compilers inserting instructions into programs to shut down and wake up components whenever appropriate, based on a data-flow analysis or profiling analysis. While early studies showed this approach to be effective, there are concerns about the amount of power-control instructions being added to programs with increasing numbers of components being equipped with power-gating controls in system-on-a-chip (SoC) design platforms for embedded systems. Note that architecture designers can customize the processor with unique operation functions [Ip et al. 2002; Gonzalez 2000; Tsutsui et al. 2002]. For example, one may have extensible instructions for modules of cryptography, 3D graphics, and motion estimation, as well as variety of wireless communication modules, etc.

In this article we present a *sink-n-hoist* framework for a compiler to generate balanced scheduling of power-gating instructions. Our framework attempts to merge several power-gating instructions into a single compound instruction, thereby reducing the amount of power-gating instructions issued. Note that whilst power-gating instructions can significantly reduce leakage power, they produce recovery penalties and increase the execution time and code size of programs. Figure 2 illustrates an example of power-gating control. The lefthand panel of the figure shows two different components in use, the center panel illustrates the current practice of attempting to issue power-on and power-off

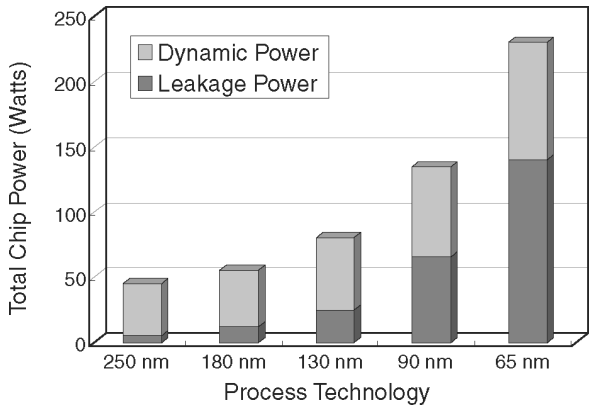

Fig. 1. Leakage power trend.

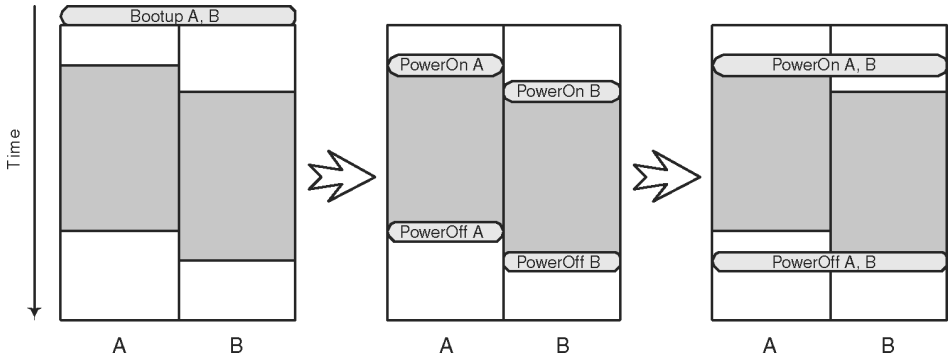

Fig. 2. Scenarios of power-gating controls (the shaded components are those in use).

instructions for these two hardware components separately, and the righthand panel shows our scheme that attempts to merge these instructions. In this article we provide a cost model and software foundation to guide this process. Our solution includes a set of data-flow equations for code motion of power-gating instructions. Our work combines a theoretical foundation and step-by-step framework for moving, grouping, and merging power-gating instructions. We have performed experiments that incorporate our compiler analysis and scheduling policies into SUIF compiler tools, and simulate the energy consumption using Wattch toolkits [Brooks et al. 2000]. Experimental results obtained using the DSPstone benchmark suite demonstrate that our mechanisms are effective in reducing both the amount of power-gating instructions and the power consumption relative to previous methods. Our sink-n-hoist framework for merging power-gating instructions reduces the code size by an average of 47.8%, and also further reduces the energy consumption due to the block version of power-gating instructions, giving better power and performance than the pointwise power-gating instructions.

The remainder of this article is organized as follows. Section 2 describes a machine architecture for the target platform, Section 3 overviews the

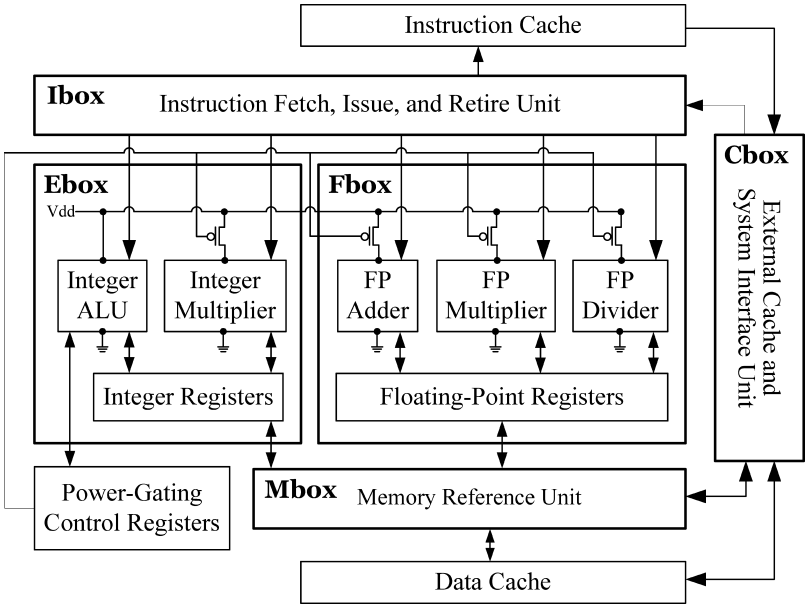

Fig. 3. DEC Alpha 21264 architecture with power-gating support.

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 study, Section 6 describes related work, and Section 7 concludes.

2. MACHINE ARCHITECTURE

The architecture model in our design has an instruction set that supports power-gating control at the component level. We focus on reducing the power consumption of certain components by invoking power-gating technology. Power gating is analogous to clock gating, except that devices are powered off by switching off their supply voltage, rather than the clock. This can be implemented by forcing transistors to be off or using MTCMOS (multithreshold voltage CMOS technology) to increase the threshold voltage [Butts and Sohi 2000; Kao and Chandrakasan 2000; Roy and Prasad 1992; Hu et al. 2004].

Figure 3 illustrates an example of our target machine architecture based on a DEC Alpha 21264 processor with an instruction fetch, issue, and retire unit (Ibox), a block of integer-function units (Ebox), a block of floating-point-function units (Fbox), a memory reference unit (Mbox), and an external cache and system interface unit (Cbox) [Compaq 1999]. In the adapted DEC Alpha 21264 architecture model, Ebox and Fbox were equipped with power-gated functions. The power state of each unit is controlled by the 64-bit integer power-gating control register (PGCR). In this case, 1 bit is used for the integer multiplier unit and 3 for the floating-point function units. Setting the power-gating bit to true powers on the corresponding module, and clearing the bit to 0 powers off the corresponding module immediately in the following clock cycle. A new

Leakage-Power-Reduction Framework

Input: A source program.
Output: The program with power-gating controls.

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. Produce the power-gating instructions.

Fig. 4. The leakage-power-reduction framework.

instruction was implemented to control units with the power-gated function by moving the appropriate value from a general-purpose register to the PGCR. The integer ALU unit is always powered on, since it takes the responsibility for moving data to the PGCR.

3. LEAKAGE-POWER-REDUCTION FRAMEWORK

This section presents the compiler framework for implementing power-gating mechanisms to reduce leakage-power dissipation. We have previously presented a data-flow analysis framework, called *component-activity data-flow analysis (CADFA)*, to estimate the component activities on a microprocessor within a given program [You et al. 2002, 2006]. The analysis collects the information of the utilization of components at each point in the program. Power-gating-instruction scheduling is then performed to determine whether, where, and when power-gating controls should be employed so as to produce power reduction. Finally, power-gating instructions are inserted into the program accordingly. In the current study, we present a sink-n-hoist framework, applied in the phase immediately before power-gating instructions are inserted, to generate balanced scheduling of power-gating instructions. Our solution attempts to merge several power-gating instructions into a single compound instruction. Figure 4 presents the compiler flow of the leakage-power-reduction framework. In the figure, steps I, II, and III are conventional [You et al. 2006, 2002], and steps IV and V are proposed in this article to merge power-gating instructions. Steps I and II involve performing a component-activity data-flow analysis, step III decides if and where power-gating instructions should be inserted, step IV attempts to merge the power-gating instructions with our proposed sink-n-hoist framework, and step V produces the power-gating instructions. A motivating example of power-gating control in three floating-point units (ALU, multiplier, and divider) with this framework is illustrated in Figure 5, where each item shows the status of a component on a timeline, and a shaded item represents one that it is in use. Three scenarios are considered: leftmost items show the case without power-gating controls; middle items show the case when steps I, II, III, and V in the framework are applied; and the rightmost items show 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.

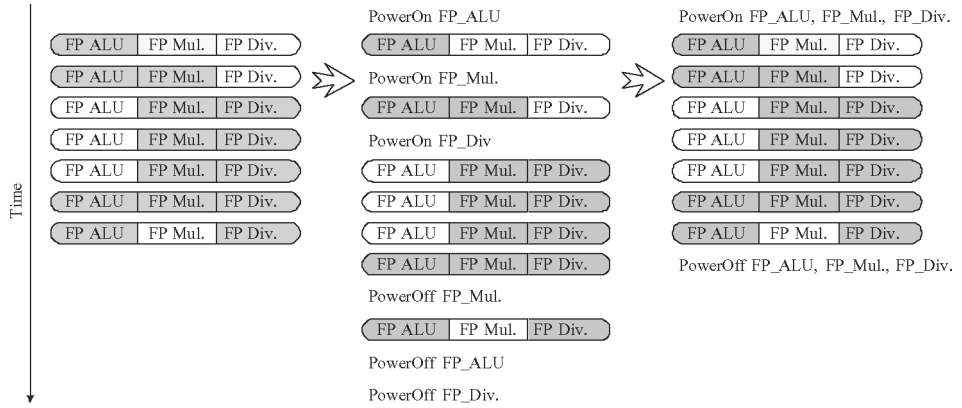

Fig. 5. An example of power-gating controls over floating-point (FP) units (shaded components are those in use).

In Sections 3.1 and 3.2, we describe the methods in steps II and III, and then steps IV and V with sink-n-hoist analysis for the code motion of power-gating instructions in Section 4.

3.1 Component-Activity Data-Flow Analysis

The goal of CADFA is to determine the utilization of components at each point in a program using a set of data-flow equations. We say a *component activity* c is generated at a block b if a component is required for execution, represented by $\mathsf{COMPONENT}_{loc}(b)$, and that it is killed if the component is released by the request, represented by $\mathsf{COMPONENT}_{blk}(b)$. 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 that are required for the first cycle of execution.

— $\mathsf{COMPONENT}_{blk}(b)$ is a set of components that are released by the execution at block b.

— $\mathsf{COMPONENT}_{in}(b)$ is a set of components that are required for execution at the beginning of block b.

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

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

— $\mathsf{COMPONENT}_{out}(b)$ is a set of components that are required for execution at the end of block b.

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

$\mathsf{COMPONENT}_{out}(b)$ can be interpreted as the information at the end of a statement, being either generated within the statement or entering at the beginning and not being killed as control flows through the statement.

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

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

where Ω is the universal set.

3.2 Power-Gating-Instruction Scheduling

Once the utilization information of components has been obtained, we can insert power-gating instructions into programs at the appropriate points (i.e., beginning and 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 whether the insertion of power-gating instructions will provide energy-consumption benefits.

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

where functions $\mathbb{E}$ and $\mathbb{P}$ return the value of energy and power consumption, respectively; $\mathbb{E}_{off}(C)$ and $\mathbb{E}_{on}(C)$ represent the energy consumption of issuing a power-off and a power-on instruction for component C, respectively; $\mathbb{P}_{leak}(C)$ 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 aforementioned inequality

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

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

The obtained component-activity information and cost model for deciding whether power-gating instructions should be employed allow us to consider scheduling mechanisms when inserting the power-gating instructions into given programs. Since the time required to instigate power-gating controls on components is influenced by conditional branches in programs, we propose the following set of scheduling policies with power-gating instructions: *Basic_Blk_Sched*, *MIN_Path_Sched*, and *AVG_Path_Sched*. 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 dataflow equation. We call this scheme *Basic_Blk_Sched*. Another case to consider is that of an inactive block containing conditional branches, since the lengths of

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

the, say, two inactive blocks that follow the branch targets may be different. For example, only one of the branchings may benefit from power gating, in which case instigating power-gating control in one branch when the other 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 may satisfy the cost model and the other 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 criterion for power gating. Finally, since the behavior of program branches depends on both the structure of and the input data to 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, rather than the minimum length. These three scheduling policies have been described in detail previously [You et al. 2002].

4. SINK-N-HOIST ANALYSIS

The main idea of sink-n-hoist analysis is to reduce the problem of excessive addition of instructions 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; that is, postponing the issuing of power-off instructions and bringing forward the issuing of power-on. This will result mainly in improvements to code size, but also in performance and energy via grouping effects. For instance, a power-off instruction can be postponed for several cycles to be merged with adjacent power-off instructions. Nevertheless, a maximum number of cycles to be sunk or hoisted should be set, since sinking or hoisting a power-gating instruction will increase leakage dissipation. A cost model is given next to determine the feasibility. For a component C, we have

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

where SINK−SLK is the number of cycles for which a power-off statement (or instruction)[2] is sunk, (i.e., the power-off statement is delayed for SINK−SLK cycles), $\mathbb{E}_{fet-dec-off}(C)$ returns that of fetching and decoding a power-off instruction, $\mathbb{E}_{exe-off}(C)$ returns that of executing a power-off instruction, and N is the number of power-gated components. Note that the sum of $\mathbb{E}_{fet-dec-off}(C)$ and $\mathbb{E}_{exe-off}(C)$ is equal to $\mathbb{E}_{off}(C)$. The righthand side of the inequality represents energy consumed when the power-off statement is delayed for SINK−SLK cycles and merged with other $(N-1)$ power-off statements, while the lefthand side represents the energy consumed when the power-off statement is called immediately after the end of an active interval. In consequence, we have a maximum sinkable slack for each component C, called $\mathsf{MAX{-}SINK{-}SLK}_C$, that sustains the

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

Sink-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 sinkable analysis and hoistable analysis, as in Eqs. (3)–(6).
(2) Perform grouping-off analysis, grouping-on analysis, and grouping-switch analysis, as in Eqs. (7)–(12)
(3) Perform power-gating-instruction placement.

Fig. 6. Sink-n-hoist algorithm.

previous inequality.

$$\mathsf{MAX{-}SINK{-}SLK}_C = \left\lfloor \frac{(N-1)\cdot \mathbb{E}_{fet-dec-off}(C)}{N\cdot(\mathbb{P}_{leak}(C)-\mathbb{P}_{rleak}(C))} \right\rfloor$$

Similarly, we have a maximum hoistable slack for each component.

$$\mathsf{MAX{-}HOIST{-}SLK}_C = \left\lfloor \frac{(N-1)\cdot \mathbb{E}_{fet-dec-on}(C)}{N\cdot(\mathbb{P}_{leak}(C)-\mathbb{P}_{rleak}(C))} \right\rfloor$$

With such cost constraints as the basis, we now present a set of data-flow equations to collect information for the code motion of power-gating instructions. Figure 6 shows the algorithm for sink-n-hoist analysis. The complete set of equations used is presented in Figure 7. Sink-n-hoist analysis consists of three main phases: (1) *sinkable analysis* and *hoistable analysis*, which compute the information of possible positions for each power-gating instruction; (2) *grouping-off analysis*, *grouping-on analysis*, and *grouping-switch analysis*, which group together the power-gating instructions that can be merged; and (3) *power-gating-instruction placement*, which determines appropriate positions for power-gating instructions.

4.1 Sinkable Analysis and Grouping-Off Analysis

The predicates for collecting SINKABLE and GROUP−OFF information are given as follows. The SINKABLE predicate gives that to collect the information required to determine how far the power-off instructions of component activities can be sunk, and the GROUP−OFF predicate gives that to partition power-off instructions into groups. We can then use this information to group them by selecting the produced instructions:

—$\mathsf{SINKABLE}_{loc}(b)$ is a set of power-off statements that occur within block b and which can be safely moved to the end of the block. Each statement is associated with an integer number $\mathsf{SINK{-}SLK}_C^b$, which is the slack time for component C for indicating how many cycles the power-off statement can be sunk at block b. The initial value of $\mathsf{SINK{-}SLK}_C^b$ is set as $\mathsf{MAX{-}SINK{-}SLK}_C$.
—$\mathsf{SINKABLE}_{blk}(b)$ is a set of power-off statements that cannot be safely moved from the start to the end of bock b; that is, a set of power-off statements whose associated $\mathsf{SINK{-}SLK}_C^b$ value is zero.

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

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

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

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

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

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

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

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

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

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

$$\mathsf{GROUP\text{–}SWH}_{in}(b) = \begin{cases} \{\mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{–}SWH}_{out}(p)))\} \\ \emptyset, \quad \text{if } \mathbb{MIN}_{p \in Pred(b)}(\Phi(\mathsf{GROUP\text{–}SWH}_{out}(p))) = \infty \end{cases} \quad (11)$$

$$\mathsf{GROUP\text{–}SWH}_{out}(b) = \mathsf{GROUP\text{–}SWH}_{loc}(b) \cup (\mathsf{GROUP\text{–}SWH}_{in}(b) - \mathsf{GROUP\text{–}SWH}_{blk}(b)) \quad (12)$$

$^{\dagger}\mathbb{MIN}$ is a function that returns the minimum value of its parameters

Fig. 7. Component-activity data-flow analysis and sink-n-hoist analysis equations.

—$\mathsf{SINKABLE}_{in}(b)$ is a set of power-off statements that can be safely moved to the beginning of block b.

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

The value of $\mathsf{SINK\text{–}SLK}_C^b$ would be the minimum one among the predecessors of block b if the values of $\mathsf{SINK\text{–}SLK}_C^p$ for each p are inconsistent with each other, where p is a predecessor of block b. This means that the sinkable slack from one predecessor would be reduced if other predecessors have a smaller sinkable slack. This implements the consideration that a power-off statement should not be sunk to a position that may cause a reverse effect. Moreover, the value of each $\mathsf{SINK\text{–}SLK}_C^b$ is decreased by one in accordance with the following definition.

$$\mathsf{SINK\text{–}SLK}_C^b = \mathbb{MIN}_{p \in Pred(b)}(\mathsf{SINK\text{–}SLK}_C^p) - 1$$

—$\mathsf{SINKABLE}_{out}(b)$ is a set of power-off statements that can be safely moved to the end of block b.

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

The value of $\mathsf{SINK\text{–}SLK}_C^b$ is given from that of the associated $\mathsf{SINK\text{–}SLK}_C^b$ in $\mathsf{SINKABLE}_{loc}(b)$ if there exists a power-off-C statement in $\mathsf{SINK\text{–}ABLE}_{loc}(b)$;

otherwise, it is given from the one in $\mathsf{SINKABLE}_{in}(b)$. In fact, $\mathsf{SINKABLE}_{out}(b)$ presents the set of power-off statements (whether sunk or not) that can be issued at block b.

We now give the data-flow equations for GROUP−OFF, whose main concept is to partition power-off instructions into groups in which the possible positions of each such instruction (information that can be derived from $\mathsf{SINKABLE}_{out}$) overlaps with at least one of those of the other instructions. In other words, it clusters together power-off instructions that might be merged. The predicates for computing GROUP−OFF are as follows:

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

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

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

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

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

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

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

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

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

—$\mathsf{GROUP{-}OFF}_{out}(b)$ is an integer singleton (a group number) that can be assigned to the end of block b or an empty set.

$$\begin{aligned} \mathsf{GROUP{-}OFF}_{out}(b) = \mathsf{GROUP{-}OFF}_{loc}(b) \cup \\ (\mathsf{GROUP{-}OFF}_{in}(b) - \mathsf{GROUP{-}OFF}_{blk}(b)) \end{aligned}$$

In fact, the element in $\mathsf{GROUP{-}OFF}_{out}(b)$ gives the group number to which block b belongs.

We now give a running example to illustrate how the analysis works. Suppose that two components, **A** and **B**, are considered for analyses. Given a control-flow

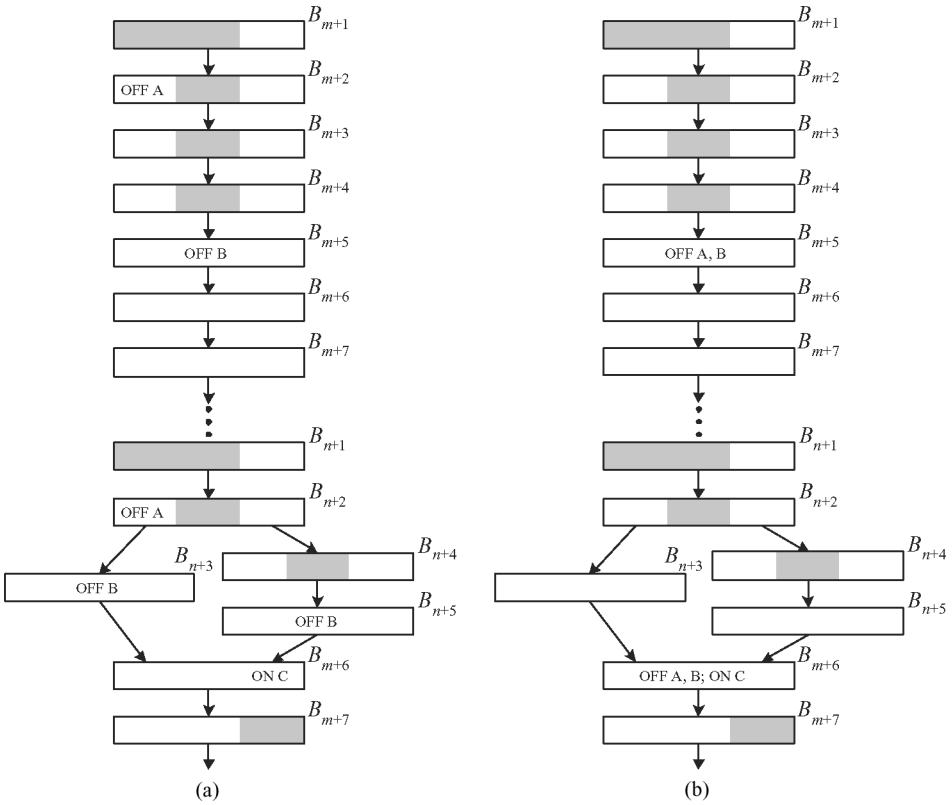

Fig. 8. An example of sinking power-off statements, where the left and right halves of a block correspond to the activity of components **A** and **B**, respectively (shaded components are those in use).

graph as shown in Figure 8(a), where each block in the graph contains only a statement, we can determine where power-gating statements should be located by performing steps I, II, III, and V in Figure 4. This includes CADFA and power-gating-instruction scheduling.

In this example, it is found that components **A** and **B** should be powered off at blocks B_{m+2} and B_{n+2}, and at blocks B_{m+5}, B_{n+3}, and B_{n+5}, respectively. To reduce the amount of power-gating instructions issued, we apply sinkable analysis. By the definition of $\mathsf{SINKABLE}_{loc}(b)$, a set of power-off statements that occur within block b, we have $\mathsf{SINKABLE}_{loc}(B_{m+2}) = \{\text{PowerOff } \mathbf{A}(4)\}$, $\mathsf{SINKABLE}_{loc}(B_{m+5}) = \{\text{PowerOff } \mathbf{B}(2)\}$, $\mathsf{SINKABLE}_{loc}(B_{n+2}) = \{\text{PowerOff } \mathbf{A}(4)\}$, $\mathsf{SINKABLE}_{loc}(B_{n+3}) = \{\text{PowerOff } \mathbf{B}(2)\}$, and $\mathsf{SINKABLE}_{loc}(B_{n+5}) = \{\text{PowerOff } \mathbf{B}(2)\}$, where the numbers in parentheses indicate the value of the associated $\mathsf{SINK{-}SLK}_C$ (in fact, the values come from $\mathsf{MAX{-}SINK{-}SLK}_A$ and $\mathsf{MAX{-}SINK{-}SLK}_B$), and $\mathsf{SINKABLE}_{loc}$ for the other blocks is an empty set. To simplify representation, the word "PowerOff" is removed and the value of the associated $\mathsf{SINK{-}SLK}_C$ is superscripted (e.g., $\mathsf{SINKABLE}_{loc}(B_{m+2}) = \{\mathbf{A}^4\}$). Table I gives the computation results of $\mathsf{SINKABLE}_{blk}(b)$, $\mathsf{SINKABLE}_{in}(b)$, and

Table I. SINKABLE Predicates for the Example in Figure 8

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

‡The superscript represents the value of the associated $\mathsf{SINK}-\mathsf{SLK}^b_C$.

Table II. GROUP−OFF Predicates for the Example in Figure 8

Block	$\mathsf{GROUP}-\mathsf{OFF}_{loc}(b)$	$\mathsf{GROUP}-\mathsf{OFF}_{blk}(b)$	$\mathsf{GROUP}-\mathsf{OFF}_{in}(b)$	$\mathsf{GROUP}-\mathsf{OFF}_{out}(b)$
B_{m+1}				
B_{m+2}	{1}			{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}			{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}	

$\mathsf{SINKABLE}_{out}(b)$ for each block. Note that all elements without a designated value in this table represent empty sets. Actually, $\mathsf{SINKABLE}_{out}(b)$ 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 words, the power-off statements of component **A** can be issued at blocks B_{m+2} to B_{m+5} and blocks B_{n+2} to B_{n+6}. We then compute $\mathsf{GROUP}-\mathsf{OFF}_{loc}(b)$, $\mathsf{GROUP}-\mathsf{OFF}_{blk}(b)$, $\mathsf{GROUP}-\mathsf{OFF}_{in}(b)$, and $\mathsf{GROUP}-\mathsf{OFF}_{out}(b)$ for each block so as to group those blocks in which the power-off statements of the component that appears in this group should be issued exactly once. Table II gives the grouping results: Blocks B_{m+2} to B_{m+6} belong to group 1 and blocks B_{n+2} to B_{n+6} belong to group 2.

4.2 Hoistable and Grouping-On Analysis

Hoistable and grouping-on analyses are similar to sinkable and grouping-off analyses, except that 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:

—$\mathsf{HOISTABLE}_{loc}(b)$ is a set of power-on statements that occur within block b and which can be safely moved to the start of the block. Each statement is associated with an integer number $\mathsf{HOIST{-}SLK}_C^b$, which is the slack time for component C that indicates how many cycles the power-on statement can be hoisted at block b. The initial value of $\mathsf{HOIST{-}SLK}_C^b$ is set as $\mathsf{MAX{-}HOIST{-}SLK}_C$.

—$\mathsf{HOISTABLE}_{blk}(b)$ is a set of power-on statements that cannot be safely moved from the end to start of bock b; that is, the set of power-on statements whose value of the associated $\mathsf{HOIST{-}SLK}_C^b$ is zero.

—$\mathsf{HOISTABLE}_{out}(b)$ is a set of power-on statements that can be safely moved to the end of block b.

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

The value of $\mathsf{HOIST{-}SLK}_C^b$ would be the minimum among those successors of block b if the values of $\mathsf{HOIST{-}SLK}_C^s$ for each s are inconsistent with each other, where s is a successor of block b. This means that the hoistable slack from one successor would be reduced if other successors have a smaller hoistable slack. This implements the consideration that a power-on statement should not be hoisted to a position that may cause a reverse effect. Moreover, the value of each $\mathsf{HOIST{-}SLK}_C^b$ is decreased by one in accordance with the following definition.

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

—$\mathsf{HOISTABLE}_{in}(b)$ is a set of power-on statements that can be safely moved to the start of block b.

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

The value of $\mathsf{HOIST{-}SLK}_C^b$ is given from the value of the associated $\mathsf{HOIST{-}SLK}_C^b$ in $\mathsf{HOISTABLE}_{loc}(b)$ if there exists a power-on-C statement in $\mathsf{HOISTABLE}_{loc}(b)$; otherwise, it is given from the one in $\mathsf{HOISTABLE}_{out}(b)$. In fact, $\mathsf{HOISTABLE}_{in}(b)$ presents the set of power-on statements (hoisted or not) that can be issued at block b.

—$\mathsf{GROUP{-}ON}_{loc}(b)$ is a set with at most one element (i.e., a singleton or empty set) in which the element (if it exists) is an integer representing a group number and never appears in other sets of $\mathsf{GROUP{-}ON}_{loc}$. Block b belongs to the group it enumerates and is the beginning block of a set of successive blocks if $\mathsf{GROUP{-}ON}_{loc}(b)$ is not empty. The $\mathsf{GROUP{-}ON}_{loc}(b)$ 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 numbers (in the sets of $\mathsf{GROUP{-}ON}_{loc}$ of all blocks) are unique is to assign each element to the value of an integer counter, and increment the counter once an element is assigned.

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

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

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

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

In addition, we can replace all of the $\mathsf{GROUP{-}ON}_{out}$ set of its predecessors by $\mathsf{GROUP{-}ON}_{in}(b)$ if the $\mathsf{GROUP{-}ON}_{out}$ set of the predecessor of b is not empty. Note that this provides opportunity for further grouping.

—$\mathsf{GROUP{-}ON}_{out}(b)$ is an integer singleton (a group number) that can be assigned to the end of block b or to an empty set.

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

In fact, the element in $\mathsf{GROUP{-}ON}_{out}(b)$ gives the group number to which block b belongs.

4.3 Grouping-Switch Analysis

In order to collect more grouping information for later analysis, we introduce *grouping-switch analysis*, which groups together all power-on and power-off instructions that might be merged. The analysis is similar to grouping-off and grouping-on analyses. The predicates for computing GROUP−SWH are as follows:

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

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

or

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

A simple way to ensure that all numbers in the sets of $\mathsf{GROUP{-}SWH}_{loc}$ of all blocks are unique is to assign each element to the value of an integer counter, and increment the counter once an element is assigned.

—GROUP−SWH$_{blk}(b)$ is a universal set of integers, namely Ω, or an empty set. Block b is one (or the only) of the end blocks of a set of successive blocks if GROUP−SWH$_{blk}(b)$ is not empty, which is the case when

$$\text{SINKABLE}_{out}(b) = \emptyset \text{ and } \bigcup_{p \in Pred(b)} \text{SINKABLE}_{out}(p) \neq \emptyset$$

and

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

—GROUP−SWH$_{in}(b)$ is an integer singleton (a group number) that can be assigned to the start of block b or to an empty set.

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

In addition, we can also replace all of the GROUP−SWH$_{out}$ set of its predecessors by GROUP−ON$_{in}(b)$ if the GROUP−SWH$_{out}$ set of the predecessor of b is not empty. Note that this provides opportunity for further grouping.

—GROUP−SWH$_{out}(b)$ is an integer singleton (a group number) that can be assigned to the end of block b or to an empty set.

$$\text{GROUP−SWH}_{out}(b) = \text{GROUP−SWH}_{loc}(b) \cup (\text{GROUP−SWH}_{in}(b) - \text{GROUP−SWH}_{blk}(b))$$

In fact, the element in GROUP−SWH$_{out}(b)$ gives the group number to which block b belongs.

4.4 Power-Gating-Instruction Placement

We use information from the SINKABLE$_{out}$, HOISTABLE$_{in}$, GROUP−OFF$_{out}$, GROUP−ON$_{out}$, and GROUP−SWH$_{out}$ predicates described in Sections 4.1, 4.2, and 4.3 to determine how to place power-gating instructions, that is, whether power-gating instructions should be combined or issued separately.

Figure 9 outlines an algorithm for placing power-gating instructions in a group-by-group manner. It first determines all possible policies for issuing power-gating instructions; a legitimate policy is one in which all power-gating instructions are issued at block b in which SINKABLE$_{out}(b)$ or HOISTABLE$_{in}(b)$ is not empty, and where each type of power-gating instruction appearing within a group must be issued exactly once only. It then uses an energy-cost model (including leakage energy, the energy associated with issuing power-off instructions, etc.) to determine which policy results in the lowest energy consumption. The algorithm for power-gating-instruction placement is basically a method of exhaustion, yet can be regarded as a simple and valid method. Towards the actual time spent in our experiments the process only contributes a very small fraction: less than 0.6% of our proposed framework.

In the following, we elaborate the idea by continuing the example presented in Section 4.1. An energy-cost model is established with the information of SINKABLE$_{out}$ and GROUP−OFF$_{out}$, and evaluated for each case of issuing power-off-instruction policies under the guideline that power-off instructions must be

Algorithm for Power-Gating-Instruction Placement

Input: SINKABLE$_{out}$, GROUP–OFF$_{out}$, HOISTABLE$_{in}$, and GROUP–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 lowest power*/
    best_policy = get_best_policy(policy_list);

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

Fig. 9. Power-gating-instruction placement.

issued at the block in which SINKABLE$_{out}$ is not empty, and each type of power-gating instruction appearing within a group must be issued exactly once only. For example, 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 1". The policy with minimum energy cost as evaluated by the model is chosen, since this should give the lowest power consumption. Finally, power-off instructions are inserted at appropriated points, as shown in Figure 8(b): The power-off statements within each group are merged.

5. EXPERIMENTAL RESULTS

5.1 Platform

We used a DEC-Alpha-compatible architecture with the power-gating controls and instruction sets as described in Figure 3 as the target architecture for our experiments. The proposed leakage-power-reduction framework was incorporated into the compiler tool with SUIF [Stanford Compiler Group 1995] and Machine-SUIF [Smith 1998], and evaluated by the Wattch simulator with a 0.10-μm process parameter and a 1.9-V supply voltage [Brooks et al. 2000]. Table III summaries the baseline configuration of the simulator in our experiment. By default, the simulator performed out-of-order executions. We used the "-issue:inorder" option in the configuration so that instructions would be executed in order for ensuring the correctness of power-gating controls. Nevertheless, our approach can also be applied to out-of-order issue machines if the additional hardware supports proposed in You et al. [2006] are employed. The benchmarks used in our experiments were from the floating-point version

Table III. Baseline Processor Configuration

Parameter	Value
Clock	600 MHz
Process parameters	0.10 μm, 1.9 V
Instruction issuing	In-order
Decode width	8 instructions/cycle
Issue width	8 instructions/cycle
Commit width	8 instructions/cycle
RUU size	128
LSQ size	64
Functional units	4 integer ALUs 1 integer multiply/divide unit 4 FP ALUs 1 FP multiply/divide unit
Register files	32 64-bit integer registers 32 64-bit FP registers 1 64-bit power-gating control register

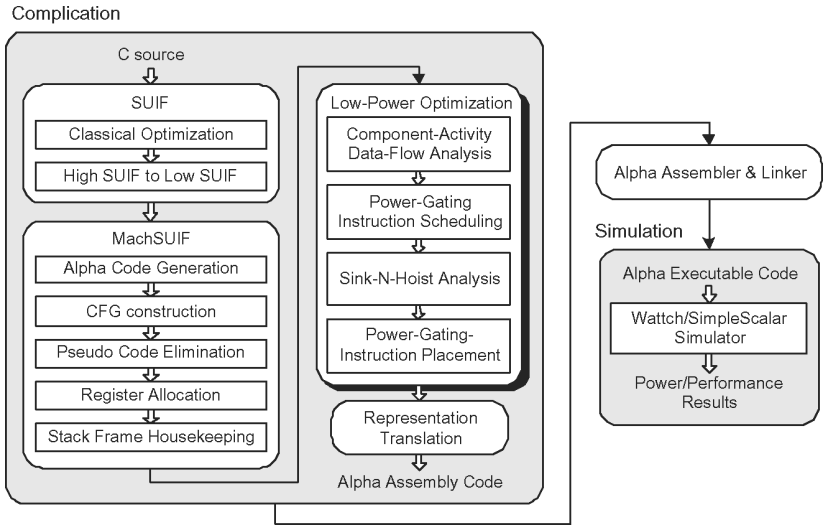

Fig. 10. Compilation and simulation framework.

of the DSPstone benchmark suite [Zivojnovic et al. 1994]. The average IPC (instructions per cycle) of the benchmarks is 0.36 with the configuration in Table III.

Figure 10 illustrates the phases in the compilation and simulation framework. We incorporated the low-power optimization phase just before code generation; that is, after all traditional performance optimizations are performed. Hence, the additional phase has little or no influence on performance; it only inserts power-gating instructions and thus barely affects execution behavior. The implementation was based on SUIF2 and the *Control Flow Graph (CFG)* and *Machine* libraries from Machine-SUIF. Programs were first transformed from high-SUIF to low-SUIF format with SUIF, and then translated to the machine-level or instruction-level CFG form with Machine-SUIF. The proposed four

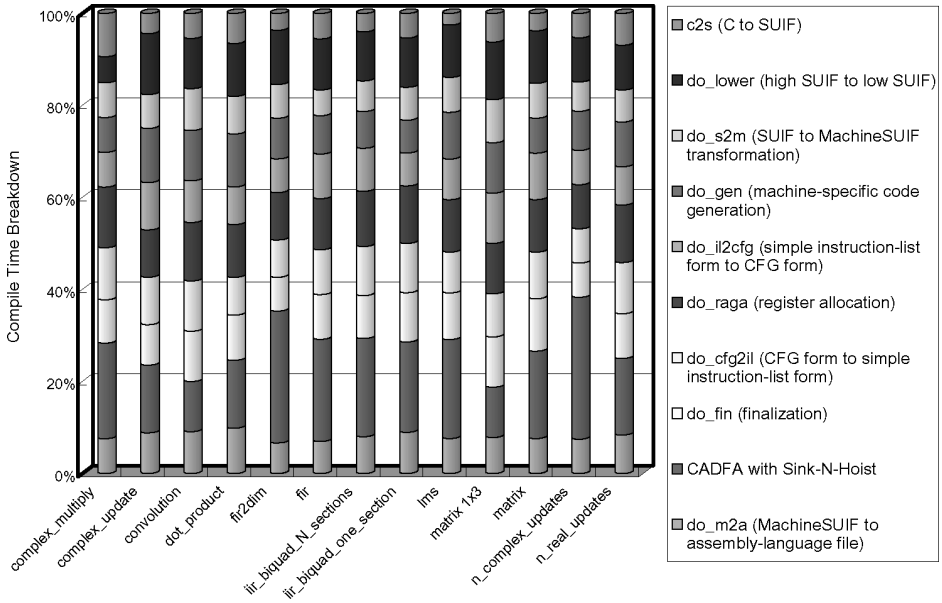

Fig. 11. Compile-time breakdown.

components of the low-power optimization phase (implemented as a MachineSUIF pass) were then performed, and finally, the compiler generated DEC Alpha assembly codes with power-gating controls. We also examined the breakdown of the overall compile time, as shown in Figure 11. It is observed that the proposed approach, CADFA with sink-n-hoist, contributes an average of 19.2% of overall compile time.

In addition, the power-gating mechanism is absent in the original DEC Alpha processor, and thus there are no power-gating instructions in its instruction set. We therefore treated power-gating instructions as a set of special instructions so that they are recognized by the DEC Alpha assembler and linker: "stl $24, negative_offset($31)", where *negative offset* is a negative integer that is used for indicating the functional unit to be powered on or off. The instruction stores the value of register $24 into the memory address below zero, which is an invalid memory address ($31 is a constant zero register) and should never be generated by standard compilers. To prevent processors from accessing the invalid memory addresses, we made a small modification in Wattch: When the instruction decoder deciphers such instructions, it extracts the power-gating information and converts it to an NOP (no-operation) instruction. Furthermore, since Wattch does not model leakage at the component level per se, we assumed that leakage power contributes 10% of the total power consumption. Furthermore, we assumed that wake-up operations of power-gating controls have a 3-cycle latency [Hu et al. 2004] and that it took 4 and 10 times the leakage energy per cycle to power a component off and on, respectively. The energy consumption of fetching and decoding a power-gating instruction was assumed to be 2 times the leakage power. Also, the baseline data was provided by the

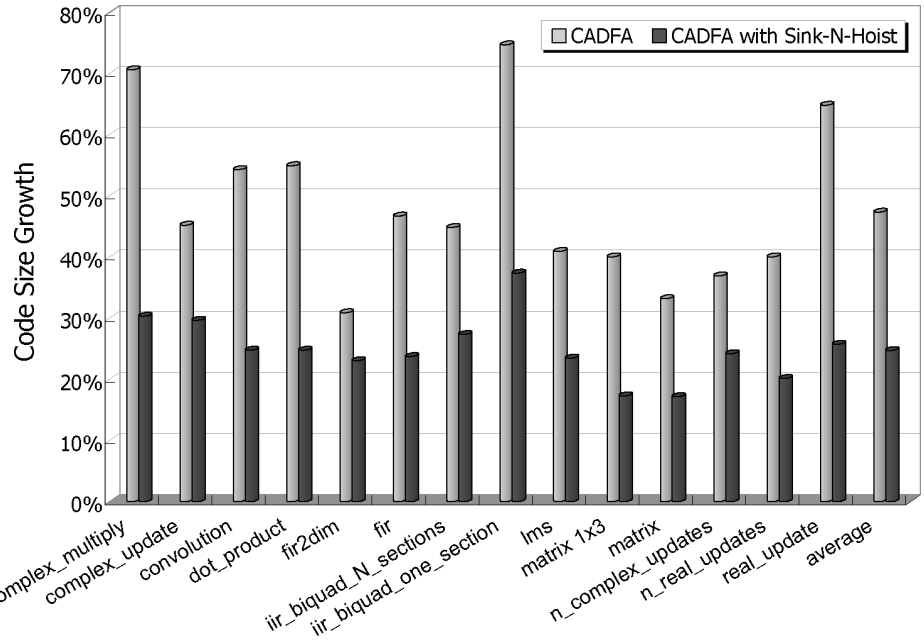

Fig. 12. Code size growth.

power estimation of Wattch *cc3* with a clock-gating mechanism, which gates the clocks of those unused resources in multiported hardware to reduce the dynamic power; however, leakage power is still exuded.

5.2 Results and Discussion

The results from three types of experiment are compared: (1) no power-gating mechanism (baseline); (2) CADFA as from a previous work [You et al. 2006, 2002] in which only steps I, II, and III of Figure 4 were performed; and (3) sink-n-hoist analysis involving all phases in Figure 4. In addition, three policies for power-gating-instruction scheduling were proposed in step III of Figure 4 to deal with conditional branches in programs. Without loss of generality, we used the *Min_Path_Sched* policy to schedule power-gating instructions in this experiment.

Figures 12–14 give the compilation and simulation results of two approaches: CADFA and CADFA with sink-n-hoist when the integer multiplier, 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 12 presents the code-size growth due to power-gating instructions, which shows that sink-n-hoist reduces the code size by about 47.8% on average (from 60.3% to 25.4%) compared with the method without the sink-n-hoist framework, namely, CADFA. Moreover, our scheme also further reduces total energy consumption compared to that without the sink-n-hoist framework, which is due to the block version of the power-gating instructions giving better power and performance characteristics than the pointwise version. Figure 13

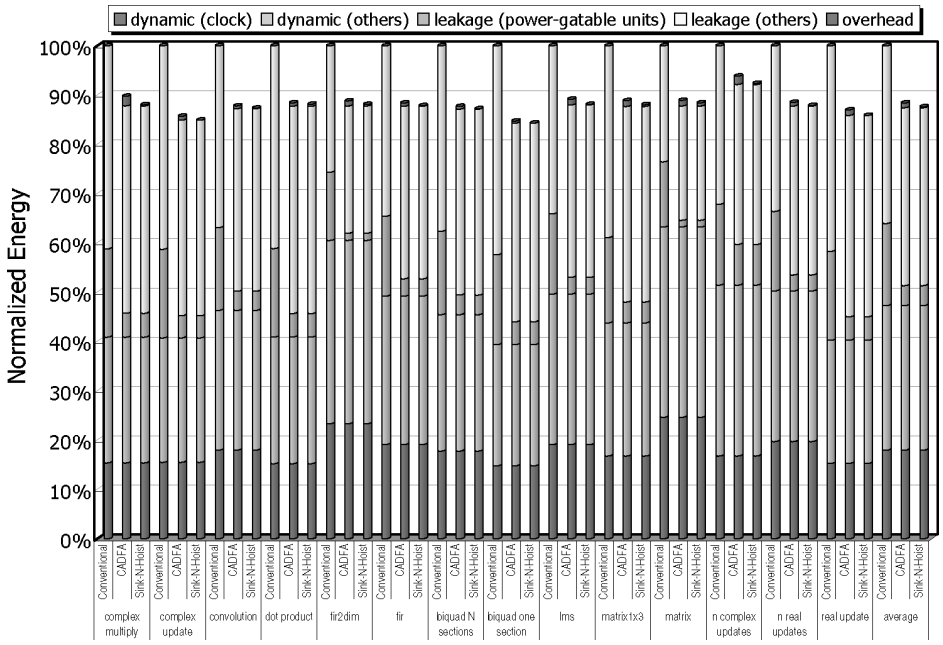

Fig. 13. Normalized total energy consumption.

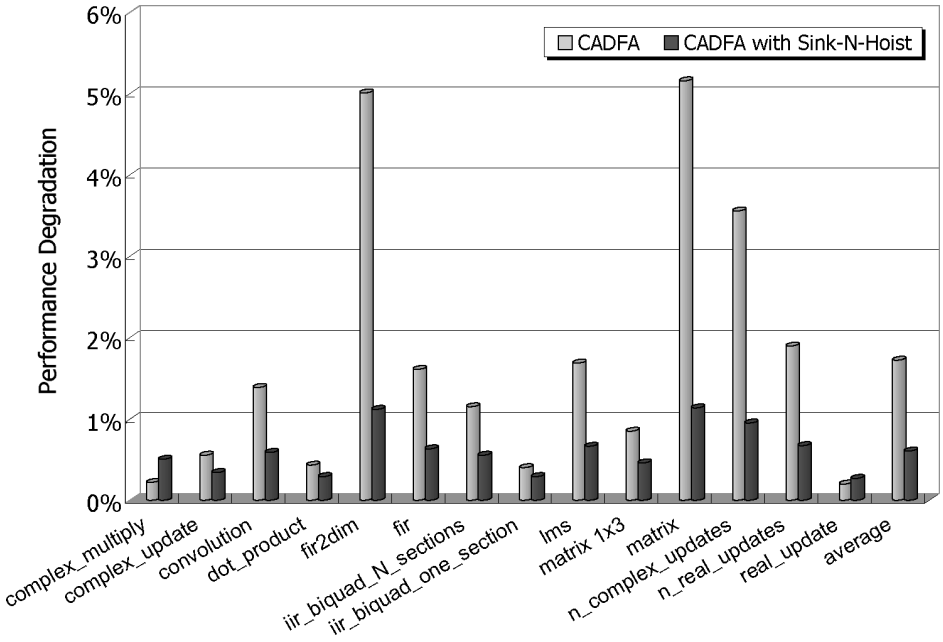

Fig. 14. Performance degradation.

illustrates the normalized energy breakdown with conventional, CADFA, and CADFA with sink-n-hoist compilation strategies. The energy consumption was measured by 5 categories: the dynamic energy dissipated by clock circuits and that by the whole processor except for clock circuits, the leakage energy dissipated by power-gatable units and that by the whole processor except for power-gatable units, and the overhead energy consumption due to extra power-gating instructions. The overhead includes not only the energy dissipated by power-gating instructions themselves, but also the negative impact of memory, buses, etc. The average impact of using CADFA and CADFA with sink-n-hoist are 0.98% and 0.20%, respectively, in which about 20% of the energy is contributed to power-gating operations and the other 80% to dissipation in the caches, fetch and decode units, buses, etc. Figure 13 shows that our scheme reduces average power by 11.9% compared with the conventional method. Note that the average reduction in total energy does not seem high, but this is attributable to the fact that only 3 types of functional units (the integer multiplier, floating-point adder, and multiplier) are under power-gating control in this experiment. In fact, the CADFA method has already achieved average energy reductions in combined dynamic and leakage power of 70.4% and 72.6% for the adder and multiplier, respectively [You et al. 2006, 2002]. Figure 13 also shows that our scheme is superior to CADFA in terms of energy reduction, which is also due to the block version of power-gating instructions improving power consumption more than the pointwise. In addition, we also compile the breakdown of the execution cycle in terms of function unit activities. It is observed that for the integer multiplier, floating-point adder, and floating-point multiplier, 76.4%, 76.2%, and 77.0% of idle cycles, respectively, were controlled with the power-gating mechanism by CADFA with the sink-n-hoist approach.

Figure 14 shows that the performance impact of power-gating mechanisms is less than 5% for most of the benchmarks for both CADFA and CADFA with sink-n-hoist. The only exceptions are *fir2dim* and *matrix*, which are due to the fact that the number of power-gating instructions placed within loops are much greater than for the other benchmarks. Therefore, *fir2dim* and *matrix* execute more power-gating operations, and thus consume more execution cycles. The performance degradation is reduced by an average of about 64.81% over the CADFA method. Our method exhibits an advantage over the one without the sink-n-hoist framework due to reduction in number of power-gating instructions. Note that the performance penalty is less than the increase in number of instructions, since most instructions are added outside the loop kernel. Nevertheless, the reduction in number of power-gating instructions still yields a performance advantage.

In addition, Figure 15 gives the normalized energy breakdown in four categories (dynamic energy by the whole processor, leakage energy dissipated by power-gatable units, leakage energy dissipated by the whole processor except the power-gatable units, and for overhead energy consumption due to extra power-gating instructions) with different configurations of the leakage contribution (from 10% to 90%). It shows that our technique is effective in helping leakage control at/beyond new technology generations. Generally, the

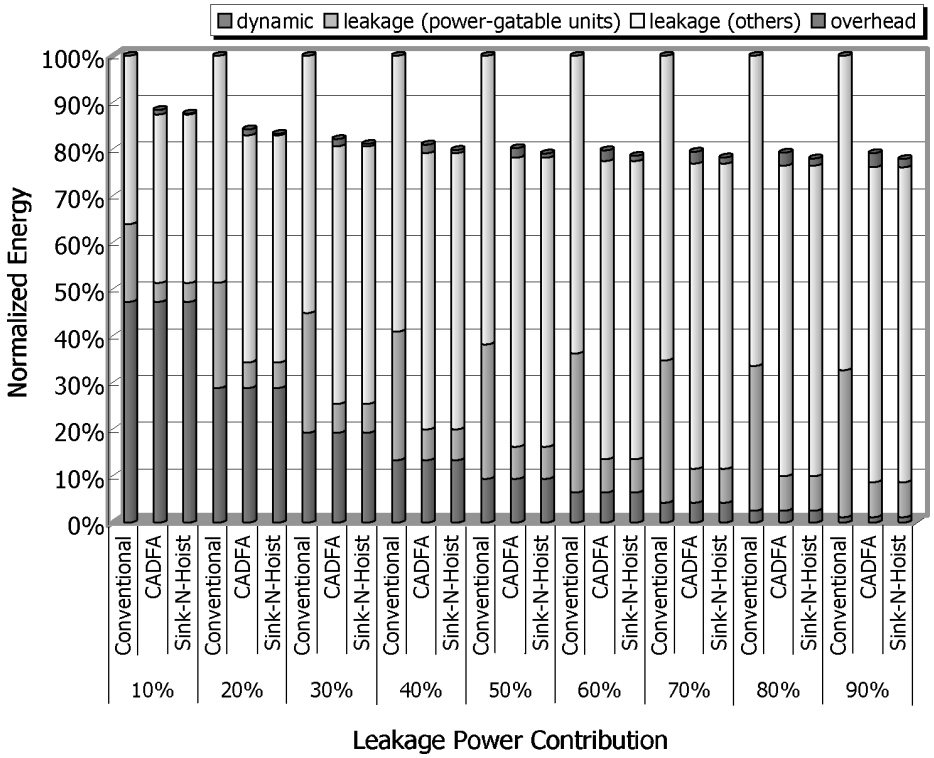

Fig. 15. Normalized energy consumption with different leakage contributions.

effectiveness of our approach becomes greater as leakage contribution rises. However, this trend stabilizes when the leakage contribution becomes greater than 70% due to the dominating leakage and growing impact on overheads in memory, bus, and other uncontrolled units with respect to power gating. Recall that we only attempt to do power-gating control on units of the integer multiplier, floating-point adder, and floating-point multiplier in this experiment setup.

6. RELATED WORK

Recent studies have attempted to reduce leakage power using integrated architecture and compiler power-gating mechanisms [Dropsho et al. 2002; Rele et al. 2002; You et al. 2006, 2002; Zhang et al. 2004, 2003]. Dropsho et al. [2002] proposed an analytical energy model for architecture-level analysis, and described the benefits of employing a dual-threshold-voltage technique to reduce subthreshold leakage current in the integer functional units of a processor. They also proposed a simple architecture design, called *gradual sleep*, to reduce the overhead of activating the sleep mode for smaller idle periods. The work of Rele et al. [2002] is based on a profiling approach to identify those blocks in which functional units are expected to be idle (based on the execution frequencies of each basic block), and then inserting off and on instructions at entry and exit

points of such blocks, respectively. You et al. [2002] proposed a more formal compiler methodology that uses a data-flow analysis approach to collect the information of activities of each functional unit at each point of a program, inserting power-gating instructions by using a scheduling algorithm to deal with the uncertainty of idle periods due to conditional branches. They also proposed an architecture to make power-gating controls applicable to out-of-order issue processors [You et al. 2006]. Aside from controlling leakage energy of functional units, Zhang et al. [2004] presented a compiler-directed approach that inserts power mode instructions for cache lines to control leakage energy consumed in the instruction cache.

The previously described approaches have shown that leakage power can be effectively suppressed with help from compilers. However, there are concerns about the amount of power-control instructions being added to programs as increasing numbers of components are equipped with power-gating controls in SoC design platforms. Whilst power-gating instructions can significantly reduce leakage power, they produce recovery penalties and increase the execution time and code size of programs. Our sink-n-hoist framework for a compiler solution attempts to merge several power-gating instructions into a single compound instruction so as to reduce the amount of power-gating instructions.

7. CONCLUSION

In summary, our experiments have demonstrated that the sink-n-hoist analysis framework proposed in this article improves code size, energy consumption, and performance. It reduces the overall energy consumption and code size growth by an average of about 0.9% and 47.8% , respectively, compared with the CADFA scheme without our sink-n-hoist approach, and impacts performance by an average of less than 1%. As the compiler phase is done one phase after another, our framework provides a sound theoretical foundation capable of working with other improvements, such as adding more slackness for low power. We are currently in the process of incorporating more components (such as cryptography modules) into our architecture and simulator. We expect that our scheme will be even more beneficial as more extensible modules are equipped with power-gating controls in SoC design platforms.

ACKNOWLEDGMENTS

The work was supported in part by the National Science Council (under grant numbers. NSC 95-2220-E-007-001 and NSC 95-2220-E-007-002), the Ministry of Economic Affairs (under grant numbers 95-EC-17-A-01-S1-034 and 96-EC-17-A-01-S1-034), and ITRI (under an ITRI/NTHU research grant). We are also grateful to the National Center for High-performance Computing for computer time and facilities.

REFERENCES

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

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture* (Vancouver, Canada), 83–94.

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

Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. 1992. Low-Power CMOS digital design. *IEEE J. Solid-State Circ. 27*, 4, 473–484.

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

Compaq Computer Corp. 1999. *Alpha 21264 Microprocessor Hardware Reference Manual*.

Doyle, B., Arghavani, R., Barlage, D., Datta, S., Doczy, M., Kavalieros, J., Murthy, A., and Chau, R. 2002. Transistor elements for 30 nm physical gate lengths and beyond. *Intel Technol. J. 6*, 2 (May), 42–54.

Dropsho, S., Kursun, V., Albonesi, D. H., Dwarkadas, S., and Friedman, E. G. 2002. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)* (Istanbul, Turkey), 321–332.

Feremans, C., Labbé, M., and Laporte, G. 2003. Generalized network design problems. *Eur. J. Oper. Res. 148*, 1–13.

Gonzalez, R. E. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro. 20*, 2, 60–70.

Horowitz, M., Indermaur, T., and Gonzalez, R. 1994. Low-Power digital design. In *Proceedings of the IEEE Symposium on Low Power Electronics* (San Diego, CA), 8–11.

Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H., and Bose, P. 2004. Microarchitectural techniques for power gating of execution units. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)* (Newport Beach, CA), 32–37.

Ip, H., Low, J., Cheung, P. Y. K., Constantinides, G. A., Luk, W., Seng, S. P., and Metzgen, P. 2002. Strassen's matrix multiplication for customisable processors. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)* (Hong Kong), 453–456.

Jones, R. 2004. Modeling and design techniques reduce 90 nm power. *EE Times*. http://www.eetimes.com/showArticle.jhtml?articleID=26806450.

Kao, J. T. and Chandrakasan, A. P. 2000. Dual-Threshold voltage techniques for low-power digital circuits. *IEEE J. Solid-State Circ. 35*, 7, 1009–1018.

Karnik, T., Borkar, S., and De, V. 2002. Sub-90nm technologies—Challenges and opportunities for CAD. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)* (San Jose, CA), 203–206.

Kim, N. S., Austin, T., Blaauw, D., Mudge, T., Flautner, K., Hu, J. S., Irwin, M. J., Kandemir, M., and Narayanan, V. 2003. Leakage current: Moore's law meets static power. *IEEE Comput. 36*, 12, 68–75.

Koster, A. M., van Hoesel, S. P., and Kolen, A. W. 1998. The partial constraint satisfaction problem: Facets and lifting theorems. *Oper. Res. Lett. 23*, 89–97.

Lee, C., Lee, J. K., Hwang, T.-T., and Tsai, S.-C. 2003. Compiler optimizations on VLIW instruction scheduling for low power. *ACM Trans. Des. Autom. Electron. Syst. 8*, 2, 252–268.

Lee, M. T.-C., Tiwari, V., Malik, S., and Fujita, M. 1997. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans. Very Large Scale Integr. Syst. 5*, 1 (Mar.), 123–133.

Rele, S., Pande, S., Onder, S., and Gupta, R. 2002. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction (CC)* (Grenoble, France), 261–275.

Roy, K. and Prasad, S. C. 1992. SYCLOP: Synthesis of CMOS logic for low power applications. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA), 464–467.

Semiconductor Industry Assoc. 2004. International technology roadmap for semiconductors.

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

Stanford Compiler Group. 1995. *The SUIF Library*. Stanford Compiler Group, Stanford University.

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

Tiwari, V., Singh, D., Rajgopal, S., Mehta, G., Patel, R., and Baez, F. 1998. Reducing power in high-performance microprocessors. In *Proceedings of the Design Automaton Conference* (San Francisco, CA), 732–737.

Tiwari, V., Donnelly, R., Malik, S., and Gonzalez, R. 1997. Dynamic power management for microprocessors: A case study. In *Proceedings of the International Conference on VLSI Design* (Hyderabad, India), 185–192.

Tsutsui, H., Masuzaki, T., Izumi, T., Onoye, T., and Nakamura, Y. 2002. High speed JPEG2000 encoder by configurable processor. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)* (Singapore), 45–50.

Yang, H., Govindarajan, R., Gao, G. R., Cai, G., and Hu, Z. 2002. 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)* (Charlottesville, VA).

You, Y.-P., Lee, C., and Lee, J. K. 2006. Compilers for leakage power reduction. *ACM Trans. Des. Autom. of Electron. Syst. 11*, 1 (Jan.), 147–164.

You, Y.-P., Lee, C., and Lee, J. K. 2002. Compiler analysis and supports for leakage power reduction on microprocessors. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)* (Washington, DC), 63–73. Lecture Notes in Computer Science, vol. 2481, Springer.

Zhang, W., Hu, J. S., Degalahal, V., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. 2004. Reducing instruction cache energy consumption using a compiler-based strategy. *ACM Trans. Architect. Code Optimi. 1*, 1 (Mar.), 3–33.

Zhang, W., Kandemir, M. T., Vijaykrishnan, N., Irwin, M. J., and De, V. 2003. Compiler support for reducing leakage energy consumption. In *Proceedings of the 6th Design Automation and Test in Europe Conference (DATE)* (Messe Munich, Germany), 1146–1147.

Zivojnovic, V., Martinez, J., Schlager, C., and Meyr, H. 1994. DSPstone: A DSP-Oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT)* (Dallas, TX), 715–720.

Received October 2006; revised May 2007; accepted May 2007