Compiler-Directed Fine Grain Power Gating for Leakage Power Reduction in Microprocessor Functional Units

Toshiya Komoda
Research Center for Advanced Science and Technology, The University of Tokyo
komoda@hal.rcast.u-tokyo.ac.jp

Hiroshi Sasaki
Research Center for Advanced Science and Technology, The University of Tokyo
sasaki@hal.rcast.u-tokyo.ac.jp

Masaaki Kondo
Graduate School of Information Systems, The University of Electro-Communications
kondo@is.uec.ac.jp

Hiroshi Nakamura
Research Center for Advanced Science and Technology, The University of Tokyo
nakamura@hal.rcast.u-tokyo.ac.jp

ABSTRACT

As semiconductor technology scales down, leakage-power becomes dominant in the total power consumption of LSI chips. We propose a compiler technique to turn off functional units that are expected to be idle for long periods of time for reducing leakage-power using fine grain power gating technique. Also, we propose a hybrid technique which combines a compiler and hardware based technique to maximize the chance of power gating. The results of experiments show that our proposed technique can reduce leakage-power of functional units for a wide range of break even time (BET) and applications.

1. INTRODUCTION

Nowadays, reducing the power consumption of microprocessors is a very important issue not only for embedded systems, but also for general purpose and high-end processors. Therefore, many techniques to reduce the power consumption have been developed so far. Generally, power consumption is consisted of two main parts, the dynamic power consumption, and the static (leakage) power consumption. Decades ago, leakage power consumption was negligible and the main concern was how to reduce the dynamic power consumption. However, as the semiconductor process size got smaller and smaller, the leakage power consumption became relatively large and now it is comparable to the dynamic power consumption.

Many embedded processors provide the mechanism to reduce leakage energy consumption in the form of *sleep* mode. In such a system, the processor core goes into the sleep mode when the operating system detects a long idle period. These techniques are effective and widely used to reduce the leakage when the entire processor core is in idle state. However, larger leakage power consumption means that the leakage can be the dominant factor of power consumption even when the processor core runs the application. Considering this situation, we need the technique to reduce the *run-time* leakage power consumption in addition to the techniques applicable to long idle period. Run-time leakage power reduction techniques have focused primarily on caches, which occupy a large area on the processor die [5, 9]. On the other hand, it is important to reduce the leakage power consumed by functional units for reduce total leakage power consumption of processors because the transistors in them are said to consume more leakage power than those in other part of processors [3]. Recently many approaches has been investigated to reduce the leakage power consumption of functional units [8, 11, 15].

Among several circuit techniques which reduce the leakage power, power gating is known as one of the most useful techniques because it can reduce most of the leakage with small energy and performance overhead. Power gating reduce the leakage by cutting off the supply voltage of the target circuit components, making them "asleep". The energy overhead is caused by turning on/off the high threshold voltage power switch transistors to cut off the supply voltage. If the leakage reduction is less than the overhead, we cannot gain energy savings. Therefore, the mode control strategy for power-gating should be carefully designed with taking the energy overhead into account.

There have been several work which attacked on how to detect the idle period either by software or hardware, and they successfully detect the idle periods by simple time based techniques or compiler analysis. However, there are applications in which the existing techniques are not effective. In this paper, we propose a compiler based method which reduce run-time leakage power consumption of the functional units of the processor by fine grain power gating with precise idle time prediction. The proposed method can detect the short idle period derived from the instruction sequence. We applied an interprocedural analysis in the proposed technique to precisely predict the idle periods. We also propose a hybrid power gating control method which combines the compiler based method and the hardware based method. Through the experiment, we show that the proposed method is effective for leakage energy reduction of functional units in a wide range of applications.

The rest of the paper is organized as follows. Section 2 describes the power gating technique and the architectural support for our proposed technique. Section 3 gives the details of the proposed compiler technique and the hybrid technique. Experimental results of the proposed techniques will be presented in Section 4. Section 5 describes related work and we conclude the paper in Section 6.

2. KEY TECHNOLOGIES

This section describes (2.1) power gating, which is the key circuit level technique for the proposed compiler based leakage reduction method and (2.2) the target architecture and the architectural support which enables the compiler to control power gating.

2.1 Leakage Reduction by Power Gating

Power gating with the MTCMOS technology is a well-known technique to reduce leakage power. In MTCMOS, power supply to logic blocks or units, which consist of low-Vth transistors, is gated by high-Vth power switch transistors. This puts power gated units into sleep mode and no operation can be performed in these units.

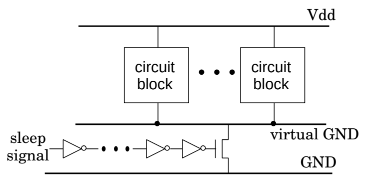


Figure 1: Overview of power gating.

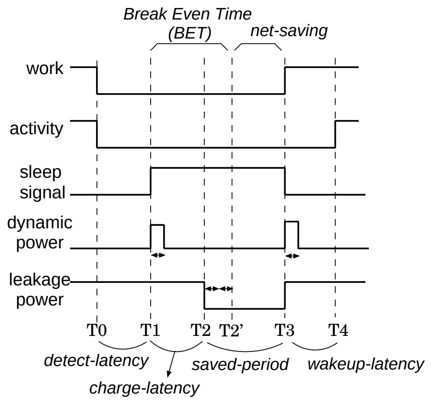


Figure 2: Overhead for mode transition.

The power switch transistors are inserted between a logic block and the Vdd line or between a logic block and the ground line, or both of them.

Figure 1 illustrates an overview of power gating when the power switches are inserted between the logic block and the ground line. When the sleep signal is asserted, the power switch transistors between the ground line and the virtual ground (VGND) line becomes off, and consequently power is gated and leakage current is suppressed. The mode transition incurs the time and energy overheads due to the sleep signal propagation, power switch driving, and discharging the electrical charge which is stored in parasitic capacitance on VGND line in the sleep mode.

Figure 2 shows the activity and power consumption of the unit which is the target of power gating. There becomes no workload to be done on the target unit at time T0 and becomes inactive. The sleep signal is asserted by detecting the idle period and the unit becomes power gated at T1. Even if the unit is in the sleep mode, the leakage power cannot be saved right after the time T1 since the leakage current flows to the VGND line until it is fully charged. Then after time T2, leakage power can be saved. The workload arrives at T3 and the sleep signal is deasserted to wake up the unit. The restart of the execution is delayed to T4 since the unit needs to wait for discharging of parasitic capacitance on the VGND line.

In the figure, *wakeup-latency* indicates the delay for restarting the execution when the power switches become on. This is the "performance overhead" caused by power gating. Even though there exists no workload from time T0 to T3, the period in which leakage energy is saved is only from T2 to T3 denoted as *saved-period*. This is due to the *detect-latency* which is the required time to detect the idle period for power-gating, and *discharge-latency* which is the interval between the time the power-switches turn off and the time the unit actually starts saving the leakage power. Moreover, the mode transition introduces dynamic power dissipation which is the "energy overhead" in power gating. Therefore, actual energy reduction denoted as *net-saving* is the amount of energy equivalent to leakage saving for *saved-period* minus the dynamic power overhead.

The period in which leakage energy savings balances with the dynamic power overhead is called the *break even time (BET)*. Once the unit switches to the sleep mode, the period in the sleep mode should be longer than BET, otherwise the energy consumption increases. Therefore, the mode control strategy for power gating should be carefully designed with taking BET into account. Note that BET depends on the semi-conductor process technology, temperature of the target unit, the structure of the unit, etc.

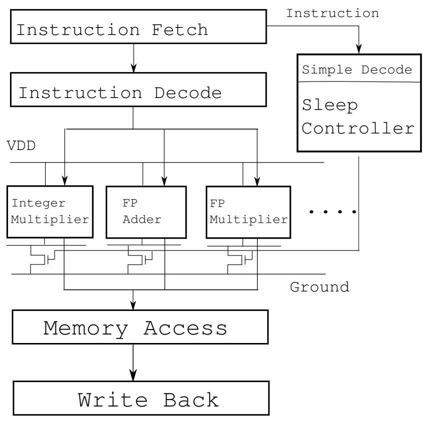


Figure 3: Target architecture with power gating control.

2.2 Architectural Support

The target architecture is a low-end processor for an embedded system which is a single issue, in-order processor with blocking caches and supports power gating at the functional unit level as shown in figure3. The sleep controller in the figure is the logic which determines and sends the sleep/wakeup signal to the target functional unit. The wakeup-latency of the power gating which we described in the previous subsection is reported that it can be controlled to be as few nanoseconds in 90 nm CMOS technology by optimizing the number of power switches to be inserted per each target unit [13]. This indicates that the wakeup-latency can be completely hidden by waking up the functional unit at the fetch stage when we assume a processor slower than 200 MHz (which is quite common in the embedded area) because there is at least two cycles (10 ns for 200 MHz processor) from the decode stage to the execution stage. From this insight, we assume that the functional units are waken up by the hardware when the instructions which use them are fetched. Thus, we do not have to wake up the functional units explicitly by software and do not consider about the performance overhead of power gating in this work.

As similar to the previous proposed compiler based power gating techniques [11, 12, 14, 15], we support instructions that allow the compiler to keep functional units to be power on or turn them to be power off after the execution. Our solution is to add a sleep bit to each instruction. The sleep bit is a suffix which specifies whether to "keep on" or "switch off" the functional units used by the instruction. For example, an integer multiplier instruction with a sleep bit "off" turns the integer multiplier off after its execution. We have implemented the sleep-bit to the MIPS ISA which have enough encoding space to allow us to implement the sleep-bit without increasing the instruction bit-width.

3. PROPOSED COMPILER BASED METHOD

3.1 Overview of the Compiler-Assisted Power Gating Technique

The proposed technique analyzes the assembly code of a program and predicts the idle time of each functional unit after execution. According to this information, each functional unit is selected to be kept on or turned off after execution. More precisely, if the predicted idle time of an instruction is longer than the target BET, the compiler sets the sleep-bit "switch off" so that the target functional unit sleeps after the execution. Note that we do not have to wake up the functional units by instructions because they are known to be used when the instructions are fetched and will be waken up by the sleep controller shown in Figure 3. Therefore, our goal in this section is to determine when to turn the functional units off by compiler analysis.

The advantage of the proposed technique is that there is no need to add an additional hardware for prediction because the prediction of the idle length is made statically during the compile time. Because the BET is predicted to become shorter in the future [3], applying power gating in a fine grain time scale becomes more effective. However, in order to apply power gating effectively, we have to precisely predict the idle time of the target units as mentioned in 2.1. When we try to predict this idle time by an additional hardware, it has to consume additional power throughout the execution as it must be active all along, and it cannot be ignored. From this point of view, the proposed technique is a promising approach to apply fine grain power gating.

Compiler analysis is effective for short stalls (such as a few to tens of cycles) determined by the instruction sequences. On the other hand, long stalls are often determined by the long latencies such as cache misses which occurs dynamically. These idle periods can be easily detected by hardware and also be effectively utilized for power gating by just simply sending sleep signals to functional units although they are impossible to predict by the compiler. From this insight, we propose a hybrid technique which combines the compiler assisted power gating and the cache miss based power gating together. By using both techniques, it becomes possible to capture both statically and dynamically determined idle periods very effectively.

Another reason of the long idle period comes from the rare use of the target unit. These idle periods are difficult to predict by the compiler without doing the analysis over the procedure calls. Thus, we also propose a technique to analyze the program in a global manner by focusing not only inside the basic block or the procedure but also by analyzing over the procedure calls. By the proposed analysis, we will be able to predict a long idle time which is a great chance for power gating. We will describe the analysis in detail in the next section.

3.2 Compiler Analysis in Detail

3.2.1 Overview

In this section, we describe the compiler analysis method to predict the idle time of functional units. We propose a method analogous to data-flow analysis [1], which is conventionally used in compiler optimization techniques. In this method, we analyze a *control flow graph (CFG)*, and obtain the expected idle time for each functional unit. Figure 4 shows an example of the CFG. Each node except for e represents a single instruction in the code. Node e is the exit node, which represents the exit point of the procedure. If we assume that we want to analyze the idle time of the multiplier, we count the expected number of nodes which lie between the target multiply instruction and a succeeding multiply instruction for each multiply instruction in the CFG. Assuming each instruction is executed in a single cycle, the above average number of nodes gives the expected idle cycles of the multiplier after the multiply instruction is executed. Although this assumption that each instruction is executed in a single cycle is an ideal case in our assumed architecture, it is reasonable because the situation is true when the instruction is executed without any stalls in the pipeline. Other functional unit's idle periods are analyze in the same fashion.

Usually, there are many procedure calls and loop structures in a typical programs' source code. The accuracy of the expected usage interval of functional units can be strongly affected by them. Therefore, we need to analyze beyond branches or procedure calls to use global information for precise prediction. We construct a *call graph (CG)*, which illustrates the relation between procedure calls for interprocedural analysis.

3.2.2 Intraprocedural Analysis

First of all, we describe the analysis of the procedure which has no procedure call in it. The basic framework is similar to a data flow analysis scheme [1]. However, we define real number variables (RNV), which express the expected usage interval of functional units, for each node in the CFG instead of typical data-flow values such as reaching definitions. In addition, a resource-utilization table is adopted to give the resource requirement for each instruction.

This analysis depends on information which is computed in the reverse order of the control flow in a program because we want to know where the following instruction next use the target unit.

First, we define RNV for the nodes in a CFG.

$$IN_D[s], IN_P[s], OUT_D[s], OUT_P[s]. \tag{1}$$

$OUT_D[s]$ expresses the expected number of instructions between the node s and the next instruction which uses the target functional unit. Here the target functional unit means the functional unit which we want to analyze. Therefore, $OUT_D[s]$ indicates the predicted idle time of the functional unit with the assumption that every instruction is executed in a single cycle. $IN_D[s]$ represents the same meaning value as $OUT_D[s]$ defined for the point right before the node s. $OUT_P[s]$ expresses the probability of reaching to the exit point of the procedure from the point after the node s without executing the instruction which use the target unit. For example, if we assume the branch probability of the instruction a as $\frac{1}{2}$ in Figure 4, $OUT_P[a]$ will be $\frac{1}{2}$. $IN_P[s]$ is the same meaning value as $OUT_P[s]$ defined for the point right before node s.

Next, we give the data-flow equation, which gives the constraint between the variables of nodes. For preparation, we define two constant values $T_D[s]$ and $T_P[s]$. These variables are defined for each node in the CFG, and their values are determined whether the node uses the target functional unit.

$$T_D[s] = \begin{cases} 0 & (\text{if } s \text{ uses the target unit.}) \\ 1 & (\text{otherwise}) \end{cases} \tag{2}$$

$$T_P[s] = \begin{cases} 0 & (\text{if } s \text{ uses the target unit.}) \\ 1 & (\text{otherwise}) \end{cases} \tag{3}$$

$T_D[s]$ expresses the expected idle time of the functional unit from the program point right before the node s to the program point right after the node s. $T_P[s]$ expresses the probability of reaching to the program point right after the node s from the program point right before the node s without executing the instruction which uses the target functional unit. In the intraprocedural analysis of the procedure which has no procedure call, these two values seem to be trivial because there is always only one instruction between the program point right before the node s and the program point right after the node s. However, there can be several instructions between the two program point right before and after the node s in the interprocedural analysis, which we describe later, because of the presence of the procedure call instructions.

By using these values, the data flow equation are given as the following equations.

$$IN_D[s] = T_P[s]OUT_D[s] + T_D[s]$$
$$IN_P[s] = T_P[s]OUT_P[s] \tag{4}$$

$$OUT_D[s] = \begin{cases} q[s]*IN_D[s_{suc1}]+(1-q[s])*IN_D[s_{suc2}] \\ (\text{if } s \text{ is branch.}) \\ IN_D[s_{suc}] \\ (\text{otherwise}) \end{cases} \tag{5}$$

$$OUT_P[s] = \begin{cases} q[s]*IN_P[s_{suc1}]+(1-q[s])*IN_P[s_{suc2}] \\ (\text{if } s \text{ is branch.}) \\ IN_P[s_{suc}] \\ (\text{otherwise}) \end{cases} \tag{6}$$

s_{suc} indicates the following node after the node s when the node s is not a branch instruction. s_{suc1} and s_{suc2} indicate the following two nodes after the node s when the node s is a branch instruction. Moreover, $q[s]$ is the probability that the branch node s jumps to node s_{suc1}. The values $q[s]$s are control parameters and can be set for each branch node respectively. The $q[s]$ values can be obtained in several ways such as the dynamic profiling technique or static branch prediction techniques [1].

Finally, we give the initial and boundary values for the iterative calculation by equations (4)–(6). The initial values for each node s except the exit node in the CFG are given as below.

$$IN_D^{(0)}[s] = T_P[s] * T_D[s], IN_P^{(0)}[s] = T_P[s],$$
$$OUT_D^{(0)}[s] = 0, OUT_P^{(0)}[s] = 0. \tag{7}$$

Let s_{exit} be the exit node of a procedure in the CFG, the boundary values are given as below.

$$IN_D^{(b)}[s_{exit}] = 0, IN_P^{(b)}[s_{exit}] = 1. \tag{8}$$

Note that it is not clear whether the algorithm will terminate or not because it contains a while loop. We can prove that the data flow values converge to a single solution through the iterative calculation based on equations (4)–(6). We give an brief overview of the proof below. Fist of all, the one step of the iterative calculation can be written in a form of a matrix-vector product as below.

$$V^{(k+1)} = AV^{(k)} + B \tag{9}$$

Where V is the vector of the data flow values of the nodes in the CFG, A is the update matrix which is determined by the CFG structure and the branch probabilities, and B is the constant vector determined by the character of the nodes. According to the liner algebra, the iterative calculation based on the equation (9) will converge to a single solution if and only if the all eigenvalues of the matrix A are inside the unit circle on the complex plain. We can prove that all eigenvalues of the matrix A are inside the unit circle on the complex plain by using the general characteristics of the matrix A. Therefore, we can conclude that the iterative calculation based on the equation (9), which is equivalent to the original iterative calculation based on (4)–(6), will converge.

Let's look at an example of the iterative calculation. In the Figure 4, the nodes a, b, c, d, and e represent the instructions. a and d are branches, c is the instruction which uses the target unit, and e is the exit node of the procedure. First, we calculate the constant values T_D and T_P according to equations (2) and (3). The values are

[1]We used a fix value $\frac{1}{2}$ for $q[s]$ during the evaluation in Section 4

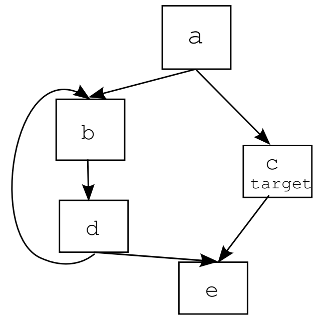


Figure 4: Example of a CFG (instruction C uses the target unit).

```
Intraprocedural Analysis
INPUT a control flow graph with resource utilizaton information.
OUTPUT expected idle cycles for each node in CFG.

$IN_P[s_{exit}] \leftarrow 0, IN_D[s_{exit}] \leftarrow 0$ {set boundary values according to (8)}
for each node $s \in$ CFG other than $s_{exit}$ do
  Set initial values of $OUT_P[s]$, $OUT_D[s]$ {set initial values according to (7)}
end for
while changes to any $OUT_P$ or $OUT_D$ occur do
  for each node $s \in$ CFG other than $s_{exit}$ do
    update $IN_D[s], IN_P[s], OUT_D[s], OUT_P[s]$ {calculation based on equations (4), (5),(6)}
  end for
end while
```

Figure 5: Pseudo code of the intraprocedural analysis algorithm.

$T_D[A] = T_P[A] = 1$, $T_D[B] = T_P[B] = 1$, $T_D[C] = T_P[C] = 0$, $T_D[D] = T_P[D] = 1$, and $T_D[E] = T_P[E] = 1$. The steps of iterative calculation with the above values and the equations (4)–(6) are shown in the table 6. The initial and boundary values are set by the equations (7) and (8). In the table, each cell represents $(IN_D, IN_P, OUT_D, OUT_P)$ for each node and each step. The lowest column gives the theoretical solution which can be obtained by this calculation. Note that the order of calculation is d, c, b, and a because the information is propagating backwards in our analysis, and we consider the 4 update operations to the nodes d, c, b, and a as 1 step. The node e do not have to be updated because it is the exit point, which gives the boundary values.

We need to perform an infinite time iteration in order to obtain the exact solution. However, for idle time prediction it is enough to obtain the integer value. Therefore, the iterative time will not be so long. The pseudo code of this algorithm is shown in Figure 5.

3.2.3 Interprocedural Analysis

In 3.2.2, we described the analysis for the procedure which does not include any procedure calls. Here, we describe how to handle the procedure calls in a procedure, which is often the case in a typical program. The simplest way to handle the call is to assume that the instruction which uses the target unit exists at the entrance of the called procedure. However, it is easy to imagine that it would result in a very conservative and rough prediction which is far from the precise value. Therefore, we need to apply the interprocedural analysis.

In the following, we use a pseudo instruction *jal* as the procedure call instruction for explanation. Now we describe the overview of the interprocedural analysis by the following three steps. (1) Preprocessing and analyzing the call graph (CG). (2) Seeking the transfer constant values of each procedure (described later). (3)

Node	e	d	c	b	a
step 0	(0, 1, 0, 0)	(1, 1, 0, 0)	(0, 0, 0, 0)	(1, 1, 0, 0)	(1, 1, 0, 0)
step 1	(0, 1, 0, 0)	(3/2, 1, 1/2, 1)	(0, 0, 0, 1)	(5/2, 1, 3/2 , 1)	(9/4, 1/2 , 5/4, 1/2)
step 2	(0, 1, 0, 0)	(9/4 , 1, 5/4, 1)	(0, 0, 0, 1)	(13/4, 1, 9/4, 1)	(21/8, 1/2, 13/8, 1/2)
⋮	⋮	⋮	⋮	⋮	⋮
result	(0, 1, 0, 0)	(3, 1, 2, 1)	(0, 0, 0, 1)	(4, 1, 3, 1)	(3, 1/2, 2, 1/2)

Figure 6: A concrete example of the iterative calculation.

Passing the information from the exit point of the program to each procedure. The overall process is similar to the Region-Based analysis [1]. We will give a brief view of the three steps.

In the interprocedural analysis, we analyze the procedures as described in 3.2.2. The main difference from the single procedure analysis is that we use the information of other procedures which has been analyzed for each procedure analysis. In step 2 and 3, the analysis order of procedures is important because we need to exchange information properly between procedures. Therefore, we first analyze the call graph in step 1 to determine the proper analysis order.

In step 1, we decompose the CG and make a new graph CG' where each node is a strongly connected component of the original CG. We decompose the CG into strongly connected components to handle the loops which are formed by recursive procedures or procedures which call themselves each other. Next, we give the post-order label to the nodes in the CG' through depth first searching. Figure 7 illustrates an example of post order labeling by the depth first searching. The large circles express the strongly connected components and the small circles in the large circles express the procedures. We analyze each strongly connected components in the labeled order in step 2, and analyze in the reverse order in step 3. We analyze the procedures which are in the same component as below. We analyze the procedures in the same component several times. For example, assume that two procedures A and B are in the same component. Then, we analyze the procedures as A-B-A-B. Here we set the number of iterative times to two, which is the number of procedures in the component, to assure that the information will be passed over the components.

Next, we describe how to exchange the information between procedures in step 2. For this purpose, we define the constant values T_D and T_P, which were defined for the *instructions* (nodes) in CFG, for the *procedures*. The values are given as bellow.

$$T_D[prc] := IN_D[s_{ent}], T_P[prc] := IN_P[s_{ent}] \qquad (10)$$

Where prc is the procedure, s_{ent} is the entrance instruction of procedure prc. We assume that the procedure prc has been already analyzed for getting the proper $IN_D[s_{ent}]$ and $IN_P[s_{ent}]$ values. We redefine the constant values T_D, T_P for the *instructions* (nodes) as below.

$$T_D[s] = \begin{cases} 0 & (\text{if } s \text{ uses a target unit.}) \\ T_D[prc_{called}] & (\text{if } s \text{ is "jal"}) \\ 1 & (\text{otherwise}). \end{cases} \qquad (11)$$

$$T_P[s] = \begin{cases} 0 & (\text{if } s \text{ uses a target unit.}) \\ T_P[prc_{called}] & (\text{if } s \text{ is "jal"}) \\ 1 & (\text{otherwise}) \end{cases} \qquad (12)$$

prc_{called} is the procedure which is called by the *jal* instruction.

In step 2, we analyze the procedures according to the order obtained in step 1. We use the definitions (11) and (12) instead of (2) and (3) used in the intraprocedural analysis. With the same initial

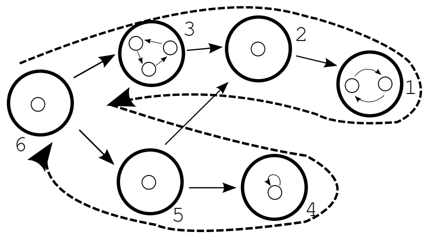


Figure 7: One example of post order labeling in a CG'.

and boundary values given by equations (7) and (8), we seek the solution through an iterative calculation in each procedure. From the result of them, we can obtain the constant values T_D and T_P of the procedures, which will be used in the analysis of other procedures which call the procedure.

Finally, we decide the expected usage interval of functional units at each point by passing the boundary values from the exit point of the program to the exit point of each procedure in reverse order of the control flow direction in step 3. Here, in order to analyze a certain procedure, we set the OUT_D values of the jal instruction s'_{call} which calls the procedure to the boundary value $IN_D[s_{exit}]$ instead of setting the boundary values given by (8). Note that the instruction s'_{call} generally exists in another procedure. Therefore, we have to analyze the procedures in the proper order to pass the boundary values to each procedure. With the passed boundary values, we can calculate the correct expected usage interval of the target unit which considers global information beyond procedures.

In summary, we seek the expected usage interval of functional units through passing the information between procedures appropriately. We analyze each procedure with iteratively using equations (10)– (12) and the data flow equations (4)– (6). The pseudo code of interprocedural analysis is shown in Figure 8.

3.3 Hybrid Technique

In this section, we propose a hybrid technique which combines the static prediction of the compiler based technique described in Section 3 and the hardware based technique. The hardware based technique is a simple cache miss detection based technique which is described in several articles [8, 13]. The advantage of this technique is that it almost requires no hardware modification to the microprocessor. In the following, we first present the hardware based technique in detail and then explain the characteristics and the advantage of our hybrid technique.

Generally, a cache miss is one of the main source of processor stalls, which offers many opportunities for power gating. From this insight, a simple cache miss based idle time prediction technique has been proposed and implemented as a real LSI chip [13]. In that technique, when a cache miss occurs, the desired data must be accessed from the external memory which is usually located off the processor chip. This accessing to the external memory takes much longer time than a cache access and require a processor to stall enough long time to benefit from power gating. Therefore, it would be effective to send sleep signals to all the functional units every time a cache miss occurs. The advantage of this cache miss

```
Interprocedural Analysis
INPUT a Call Graph $CG$ and control-flows graph $\in CG$.
OUTPUT expected idle cycles for each node in CFGs $\in CG$.

/*Preprocess a given call graph and determine an analyzing order.*/
Find strongly-connected components in $CG$.
Reduce the graph by treating a strongly-connected component as one node.
Denote the reduced graph as $CG'$.
Perform depth-first search on $CG'$, giving nodes post-order labels.

/*Perform the analysis on $CG'$ to compute transfer constant values.*/
for $P \in CG'$ according to the post-order labels do
  for $i = 0$ to the number of procedures in $P$ do
    for each procedure $prc \in P$ do
      do Intraprocedural analysis for $prc$ with using $T_D$, $T_P$ of called procedures.
    end for
  end for
end for

/*Perform the boundary value pass to each procedure*/
for $P \in CG'$ according to reverse of the post-order. do
  for $i = 0$ to the number of procedures in $P$ do
    for each procedure $prc \in P$ do
      do Intraprocedural analysis for $prc$ with using $OUT_D[s'_{call}]$ as $IN_D[s_{exit}]$.
    end for
  end for
end for
```

Figure 8: Pseudo code of the interprocedural analysis algorithm

based technique is that it requires minimal hardware modification. It only requires some logics to trigger the sending of the sleep signal when a cache miss occurs.

The cache miss based technique must be highly dependent on the cache architecture, the cache size, the replacement strategy, and whether it is blocking/non-blocking cache, and other aspects of the cache configuration. As our target architecture is a simple single pipeline in-order processor with a traditional 2-level blocking cache hierarchy, every cache miss causes a processor stall. In this paper, we evaluate the power gating technique triggered by an L2 cache miss.

The energy saving achieved by this technique is dependent on both the cache miss latencies, and the BET. If the cache miss latency is longer than the BET, the power gating triggered by a cache miss will always result in energy saving. If the cache miss latency is shorter than the BET, the energy saving depends on the executed instructions after the cache miss is resolved. When the target functional unit is unused for x cycles after the cache miss resolution, it will save energy if $L2\ miss\ latency + x > BET$, otherwise the functional unit will waste energy. As the memory access latency is longer than the BET which we are considering in this paper, it is a good decision to always use the L2 miss based technique.

The compiler analysis is effective for short stalls (a few to tens of cycles) because these are mostly decided by the instruction sequences. On the other hand, long stalls are often caused by cache misses which occur dynamically. As mentioned in the previous subsection, these idle periods can be easily detected by hardware and then power gating can be effectively applied. From this insight of capturing both short and long idle periods we propose a hybrid technique which combines the compiler assisted power gating and the cache miss based power gating together.

Figure9 illustrates the proposed hybrid technique. Here, we focus on the multiply instruction (mul) for the explanation. In this example, the compiler compiled the source code by assuming the

BET=20 cycle

```
   :
1. mul.on (keep on after execution)
   : 5 cycle
2. mul.off (switch off after execution)
   : 25 cycle
3. mul.on
4. : ← cache miss (switch off all functional units)
   : 100∼ cycle
   :
```

Figure 9: Power gating control by the hybrid technique

Table 1: Experimental Setup

ISA	PISA (MIPS like ISA)
Issue	in-order
Fetch & decode & issue & commit width	1
Branch prediction	Combined predictor (bimodal & 2-lev., 4K-entry)
Functional units	Int ALU: 1 Int Multiplier/Divider: 1 FP ALU: 1 FP Multiplier/Divider: 1
L1 I-cache	32KB, 32B line, 2way 1 cycle latency
L1 D-cache	32KB, 32B line, 2way 1 cycle latency
L2 unified cache	1MB, 128B line, 8way 6 cycle latency
Memory latency	100 cycle

BET as 20 cycles. First, the compiler analyzes the distance between the first mul (1. in the figure) and the second mul (2. in the figure), and the prediction was 5 cycles, so set the first mul as “mul.on” which keeps the multiplier on after execution. In the same way, it analyzes the distance between the second and the third mul, and set the second mul as “mul.off” because the distance was predicted as 25 cycles. The compiler sets the sleep-bit in a similar fashion for all the instructions. Also, when a cache miss occurs (4. in the figure) the processor stalls for a long time which the compiler can not predict, but the cache miss based power gating will let all the functional units asleep and save the leakage power.

4. EXPERIMENT

4.1 Experimental Setup

We used a cycle accurate processor simulator “sim-outorder” in the *SimpleScalar Tool Set* [2] for the evaluation. Table 1 shows the assumptions of the processor configuration. We evaluated the leakage energy consumption by using the proposed technique of three functional units, Integer multiplier, FP ALU, and FP multiplier.

We used nine programs from the SPEC CPU2000 floating point benchmark suite [7] with the *ref* input set, and two programs (*FFT* and *basicmath*) from MiBench [6] with the *large* input set. These programs selected because they use the above three functional units frequently so that the fine grain power gating control is needed in order to effectively reduce the leakage power. The programs were compiled for PISA (MIPS-like) instruction set architecture (ISA). The SPEC programs were compiled with the -O2 option and the MiBench programs were compiled with the -O3 option. We fast-forwarded one billion instructions and simulated two billion instructions for the SPEC benchmarks. The *FFT* were simulated from the beginning to completion, and the *basicmath* was simulated two billion instructions from the beginning. The number of BET cycles was varied as 5, 20, 40, 60, and 80 throughout the evaluation.

We evaluated the leakage energy consumption of each functional unit with the following power gating control methods.

- compiler-inter: Based on the compiler based analysis described in Section 3.
- compiler-intra: Based on the compiler based analysis without interprocedural analysis. The effect of procedure calls were conservatively estimated.
- L2: Based on L2 cache misses.
- hybrid: Hybrid method combining the compiler-inter and L2 method.
- best compiler: The maximum leakage energy reduction achievable by the compiler method if we can perfectly predict the idle time at compile time.

"Best compiler" shows the maximum leakage reduction through power gating control by the compiler with sleep bits but does not sleep even when a L2 cache miss occurs. Therefore, "best compiler" does not always achieve the best energy reduction among all the control methods.

The leakage energy is modeled by the total active cycles of functional units and the power gated cycles. Note that we include the energy overhead of power gating by taking BET into account. Let L_{total} be the total leakage energy consumption, w be the total active cycles with a certain sleep control method, sn be the number of active-sleep mode transitions, L be the leakage energy consumption per cycle, and e_{OH} be the energy overhead invoked by a single sleep. We evaluate L_{total} with the below equation.

$$L_{total} = w \times L + sn \times e_{OH}. \qquad (13)$$

We get the values of w and sn from the simulator for each functional unit. In the above model, BET is expressed as below.

$$e_{OH}/L = BET. \qquad (14)$$

We need to know the exact values of e_{OH} and L which depend on the hardware implementation to compute the exact L_{total}, however, we are evaluating the relative total leakage consumption so we compute it by using BET which is the ratio of e_{OH} to L.

4.2 Experimental Results

Figure 10–Figure 15 illustrate the percentage of leakage energy saving of five power gating methods described in the previous subsection with BET = 20 and 80. The x-axis represents the benchmarks (five methods for each benchmark) and the y-axis represents the relative leakage energy consumption compared to that without power gating. In the compiler based method, we must assume a certain BET value to determine where to insert sleep bits at compile time. We assume that the BET in the running environment can be known at the compile time in this evaluation.

First we focus on the effect of compiler based methods.The compiler based method with interprocedural analysis (compiler-inter) outperforms the compiler based method without interprocedural analysis (compiler-intra) in all benchmarks and BETs except for the result of *mgrid* (FP ALU and FP multiplier, BET=80). The improvements of interprocedural analysis is large especially in the results of BET=20, FP multiplier, *wupwise*, or BET=80, Integer Multiplier, *FFT*. Without interprocedural analysis (compiler-intra), the idle time prediction tends to fall into very conservative one and we lose many chances to trigger power gating like the case of BET=20, Integer multiplier, *equake*. The compiler based method without interprocedural analysis achieves 52% and 11% leakage reduction on average at BET=20 and BET=80, respectively. On the other hand, the compiler based method with interprocedural analysis achieves 86% and 54% leakage reduction on average at BET=20 and BET=80, respectively.

Table 2: The cache miss stall percentage to the total idle time

benchmark	%
art	0.80
ammp	0.73
swim	0.34
applu	0.24
equake	0.19
mgrid	0.18
mesa	0.11
wupwise	0.07
apsi	0.05
basicmath	0.05
FFT	0.02

Compiler based method with interprocedural analysis (compiler-inter) often achieves higher energy saving rate than the L2 miss based method. However, the L2 miss based method outperforms the compiler based method with interprocedural analysis in some case, for example FP ALU in *swim*, *mgrid*, and *art* or FP multiplier in applu at BET=80. This is primarily because of the characteristics of these benchmarks. One main reason of this is that L2 cache misses occur frequently in these benchmarks and the stalls caused by L2 misses occupy a large portion of the total idle time. Table 2 shows the percentage of the idle time which is caused by cache misses. The benchmarks which showed good results by applying L2 miss based method which are *swim*, *mgrid*, *applu*, and *art* shows high percentage of idle time due to cache misses.

Figure 16 and Figure 17 illustrates the cumulative percentage of the idle time length of FP ALU in *art* and *equake* respectively. In Figure 16, two large jumps are observed around 120 (single) and 220 (two consecutive) cycles due to cache misses. L2 miss based method can effectively detect these idle period so that it achieves much leakage reduction in *art*. However in *equake*, roughly 60% of the intervals were less than 100. Therefore, the compiler based method achieves better energy savings at smaller BET as shown in Figure 10. When the BET gets bigger, the L2 miss based method becomes advantageous and the energy savings become better than the compiler based method as shown in Figure11.

Though L2 cache misses occur frequently in *ammp*, the compiler based method achieves power reduction comparable to L2 based method. The reason is that the target functional units are used rarely in *ammp* so that there are long idle periods derived from both instruction sequences and cache misses. Since our compiler method can analyze the instruction code beyond procedures, the compiler based method with interprocedural analysis can detect these idle periods and achieve large energy savings in *ammp*.

The hybrid method which combines the compiler based method and the L2 miss based method outperforms other methods in all the cases except for three situations when BET=80 (FP ALU of *mgrid* and *swim* and FP multiplier of *applu*). It achieves 87% and 76% leakage energy reduction on average at BET=20 and BET=80, respectively. The results show that the compiler based method and the L2 miss based method can cooperate and reduce the leakage energy very effectively.

4.3 Sensitivity to Dynamic BET Fluctuations

Considering a more real situation, the value of BET of functional units changes dynamically mainly due to the fluctuations in temperature during run-time. Therefore, we investigate the sensitivity to the dynamic BET change of the compiler based method.

Figure 18 and Figure 19 illustrate the evaluation results of the FP ALU in *equake*, and Integer multiplier in *applu* respectively. The x-axis represents the BET, and the y-axis represents the relative energy consumption which is the same in the previous figures. In this experiment, the binary is generated by the compiler assuming

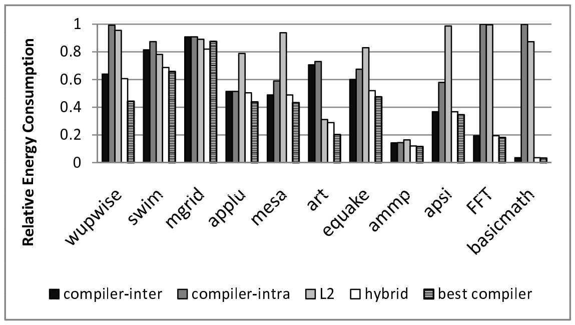


Figure 10: FP ALU, BET = 20

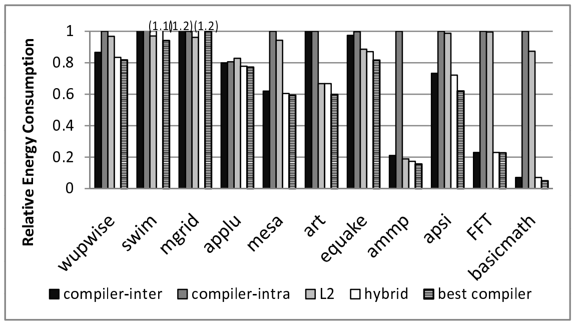


Figure 11: FP ALU, BET = 80

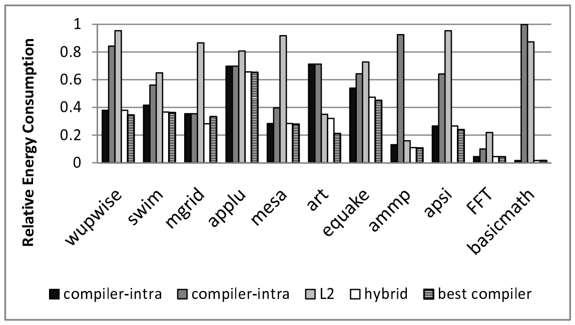


Figure 12: FP multiplier, BET = 20

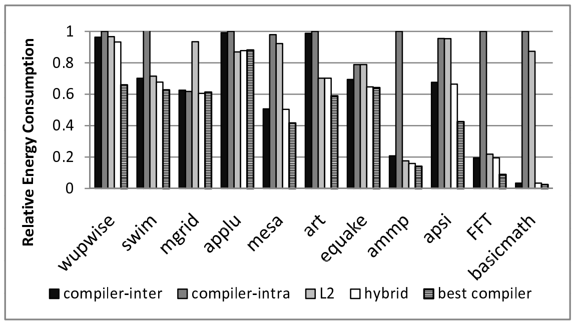


Figure 13: FP multiplier, BET = 80

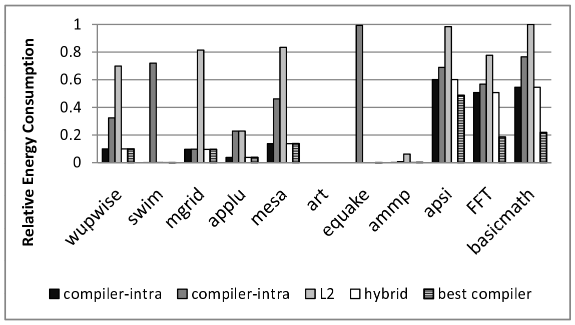


Figure 14: Int multiplier, BET = 20

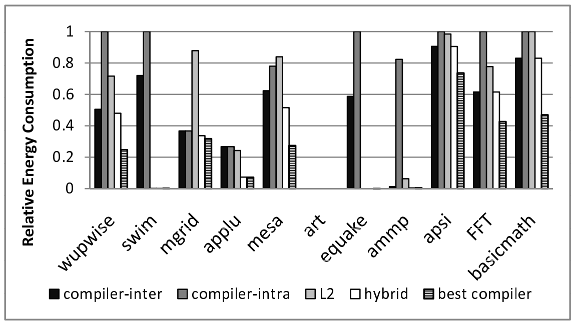
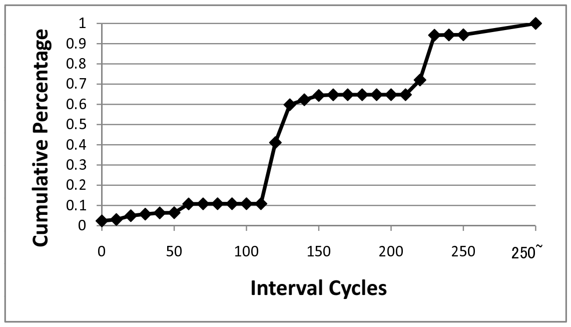


Figure 15: Int multiplier, BET = 80



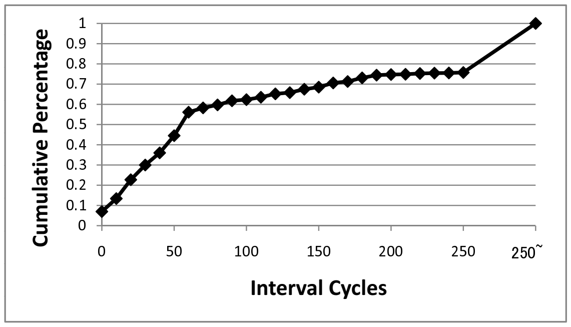


Figure 16: Cumulative Percentage of idle period. art, FP ALU. **Figure 17: Cumulative Percentage of idle period. equake, FP ALU.**

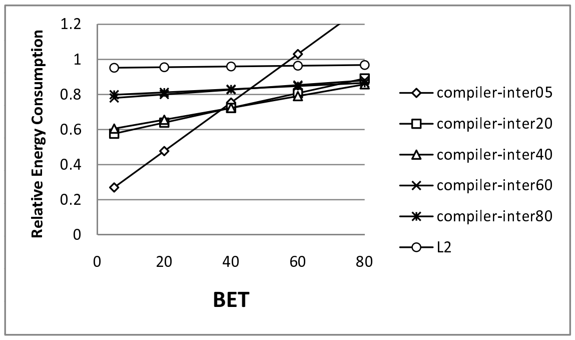


Figure 18: BET Sensitivity. equake, FP ALU

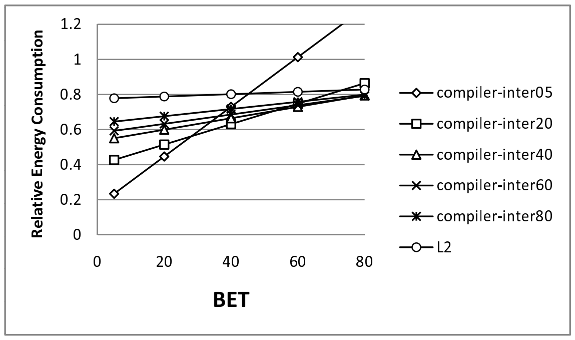


Figure 19: BET Sensitivity. applu, FP ALU

a single BET (either 5, 20, 40, 60, or 80) which corresponds to each lines in the figures. For example, by focusing on the line of compiler-inter 20 (compiled by assuming a BET=20), we can see how much leakage power reduction can be achieved when BET changes. We also show the result of L2 miss based method in the figures.

In summary, the compiler based method which assumed a small BET (compiler-inter 5 and compiler-inter 20) tends to achieve larger energy savings in the small BET region because aggressive power gating benefits from short idle periods of functional units. However, because of the frequent mode transitions, the larger the BET, the larger the energy overhead of power gating becomes very quickly. Therefore, the compiler based method with small BETs loses its energy saving effect in the large BET region (Figure 18, 19). On the other hand, the compiler based method which assumed a large BET (compiler-inter 60 an compiler-inter 80) and the L2 miss based method are less sensitive to the BET fluctuations because they cause only a small number of mode transitions. Therefore, their results are nearly horizontal in both Figure 18 and Figure 19. However, these methods can not achieve large energy savings in the small BET region. A more optimal power gating control method which can effectively reduce leakage power by adjusting to the dynamic BET fluctuations is left for future work.

5. RELATED WORK

A lot of research effort has been directed toward reducing static power consumption so far. Architectural-level leakage power reduction techniques have focused primarily on SRAMs (caches and buffers). Simple time-out based techniques for power gating caches were developed by many researchers [5, 9]. Beside the leakage reduction techniques for caches, researchers have studied to reduce leakage for logic blocks so far [4, 8, 11].

Dropsho et al. have explored analytical models to determine the sleep-mode activation policies for the integer functional units using a dual-threshold domino logic circuits [4]. Hu et al. explored the potential of similar architectural techniques to reduce leakage by power gating functional units [8]. Rele et al. presented a compiler based approach for power gating in superscalar processors [11]. You et al. proposed the compiler analysis framework for estimating the component activities and sleep instruction scheduling policies [15]. Nagpal et al. proposed a compiler instruction scheduling algorithm that assists the hardware based scheme in the context of VLIW and clustered VLIW architectures [10]. Roy et al. tried to predict the idle time of functional units in embedded microprocessors, analyzing loop structures of embedded applications [12]. Talli et al. used dynamic profile information to identify the functional unit idle periods [14].

Our approach calculates the expected idle times for each program points which is similar in the work by You et al. [15]. Our main contribution is that we apply interprocedural analysis. Also, we effectively combine the static method and the dynamic method for further leakage reduction.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a fine grain power gating control method based on a static analysis by the compiler to effectively reduce the leakage power of microprocessor functional units. The compiler analyze the distance between nodes in the CFG, and predict the expected usage interval of the target unit. Based on this prediction, the compiler inserts a sleep-bit to keep functional units to be power on or turn them to be power off after the execution. Also, We proposed and evaluated a hybrid method which combines the compiler based method and a simple and effective hardware based method which has no negative impact on the advantage of compiler based.

In the future, we intend to extend our proposed method in order to adjust to the dynamic BET fluctuations due to the temperature fluctuations at run-time.

Acknowledgment

This work was supported in part by Japan Science and Technology Agency as a CREST research program entitled “Innovative Power Control for Ultra Low-Power and High-Performance System LSIs” and by Japan Society for the Promotion of Science Grant-in-Aid for Scientific Research (A) No.18200002.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[3] J. A. Butts and G. S. Sohi. A static power model for architects. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201, New York, NY, USA, 2000. ACM.

[4] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman. Managing static leakage energy in microprocessor functional units. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 321–332, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[5] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage

power. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[7] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[8] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 32–37, New York, NY, USA, 2004. ACM.

[9] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251, New York, NY, USA, 2001. ACM.

[10] R. Nagpal and Y. N. Srikant. Compiler-assisted leakage energy optimization for clustered vliw architectures. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 233–241, New York, NY, USA, 2006. ACM.

[11] S. Rele, S. Pande, S. Önder, and R. Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 261–275, London, UK, 2002. Springer-Verlag.

[12] S. Roy, S. Katkoori, and N. Ranganathan. A compiler based leakage reduction technique by power-gating functional units in embedded microprocessors. In *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, pages 215–220, Washington, DC, USA, 2007. IEEE Computer Society.

[13] N. Seki, L. Zhao, J. Kei, D. Ikebuchi, Y. Kojima, Y. Hasegawa, H. Amano, T. Kashima, S. Takeda, T. Shirai, M. Nakata, K. Usami, T. Sunata, J. Kanai, M. Kanai, M. Kondo, and H. Nakamura. A fine-grain dynamic sleep control scheme in mips r3000. In *ICCD '08: Proceedings of the 2008 international conference on computer design*, pages 612–617, Washington, DC, USA, 2008. IEEE Computer Society.

[14] S. Talli, R. Srinivasan, and J. Cook. Compiler-directed functional unit shutdown for microarchitecture power optimization. In *IPCCC '07: Proceedings of the 26th IEEE International performance computing and communications conference*, pages 372–379, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

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