Compiler-Directed Leakage Reduction in Embedded Microprocessors

Soumyaroop Roy, Nagarajan Ranganathan, and Srinivas Katkoori
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
{sroy, ranganat, katkoori}@cse.usf.edu

Abstract— **Compiler-directed power gating is an approach in which *sleep instructions* are inserted appropriately at compile time into the application code to selectively deactivate the functional units in microprocessors during their idle periods to reduce power dissipation due to leakage. Although the effect of code transformations on dynamic and system power has been investigated and reported in the literature, such a study is lacking in the context of power gating. In this paper, we investigate and report how the leakage savings in both integer and floating point units can be improved using machine-dependent and independent optimizations in a compiler-directed power gating framework. In our study, it is ensured that power gating is applied only when the leakage savings are considerably more than the various overheads incurred in its implementation. The target embedded processor is modeled on the ARMv4 architecture, which is modified to support the power gating of its arithmetic functional units. For experimentation, GCC is used as the compiler infrastructure and Simplescalar-ARM is used as the detailed architectural simulator for reporting power and performance metrics for embedded applications belonging to the MiBench and MediaBench benchmark suites. Experimental results suggest that the additional savings in leakage energy due to one or more of the optimizations may vary largely depending on the benchmark. Moreover, the overhead of sleep instructions can be reduced by up to 50 times by performing procedure inlining.**

I. Introduction

With the proliferation of ubiquitous and portable computing devices, low power design of CMOS circuits has become an imperative consideration in ensuring long battery lives for such devices. These mobile devices range from simple microcontrollers in sensor networks to moderately sophisticated embedded microprocessors in smartphones. Till the last decade, the major contributor of the total power dissipation in CMOS circuits was dynamic power, which is due to the charging and discharging of gate capacitances during output switching. However, in the recent years, due to the sub-100nm feature sizes in fabrication technologies, leakage power, which is primarily due to the subthreshold leakage current between the source and the drain terminals of the transistor, has started to contribute significantly to the total power of CMOS circuits [1]. Therefore, developing techniques to reduce leakage power in CMOS circuits is generating tremendous interest in the low power research community these days.

Power gating [2] is a circuit-level technique that reduces the subthreshold leakage by cutting off the supply voltage to the circuit, thereby, putting it to *sleep*. To *wakeup* the circuit, the supply is simply restored to it, which brings it to the active operating mode. Power gating is considered to be one of the most useful techniques for leakage reduction because, at the cost of low energy and performance overheads, the savings in leakage achieved by power gating can be substantially larger than those achieved by any other technique. The overhead in the energy is as a result of charging and discharging of the gate capacitances of the circuit when it is turned on and off, respectively. If a circuit is brought into and out of its low-leakage mode too frequently, the energy overhead may become larger than the leakage savings achieved, thereby making power gating ineffective for leakage savings. Thus, adequate considerations about the overhead energy should be made while designing the controls to power gating to ensure that savings in leakage are achieved during the operation of the circuit.

Power gating can be applied in microprocessors to put its components to sleep during the periods when they are idle. Their controls can be designed either entirely at the microarchitectural level or in the form of special instructions (*power gating instructions*) that are embedded into the application code by the compiler, which when executed during program execution put those components in and out of low-leakage states. In the latter compiler-directed approach, static analyses of the application program are performed to identify program regions where functional units are expected to be idle and power gating instructions are inserted at the boundary of such regions. Thus, this approach is dependent on the compiler optimizations that are performed to generate the executable because they may alter the power gating opportunities of the components in a favorable or unfavorable way. This aspect of compiler-directed power gating was addressed for the first time in [3], in which four scalar integer optimizations were performed to improve the power gating opportunities of the integer multiplier. In this work, we present the results of a more comprehensive study investigating how the leakage savings in both integer and floating point units can be improved using machine-dependent and independent optimizations in GCC, which is a production compiler framework. We discuss these optimizations from the power gating perspective and verify their effectiveness with the help of simulations. The influence of a few high level compiler optimizations on dynamic power and energy consumption in microprocessors was studied by Kandemir et al. in [4], however, no such study has been performed for leakage power.

This paper is organized as follows. Section II presents the related work and Section III describes the architecture support for power gating and the technique for inserting power gating instructions into the code. Section IV discussses compiler optimization techniques that enhance opportunities for power gating. The experimental results and the conclusions are presented in Section V and Section VI, respectively.

II. Related Work

Power gating techniques at a purely microarchitectural level were first investigated for caches [5], [6]. Subsequently, Hu et al. presented a purely microarchitectural technique that was based on hardware-level branch prediction to control the sleep and wakeup of arithmetic functional units in [7]. In this paper, we simply use the term *functional units* to refer to the arithmetic functional units in the datapath of a microprocessor. Compiler-directed approaches with microarchitectural support have generated more interest in effectively employing power gating to reduce leakage power in idle functional units. Two of the early works based on compiler-directed techniques were presented by Rele et al. in [8], wherein they target reducing leakage in idle functional units in superscalar processors, and by You et al. in [9], wherein they apply dataflow analyses to inspect program regions for idleness of functional units. A brief study [10] on the impact of both input vector control and power gating was presented by Zhang et al. The need for an explicit wakeup instruction to activate the functional units was eliminated in the technique proposed by Roy et al. in [11], wherein the functional units were designed with multiple sleep transistors to facilitate a single cycle wakeup for those units. Thus, by the time an instruction reached the execution stage from the decode stage, the functional unit was already active to start a computation. Seki et al. and Komoda et al. further removed the need for an explicit sleep instruction by encoding the sleep information into the unused bits in the instruction format for each data processing instruction in [12] and [13], respectively.

III. Architecture and Compiler Support for Power Gating

The architecture model used in this work is based on the one proposed in [11] because of the following reasons:

1) The ARM architecture accounts for approximately 75% of the 16/32-bit embedded RISC processors in the world [14]. Since, this work investigates a compiler driven leakage reduction technique for embedded microprocessors, the choice of the ARM architecture seems most appropriate.
2) In [11], the ARM instruction set architecture (ISA) is extended with an explicit *sleep* instruction but there is no explicit *wakeup* instruction. A functional unit is automatically activated by the decode logic when an instruction requiring that unit is decoded.
3) The integer and floating point functional units in the target ARM processor are designed and characterized for latency and power using 70 nm PTM files.

The functional units that can be power gated are an integer multiplier, an FP adder, an FP multiplier, and an FP division and square root unit. The integer ALU is not power gated because it is the most frequently used functional unit in every application. Unlike in [11], however, the barrel shifter in this work is not considered for power gating because the ARM instruction set supports data and memory instructions in which one of the operands may be a register shifted by a constant integer [15]. During code optimizations, such instructions are generated very densely, thereby making the usage of the shifter much higher than the other functional units. The decode logic is extended to include a 4-bit register that controls the sleep transistors of the four functional units.

The assembly opcode for the sleep instruction is `slp`. It takes a 4-bit immediate integer as an argument that encodes the list of the functional units that are to be deactivated. The format of the machine code for the sleep instruction is chosen from the domain of exceptional opcodes described in the ARM reference manual. It has bits 7-0 assigned to `'F0'` and bits 31-20 to `'07F'`. Instruction bits 11-8 are reserved for the four-bit immediate integer argument that is passed to the sleep instruction. The assembler source code in Binutils [16] is modified to generate the machine code for the `slp` instruction and the source for `sim-ourorder` in Simplescalar architectural simulator [17] is modified to carry out the activation and deactivation of the functional units during the decode stage of the processor pipeline.

The approach for inserting power gating instructions during compilation is outlined in Algorithms 1-3. Algorithm 1 describes the technique of inserting sleep instructions within a procedure. The functional unit usage of all the basic blocks, loops and function are computed. In case of a procedure call instruction, if that call is to a procedure from the standard C math library, it is assumed that the instruction uses all the functional units. Otherwise, it is assumed that the instruction does not use any of the functional units. This approach, although conservative, is essential to handle

Algorithm 1 pgi-insert(f)

```
1: /* Gather functional usage information */
2: for each basic block $b \in$ CFG nodes do
3:     Note functional unit usage in $b$, $FU(b)$
4: end for
5: $FU(f) \leftarrow \cup FU(b), \forall b \in$ CFG nodes
6: Compute the loop hierarchy tree, $L$, in $f$
7: for each loop $l \in L$ do
8:     $FU(l) \leftarrow \cup FU(b), \forall b \in l$
9: end for
10: /* Insert power gating instructions */
11: for each region $r \in L$ in BFS order do
12:     if $FU(r) \subset FU(parent(r))$ then
13:         /* $r$ needs fewer functional units than $parent(r)$ */
14:         Insert sleep($FU(parent(r) - FU(r))$) at the entry of $r$
15:     end if
16: end for
```

standard library functions in the code. Then, the *loop tree* is traversed in postorder sequence for regions whose entries can be gated with sleep instructions. A *loop tree* is a rooted tree data structure, in which each node represents a loop in the procedure and the children of a node, l, represent the loops immediately contained inside l. The root of the loop tree represents the entire procedure. Algorithm 2 describes the expansion steps for a procedure, which involves performing all the intraprocedural optimizations, inserting power gating instructions, and generating its assembly code. The top level

Algorithm 2 expand(f)

```
1: Perform intraprocedural optimizations on $f$
2: pgi-insert ($f$)
3: Output the assembly code for $f$
```

driver that drives the entire compilation process is described in Algorithm 3 and is equipped to perform interprocedural analysis (IPA) and procedure inlining. If IPA is enabled, a call graph is constructed and procedure inlining is performed on the call graph. After that, the expansion of the procedures is performed in the postorder sequence of the vertices in the call graph. This is done to enable transfer of data from a callee to its caller, which may lead to better optimization opportunities during code generation of the caller. If IPA is

Algorithm 3 toplevel-driver()

```
1: if ipa-enabled then
2:     Construct the call graph
3:     Perform procedure inlining
4:     for each procedure $x \in$ call graph in postorder sequence do
5:         expand ($x$)
6:     end for
7: else
8:     for each procedure $x \in$ procedure parse sequence do
9:         expand ($x$)
10:    end for
11: end if
```

not enabled, the procedures are expanded in the order they are parsed from the source file.

It should be noted that the compiler optimization techniques discussed subsequently in this work reduce the functional unit usage in a program either by completely eliminating instructions that use those units (e.g., strength reduction) or by moving them to less frequently executed regions in the program (e.g., code motion). This results in generation of code that has more regions which do not use those units, thereby making them idle for longer periods of time. Therefore, although we perform our experiments using the technique for inserting power gating instructions decribed above, all compiler-directed power gating techniques should benefit from these optimization techniques.

IV. Performing compiler optimizations to improve power gating

In this section, we discuss how the effectiveness of a compiler-directed power gating approach may be improved with the help of compiler optimizations.

A. Dominator Optimizations

These optimizations use the dominance information of the control flow graph (CFG) for the procedure to perform various optimization tasks. Given two basic blocks X and Y, block X *dominates* block Y *if and only if* X is always executed before Y. The common dominator optimizations are *dead code elimination*, *constant and copy propagation*, and *common subexpression elimination*. *Dead code elimination* is an optimization technique that removes code that will not get executed during program runtime. *Constant propagation* and *Copy propagation* are techniques that substitute the occurence of a constant operand and a source variable in a chain of copy or move operations throughout the code, respectively. *Common subexpression elimination* (CSE) searches for instances of identical expressions and attempts to replace them with a single temporary holding the computed value. Its scope can be either local to a basic block (local CSE or just CSE) or across basic blocks (global CSE). Most of the optimizations in this category can be very effective in removing redundant arithmetic operations from the code, thereby, eliminating the need for functional units in various regions in the program.

B. Loop Optimizations

We discuss two optimizations in this category. The first is *code motion* or *code hoisting*, in which expressions that do not vary over different iterations of a loop are moved out of it. This may lead to one or more multiplication or division operations to be hoisted before the entry to the loop, thereby leaving the loop body devoid of that operation. In such a case, a sleep instruction, putting the appropriate functional units to sleep, may be inserted at the entry of the loop. The second is a family of optimizations that are performed on *induction variables*. An *induction variable* is a variable that gets increased or decreased by a constant value in every iteration of a loop or varies linearly with respect to another induction variable. *Strength reduction* is an induction variable optimization in which an iterated series of strong computations is replaced with an equivalent series of weaker computations. Although both the optimizations discussed above have traditionally been used and proven to be effective on integer instructions, if IEEE or ISO floating point (FP) precision rules are relaxed, these techniques can also be applied to FP instructions.

C. Machine Dependent Optimizations

These optimizations require a detailed knowledge of the machine architecture, like pipeline latency specifications, functional unit latencies, cache latencies, branch delay slot details, etc. One subcategory of these optimizations, known as *peephole optimizations*, includes techniques that perform code transformations on a small window of instructions. For example, in the C programming language, the expression $a[i]$, where a is an integer array on a 32-bit machine, translates to the memory address given by $a + 4 * i$. This is because the size of integer type data on a 32-bit machine is 4 bytes. If code generation is performed based on the above computation, a multiplication instruction will be introduced by the compiler prior to a load or store instruction for the array element. However, ARM also provides a complex addressing mode for its load and store instructions, in which the base address and the offset can be specified directly. Thus, the multiply instruction may be eliminated during this optimization. The other optimization in this category is *weak strength reduction*, in which multiplication and division operations in which one of the operands is a constant is replaced with a sequence of addition, subtraction, and shift instructions. For example, $a \cdot 119$ may be expressed as $a \cdot (16 + 1) \cdot (8 - 1)$. The latter expression can now be computed solely by using add, subtract, and left shift operations without the need for a multiply instruction. The optimized code is also likely to be faster than the unoptimized code because, while a multiply instruction takes multiple cycles (upto 16 cycles in ARMv4 architecture), the `add`, `sub`, and `asl` instructions each take just 1 cycle to execute. As it can be seen, the decision to carry out the above transformations requires accurate estimation of latencies of various instructions for the target machine [18], [19], which is only available at this stage of compilation.

D. Procedure Inlining

Procedure inlining is an interprocedural optimization that replaces the body of a callee procedure within the body of the caller procedure. From the perspective of power gating, procedure inlining has two distinct advantages:

1) Any modern static compilation flow features performing code generation for one procedure at a time. Therefore, procedure inlining improves the visibility of the program behavior for the compiler during code generation, thereby making the intraprocedural optimizations more effective.
2) The process of inserting sleep instructions is also performed at the procedure level and, therefore, procedure inlining may reduce the number of discrete regions at the entries of which sleep instructions are inserted. This, in turn, may reduce the number of redundant sleep instructions that are executed during runtime.

V. Experimental Setup and Results

For this work, we set up an ARM cross-compiler toolchain with GCC 4.2.1, Binutils 2.17, and Glibc 2.3.6 [16]. GCC is used as the compiler framework because the quality of code generated by GCC is generally much superior to that generated by most research compilers. Moreover, GCC has a rich library of compiler optimizations which can be explored comprehensively in this work. GCC uses three intermediate representations - GENERIC, GIMPLE, and RTL. The compiler front end parses the C source code and converts it into GENERIC representation. This is lowered into GIMPLE representation, where high level code transformations (procedure inlining, loop transformations, etc.) are performed. Lower level optimizations, including most of the machine dependent optimizations (instruction scheduling, peephole optimizations etc.), are performed at the RTL level. The control flow graph (CFG) information for a procedure is retained till late in the RTL optimization passes. Since insertion of power gating instructions into the code requires its control flow information, this pass is added as the final pass before the control flow graph for the procedure is purged and the procedure is described only as an instruction list.

A. Optimization Configurations

TABLE I
Optimization configurations

Optimization Label	Description
C0	Base configuration (only machine specific instruction scheduling is performed)
C1	C0 + Machine dependent peephole optimizations + Dominator optimizations
C2	C1 + Loop invariant code motion
C3	C2 + Induction variable optimizations

The optimizations in each of the configurations above may be performed with procedure inlining and floating point optimizations

Table I describes the various optimizations configurations defined for the purpose of experimentation in this work. C0 is the base configuration in which only machine specific instruction scheduling is performed. In C1, machine dependent peephole optimizations and all dominator optimizations (as enumerated in the table) are performed. In C2, loop invariant code motion is performed along with those in C1. Finally, in C3, induction variable optimizations are performed in addition to those performed in C2. All the intraprocedural optimizations above can also be performed after procedure inlining is performed. Along with that, the floating point arithmetic optimization flag `-ffast-math` can also be turned on during all the intraprocedural optimizations. This is particularly helpful for benchmarks that are floating point intensive.

B. Results

TABLE II
Description of Metrics

Metric label	Description
cyc	Number of simulation cycles
ovhd	Percentage of overhead cycles due to power gating, computed as 1 - (cyc without power gating/cyc with power gating)
slp	Number of sleep instructions decoded
bsy	Number of cycles for which a unit is busy
engy	leakage energy for a unit
sav	Percentage savings due to power gating, computed as 1 - (engy without power gating/engy with power gating)

We ran the experiments assuming a library of functional units which have breakeven periods of 500 cycles and sleep/wakeup latencies of 1 cycle. The leakage savings in sleep mode for the units are 50%, which indicates that the theoretical upper bound on the maximum savings is 50%. This is in agreement with the details of the functional unit library developed in [11]. Table II enumerates the metrics that are computed and reported in this section to demonstrate the effectiveness of power gating. Table III describes the processor configuration used during simulations. Experiments were performed on various benchmarks from MiBench [20] and MediaBench [21] suites.

TABLE III

ARM PROCESSOR CONFIGURATION [17]

Fetch Queue (instructions)	2
Fetch/Decode/Issue width	1
Branch Predictor	Not-taken
Functional Units	1 Integer Multiplier (16 cycles) 1 FP Adder (6 cycles) 1 FP Multiplier (10 cycles) 1 FP Div/Sqrt (19 cycles)
Instruction L1 Cache	16K, 32-way
Data L1 Cache	16K, 32-way
L2 Cache	None
Memory bus width	4-byte

TABLE IV

STATISTICS FOR *Susan Corners*

	cyc^1	Imul		Fdsq		slp^1
		sav	bsy^1	sav	bsy^1	
C0	1.00	6.10	1.00	9.34	1.00	1.00
C1^2	0.79	6.54	0.10	8.10	1.00	0.83
C2^2	0.81	9.11	0.12	19.78	0.17	8.02
C3^2	0.82	12.35	0.08	21.08	0.17	0.73

^{1} The numbers in this column are normalized with the number in C0

^{2} Performed with floating point optimizations

Susan Corners is an image processing program that detects corners in an image. In this benchmark, the integer multiplier is busy for 14% of the cycles in C0. The FP adder, multiplier and divider are busy for 0.8%, 1.4%, and 0.5% of the cycles in C0. When the optimizations are performed on this benchmark, marked improvements in leakage energy savings are observed for the integer multiplier and the FP divider (Table IV). In C1-C3, the number of busy cycles for the integer multiplier drops by 8X (0.12 vs. 1.0) to 12X (0.08 vs. 1.0) giving an increase in savings by 7% (6.54% vs. 6.1%) to 2X (12.35% vs. 6.1%). For the FP divider, the number of busy cycles drops by almost 6X (0.17 vs. 1.0), thereby increasing the savings by more than 2X (19.78% or more vs. 9.34%). As indicated in the table, floating point optimizations are performed in C1-C3. Interprocedural optimizations are not performed for this benchmark because almost the entire execution time is spent in one function that is called by the `main()` routine for corner detection. The performance overhead (ovhd) is between 0.4% and 0.6%.

Susan Edges is another image processing program (part of the same distribution as *Susan Corners*) that detects edges in an image. In this benchmark, the integer multiplier is the most frequently used unit (busy for 17.3% of the cycles in C0). The FP adder, multiplier, and divider are busy for 1% of the cycles in C0. While the usage statistics do not change for the FP units over the various optimization configurations, they change significantly for the integer multiplier (Table V). In C1, the number of busy cycles for this unit reduces by 4X (0.24 vs. 1.0), thereby increasing the savings by more than 9X (29.5% vs. 3.3%). In C3, the number of busy cycles reduces by almost 5X (0.21 vs. 1.0). This is due to strength reduction performed on induction variables, which are instrumental in replacing multiplication operations with equivalent add or subtract operations. The performance overhead (ovhd) for this benchmark ranges from 0.06% to 0.1%.

TABLE V

STATISTICS FOR *Susan Edges*

	cyc^1	Imul		slp^1
		sav	bsy^1	
C0	1.00	3.30	1.00	1.00
C1	0.74	29.50	0.24	6.97
C2	0.73	29.29	0.22	7.00
C3	0.78	29.95	0.21	1.07

^{1} The numbers in this column are normalized with the number in C0

TABLE VI

STATISTICS FOR *Mpeg2 Encode*

	cyc^1	Fadd		Fmul		slp^1
		sav	bsy^1	sav	bsy^1	
C0	1.00	36.39	1.00	38.24	1.00	1.00
C1^2	0.90	38.36	0.89	44.20	1.01	1.66
C2^2	0.93	38.02	0.95	39.00	1.01	1.70
C3^2	0.97	40.80	0.86	42.13	0.99	1.81
C3^3	0.78	41.37	0.86	42.63	1.00	0.79

^{1} The numbers in this column are normalized with the number in C0

^{2} Performed with floating point optimizations

^{3} Performed with procedure inlining and floating point optimizations

Mpeg2 Encode is a video compressing program which converts uncompressed video frames into MPEG-2 video coded bitstreams. The FP multiplier is the most frequently used unit (busy for 4.4% of the total cycles in C0), whereas FP divider is the most infrequently used units (busy for less than 0.01% in C0). Integer multiplier is busy for 3.3% of the cycles and FP adder is busy for 2.6% of the cycles (both in C0). Over all the optimization configurations, the savings in FP divider and the integer multiplier are almost uniform (50% and 31%, respectively). However, for the FP adder, 5% extra savings are achieved in C3 with procedure inlining over that in C0 (41% compared to 36%). This is a significant improvement considering the fact that the theoretical upper bound on maximum savings is 50%. For the FP multiplier, 6% extra savings are observed in C1 over those in C0 (44% compared to 38%). The range of performance overhead (ovhd) in implementing power gating is 0.08% to 0.10%.

Rsynth is a text to speech synthesis program in which the integer multiplier and the FP divider unit are used very infrequently (less than 0.1 %). Due to this, the fraction of savings for these units are almost uniform over all the optimization configurations (almost 50%). However, the FP

TABLE VII

STATISTICS FOR *Rsynth*

	cyc[1]	Fadd		Fmul		slp[1]
		sav	bsy[1]	sav	bsy[1]	
C0	1.00	0.81	1.00	0.74	1.00	1.00
C1[2]	0.82	1.01	0.74	1.31	0.95	0.02
C2[2]	0.83	1.01	0.74	1.17	0.95	0.02
C3[2]	0.83	1.23	0.74	1.98	0.95	0.02

[1] The numbers in this column are normalized with the number in C0

[2] Performed with procedure inlining and floating point optimizations

adder and the FP multiplier are used extensively (they are busy during 10% and 26% of the simulation cycles in C0). Due to such extensive usage of these units, the savings in them are negligible (less than 1%). However, the overhead of sleep instructions is very high since the source code description has numerous small procedures that carry out specific arithmetic functions. Therefore, when procedure inlining is perfomed, the number of sleep instructions inserted reduces by 50X. This is because the small procedures are inlined into their callers, thereby generating larger sequential program regions. Minor increase in savings (from 1.01% - 2.1%) can also be seen for these two units. This demonstrates the impact that procedure inlining can have on reducing the sleep instruction overhead in power gating. Table VII enumerates the statistics for these units over all the configurations. Except for C0, procedure inlining and floating point optimizations are performed in every configuration. It can be seen that when procedure inlining is performed, the busy cycles for the FP adder and FP multplier reduce by 26% and 4%, respectively, with respect to those in C0. Increase in code size due to procedure inlining is above 0.8%. The performance overhead (ovhd) due to power gating is 0.6% in C0 and 0.01%, on an average, in the rest of the configurations.

VI. CONCLUSIONS

In this work, we analyze various compiler optimization techniques from the perspective of improving the effectiveness of compiler-directed power gating of arithmetic functional units in an embedded microprocessor. We also perform extensive simulations in a strong experimental framework to substantiate the analyses above. We showed that, depending on the nature of the application, various compiler optimizations can improve the effectiveness of power gating significantly. Since the leakage component in CMOS circuits is increasing every technology generation, it is imperative that compilers will need to generate code to not only improve performance but also reduce energy consumption in microprocessors. The work presented in this paper serves as an important step towards satisfying that requirement.

REFERENCES

[1] R.K. Krishnamurthy, S.K. Mathew, M.A. Anders, S.K. Hsu, H. Kaul, and S. Borkar. High-performance and low-voltage challenges for sub-45nm microprocessor circuits. *International Conference on ASIC*, pages 283–286, 2005.

[2] K. Roy. Leakage Power Reduction in Low-Voltage CMOS Design. *IEEE International Conference on Electronics, Circuits and Systems*, pages 167–173, 1998.

[3] S. Roy, N. Ranganathan, and S. Katkoori. Exploration of Compiler Optimization Techniques for Enhancing Power Gating. *International Symposium on Circuits and Systems*, pages 1004–1007, 2009.

[4] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and Wu Ye. Influence of Compiler Optimizations on System Power. 9:801–804, 2001.

[5] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *International Symposium on Computer Architecture*, pages 240–251, 2001.

[6] K. Flautner, Z. Hu, and M. Martonosi. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *International Symposium on Computer Architecture*, pages 241–250, 2002.

[7] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural Techniques for Power Gating of Execution Units. *International Symposium on Low Power Electronics and Design*, pages 32–37, 2004.

[8] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimizing Static Power Dissipation by Functional Units Superscalar processors. *International Conference on Compiler Construction*, pages 261–274, 2002.

[9] Y. You, C. Lee, and J.K. Lee. Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors. *ACM Transactions on Design Automation of Electronic Systems*, pages 147–164, 2006.

[10] W. Zhang, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and V. De. Compiler Suppport for Reducing Leakage Energy Consumption. *Design Automation and Test in Europe*, pages 1146–1147, 2003.

[11] S. Roy, S. Katkoori, and N. Ranganathan. A Compiler Based Leakage Reduction Technique by Power-Gating Functional Units in Embedded Microprocessors. *International Conference on VLSI Design*, pages 215–220, 2007.

[12] N. Seki et al. A fine-grain dynamic sleep control scheme in MIPS R3000. *IEEE International Conference on Computer Design*, pages 612–617, 2008.

[13] N. Komoda et al. Compiler Directed Fine Grain Power Gating for leakage Reduction in Microprocessor Functional Units. *Workshop on Optimizations for DSP and Embedded Systems*, pages 42–51, 2009.

[14] Advanced RISC Machines Limited. ARM Product Backgrounder. `http://www.arm.com/miscPDFs/3823.pdf`, 2005.

[15] Advanced RISC Machines Limited. ARM7 Processor Architecture Data Sheet. `http://www.arm.com/`.

[16] GNU Project. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`.

[17] D. Burger and T. Austin. The Simplescalar Tool Set, version 2.0. Technical report, TR-97-1342, University of Wisconsin-Madison, 1997.

[18] P. Briggs and T.J. Harvey. Multiplication by Integer Constants. Technical report, Rice University, 1994.

[19] T. Granlund and P. Montgomery. Division by Invariant Integers using Multiplication. *Proc. ACM SIGPLAN, Conf. on PLDI*, pages 61–72, 1994.

[20] 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. *IEEE Annual Workshop on Workload Characterization*, pages 3–14, 2001.

[21] C Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. *International Symposium on Microarchitecture*, page 330, 1997.