Instruction Scheduling for Power Reduction in Processor-Based System Design

Hiroyuki Tomiyama Tohru Ishihara Akihiko Inoue Hiroto Yasuura
Department of Computer Science and Communication Engineering
Graduate School of Information Science and Electrical Engineering
Kyushu University
6–1 Kasuga-koen, Kasuga, Fukuoka 816 Japan

Abstract— This paper propose an instruction scheduling technique to reduce power consumed for off-chip driving. The technique minimizes the switching activity of a data bus between an on-chip cache and a main memory when instruction cache misses occur. The scheduling problem is formulated and a scheduling algorithm is also presented. Experimental results demonstrate the effectiveness and the efficiency of the proposed algorithm.

1 Introduction

Due to the growing market for portable multimedia products, power consumption has become one of severe constraints in embedded system design. Many recent high-performance embedded processors have on-chip caches which help power reduction of the system. This is because off-chip driving requires much power and the on-chip caches reduce data transfers among chips on the system board. However, even in processors with on-chip caches, power for off-chip driving is very dominating and becoming more dominating, up to 70% of the total chip power, along with the progress of transistor scales [6]. It has been becoming important to reduce off-chip power at system-level design phases.

In this paper, we concentrate on compiler optimization techniques for power reduction. There exist two approaches to reduce power consumed by off-chip drivers for microprocessor-based systems with on-chip caches. One approach is to reduce cache misses. So far, many compiler optimization techniques to improve cache performance have been proposed, such as prefetching of instructions and data, loop transformations for data caches, and code placement for instruction caches [3]. Another approach is to reduce average energy consumption per cache miss, but few efforts of compiler optimizations have been made towards this approach.

This paper proposes an instruction scheduling technique to reduce energy consumption per instruction cache miss. The proposed technique reduces transitions on a data bus between an on-chip cache and a main memory, and as a result, power consumed by off-chip drivers in the main memory is reduced. The technique is very effective for embedded system design since it require neither the additional hardware cost nor a loss of the system performance. Furthermore, most of compiler optimizations targeting cache miss reduction and the technique proposed in this paper are not exclusive but complementary to each other.

This paper is organized as follows. Section 2 summarizes compiler optimization techniques targeting low-power proposed before. In Section 3, an instruction scheduling problem for off-chip power minimization is proposed. A scheduling algorithm is proposed in Section 4, and experimental results are shown in Section 5. In Section 6, we provide conclusions and our future works.

2 Related Works

In the past few years, compiler optimizations targeting low-power systems have been studied [4, 10, 11].

Experiments by Tiwari et al. show that various compiler optimization techniques targeting high execution speed are also effective for low-energy [11]. One of the most effective approaches is register allocation optimization. Decreasing memory accesses reduces not only power consumed by the caches but also potential cache misses and pipeline stalls.

Lee et al. have studied compiler optimizations for DSPs (digital signal processors) [4]. Their experiments also indicate that compiler optimizations for high-performance result in low-energy. In [4], they also proposed a compilation technique, called operand swapping, which swaps source operands of Booth multipliers to reduce the switching activity.

Su et al. proposed an instruction scheduling technique, called cold scheduling [10]. The cold scheduling reorders instructions in such a way that the switching activity in the control path is minimized.

Some compilation techniques for cache miss reduction have been proposed before. One of the recent studies is presented in [13]. The technique places basic blocks of the program in a main memory in such a way that instruction cache misses are minimized using integer linear programming. Although the motivation is performance improvement of the system, the proposed technique is also effective for off-chip power reduction.

The approaches in the above papers can be classified into the following three categories: (a) reduction in instructions to be executed [4, 11], (b) reduction

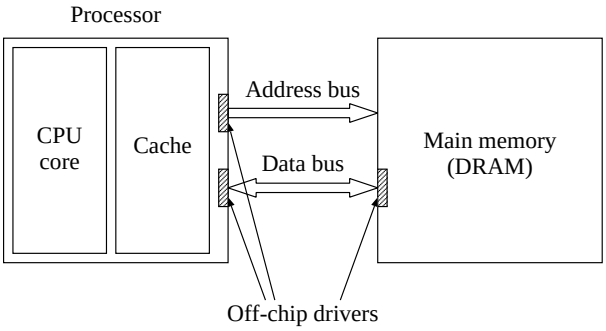


Figure 1: Hardware organization of target systems

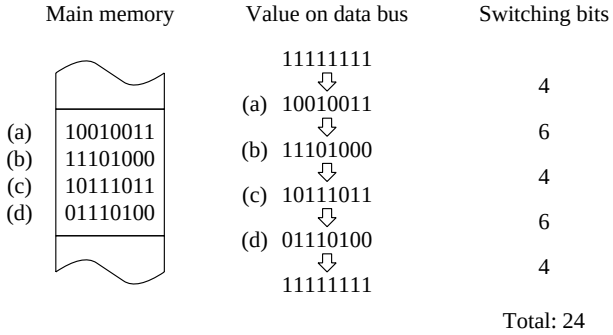


Figure 2: Bus transitions without optimization

in switching activity inside chips [4, 10, 11], and (c) reduction in off-chip transfers [11, 13]. This paper proposes another approach, reduction in bus transitions per off-chip transfer.

3 Problem Definition

3.1 Assumptions

In Figure 1, an example of our assumed hardware organizations is illustrated. The processor has a CPU core and an on-chip cache. When the CPU fails to fetch an instruction, i.e. an instruction cache miss, the processor first sends the address of the issued instruction to the main memory using the address bus. Next, the main memory sends all instructions in the memory block including the issued instruction to the cache using the data bus. Here, a memory block means a block in the main memory which can be mapped onto one cache line. After that, the CPU fetches the instruction in the cache.

For simplification, we assume as follows in this paper:

- All the machine instructions have the same bit-length.
- The bit-width of the data bus is equal to the instruction bit-length. For example, if the processor has 32-bit instructions, the data bus width is also 32 bits.
- When the data bus is not used, each wire in the data bus holds high-level to prevent from the high impedance condition.

Although the above assumptions can be relaxed by simple extensions of the technique proposed in this paper, such extensions are omitted in this paper due to the limited space for explanation.

3.2 Basic Idea

Since energy consumed by an off-chip driver is almost proportional to the number of transitions on the corresponding wire in the bus, low-energy is realized by reducing transitions on the bus. The instruction scheduling technique proposed in this paper schedules instructions in each basic block in a way that binary representations of consecutive two machine instructions in each memory block are less different.

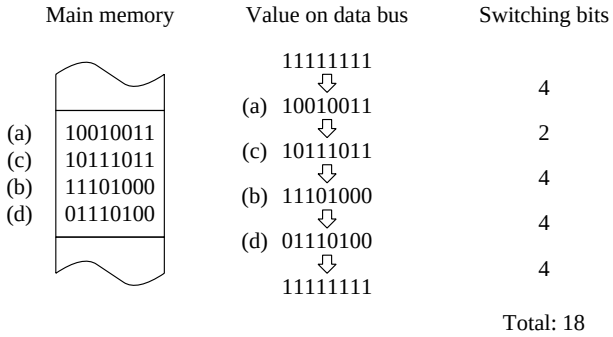


Figure 3: Bus transitions with scheduling

Let us consider the example in Figure 2, and assume 8-bit instruction width and 32-bit cache line size. There are four instructions (a)–(d) in the memory block. When the memory block is sent to the cache, the instruction (a) is sent first. At the time, four bits switch from high- to low-level. At the next cycle, (b) is sent to the cache and six bits switch to opposite level. As a result, the cache miss invokes twenty four transitions totally in the data bus. If changing the positions of two instructions (b) and (c) keeps the meaning of the program, it reduces bus transitions by 25%, from twenty four to eighteen transitions (See Figure 3). Thus the instruction scheduling can reduce transitions on the bus.

The instruction scheduling does not allow global instruction motions[1] because global instruction motions may change binary representations of instructions such as addresses of branch targets.

3.3 Problem Formulation

This section provides a formulation of the instruction scheduling problem for off-chip power reduction. In the formulation, we use the following notations.

[1] Global instruction motions move instructions beyond basic block boundaries.

- L : The number of instructions in a cache line.
- M : The number of memory blocks where an application program is located.
- I : The number of instructions in the application program. For simplification, we assume that I is equal to $L \times M$, but this assumption does not lose the generality.
- $Miss(j)$: The number of cache misses concerning the jth memory block. $0 \leq j \leq M-1$.
- $SW(j)$: The number of bus transitions when the jth memory block is sent to the cache once.
- $b(i)$: The binary representation of the ith instruction. $0 \leq i \leq I-1$.
- $sw(b, b')$: The Hamming distance between b and b'. $sw(b, b')$ corresponds to the number of switching bits between b and b'.
- $\mathbf{1}$: The bit vector where every bit is 1.

In general, $Miss(j)$'s are unknown at compilation time. If the behavior of the program does not depend on the input data to the program, or if an input data set is given, $Miss(j)$'s are predictable.

The number of transitions on the data bus without scheduling, T, is expressed by Formula (1).

$$T = \sum_{j=0}^{M-1} (Miss(j) \times SW(j)) \quad (1)$$

$$\begin{aligned} SW(j) = {} & sw(\mathbf{1}, b(j \times L)) \\ & + \sum_{k=1}^{L-1} sw(b(j \times L + k - 1), b(j \times L + k)) \\ & + sw(b((j+1) \times L - 1), \mathbf{1}) \end{aligned} \quad (2)$$

Please remember the assumption that each wire in the data bus holds high-level while the bus is not used. In Formula (1), $sw(\mathbf{1}, b(j \times L))$ denotes the number of switching bits when the first instruction in the jth memory block is sent to the cache, and $\sum_{k=1}^{L-1} sw(b(j \times L + k - 1), b(j \times L + k))$ is the switching bits when the rest of instructions in the block is sent. After that, $sw(b((j+1) \times L - 1), \mathbf{1})$ bits switches from low-level to high. The sum of the three values gives the number of bus transitions per cache miss, which is multiplied by the number of cache misses.

Next, let us denote the injection x and x^{-1} as follows.

- $x(i)$: The position of the ith instruction after scheduling. $0 \leq i \leq I-1$, $0 \leq x(i) \leq I-1$.
- x^{-1} : The inverse mapping of x. $x^{-1}(x(i)) = i$.

Then, the number of transitions on the data bus with instruction scheduling is expressed in Formula (1) and (3), which is obtained by replacing function b in Formula (2) with $b \circ x^{-1}$.

```
1:  Find_Schedule(G, PS, addr)
2:  {
3:    if (G = φ) return (φ, 0);
4:    if (Not_Found_in_Hash(G)) return Get_Hash(G);
      /* See Section 4.2 */
5:    SN := the set of schedulable nodes in G;
6:    v_last := the lastly scheduled node in PS;
7:    foreach v ∈ SN {
8:      if (addr is a boundary of memory blocks) {
9:        t_v := sw(b(v_last), 1) + sw(1, b(v));
10:     } else {
11:       t_v := sw(b(v_last), b(v_i));
12:     }
13:   }
14:   NSN := Select_N_Nodes(SN);
      /* See Section 4.3 */
15:   Increment addr;
16:   foreach v ∈ NSN {
17:     (S_v, C_v) := Find_Schedule(G − v, PS ∪ v, addr);
18:   }
19:   v := the node in NSN minimizing (C_v + t_v);
20:   Set_Hash(S_v ∪ v, C_v + t_v);
      /* See Section 4.2 */
21:   return (S_v ∪ v, C_v + t_v);
22: }
```

Figure 4: Scheduling algorithm

$$\begin{aligned} SW(j) = {} & sw(\mathbf{1}, b(x^{-1}(j \times L))) \\ & + \sum_{k=1}^{L-1} sw(\ b(x^{-1}(j \times L + k - 1)), \\ & \qquad\quad b(x^{-1}(j \times L + k))) \\ & + sw(b(x^{-1}((j+1) \times L - 1)), \mathbf{1}) \end{aligned} \quad (3)$$

The instruction scheduling problem is formally defined as follows: *For given L, M, $b(i)$'s, and $m(j)$'s, find x minimizing T keeping control and data dependencies among instructions.*

4 Scheduling Algorithm

4.1 Outline of the Algorithm

In Figure 4, a scheduling algorithm for off-chip power reduction is presented. The algorithm is applied to each basic block of an application program. Inputs to the algorithm are a DAG (directed acyclic graph) of instructions which are not scheduled yet, denoted by G, the partial schedule, PS, and the address of the main memory where the next instruction is located, $addr$. Each node and edge in the DAG represent an instruction and a control/data dependency between instructions, respectively. The algorithm outputs the schedule of G and the cost. The cost is the number of transitions on the data bus when the instructions in G are sent to the cache.

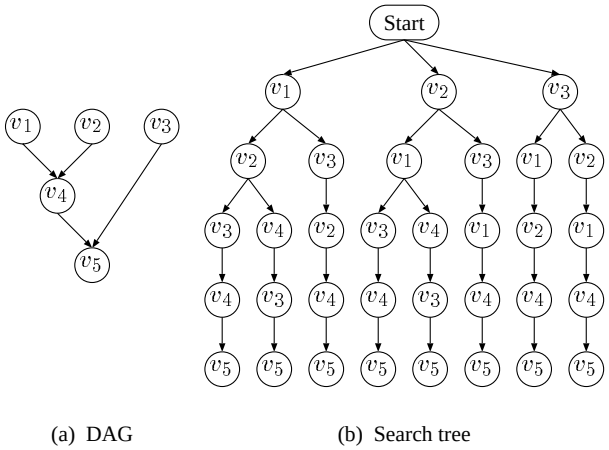


Figure 5: Example of DAG and its search tree

In this paper, how the algorithm works is described using some examples, instead of explaining each statement in Figure 4.

Basically, the algorithm examines all possible schedules of G, and finds the schedule with the lowest cost. Let us consider the DAG illustrated in Figure 5 (a). The DAG has eight possible schedules which are represented by the search tree in Figure 5 (b). The algorithm performs the depth-first search to the search tree, and outputs the best solution.

Since searching the whole search tree is inefficient in terms of computation time, the algorithm employs two techniques for speed-up. One technique avoids the potentially redundant search. Another technique limits the number of subtrees to be searched. In Section 4.2 and Section 4.3, the two techniques are described respectively.

4.2 Redundant Search Space Reduction

For example, let us consider the DAG and its search tree in Figure 6, where two subtrees enclosed by dotted circles have the same structure. If the schedule $(v_3 \rightarrow v_5 \rightarrow v_4 \rightarrow v_6)$ has the lowest cost in the left subtree, the schedule is also the best in the right subtree. Since the two subtrees have the same solution, searching both subtrees just wastes the computation time. The proposed algorithm avoids such potential redundant search by remembering the best schedule of each searched subtree in a hash table. When searching tree, if the DAG has already been registered in the hash table, the schedule and its cost in the table are used.

This mechanism is realized by the three functions, *Not_Found_in_Hash*, *Get_Hash*, and *Set_Hash* in the algorithm *Find_Schedule* (Line 4 and 20 in Figure 4).

4.3 Heuristics

If a node in a search tree have a large number of children, to search all children's subtrees consumes a long time. The scheduling algorithm selects up to N children whose subtrees are likely to have good schedules, and continues scheduling for their subtrees (Line 14 in Figure 4). Generally, larger N leads to better schedules.

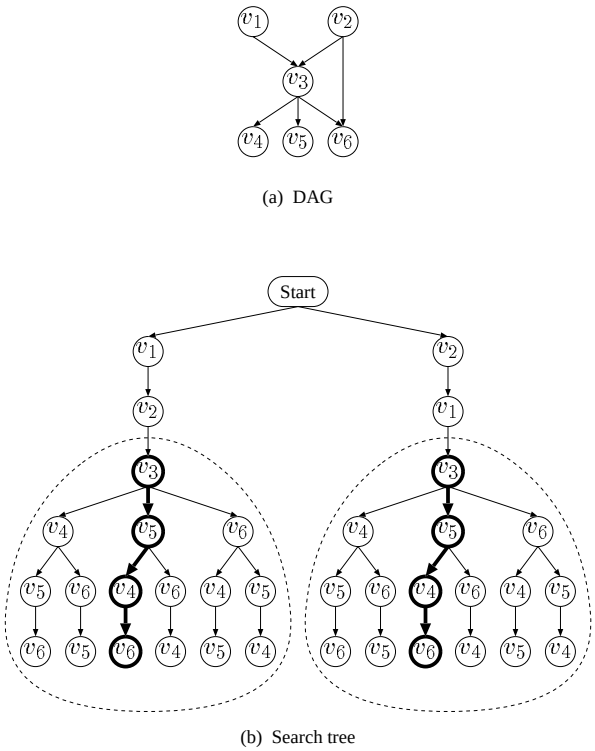
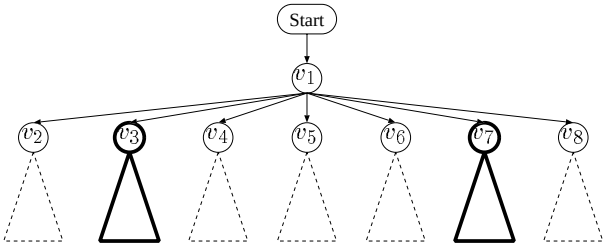


Figure 6: Example of redundant search space reduction

When selecting children to be searched, the algorithm only considers the number of the switching bits between the current node and each child of the node. Let us consider the search tree illustrated in Figure 7, and assume that N is set to 2. The binary representation of each node, $b(v_i)$, is also described in the figure. The node v_1 must be scheduled first, and there exist seven nodes schedulable after v_1. Since $sw(b(v_1), b(v_3))$ and $sw(b(v_1), b(v_7))$ are the lowest two, the algorithm selects v_3 and v_7 as the candidate nodes to be scheduled next to v_1, and continues scheduling for their subtrees. Note that if the address of the instruction next to v_1 is a boundary of memory blocks, v_4 and v_6 are selected as the candidates. This is because we assume that each wire in the data bus holds high-level while the cache hits.

The worst case time complexity of the algorithm is $O(n^2)$ if $N = 1$, otherwise $O(N^n \cdot n)$, where n is the number of instructions in the basic block.



$b(v_1) = 00001111$
$b(v_2) = 11110000, \quad sw(b(v_1), b(v_2)) = 8$
$b(v_3) = 10001111, \quad sw(b(v_1), b(v_3)) = 1$
$b(v_4) = 11111111, \quad sw(b(v_1), b(v_4)) = 4$
$b(v_5) = 00000000, \quad sw(b(v_1), b(v_5)) = 4$
$b(v_6) = 11111110, \quad sw(b(v_1), b(v_6)) = 5$
$b(v_7) = 00001100, \quad sw(b(v_1), b(v_7)) = 2$
$b(v_8) = 11110001, \quad sw(b(v_1), b(v_8)) = 7$

Figure 7: Example of heuristics

4.4 Limitations of the Algorithm

Even if N is large sufficiently[2], it is not guaranteed that the algorithm always finds the optimal solution of the scheduling problem defined in Section 3. There exist two reasons. One reason is that the algorithm takes no account of $m(j)$ which denotes the number of cache misses caused by the jth memory block. In general, $m(j)$ is unknown before executing the application program. If $m(j)$ does not depend on input data to the program, or if $m(j)$ is predictable, considering $m(j)$ during scheduling leads to better solutions.

Another reason is that, when scheduling a basic block, the algorithm only considers which instruction was scheduled last in the previous basic block, but does not consider which instruction is likely to be scheduled first in the next basic block. In order to guarantee the optimality, all basic blocks must be scheduled simultaneously. However, in this case, the scheduling time may not be acceptable.

5 Experiments

The scheduling algorithm proposed in the previous section is implemented in C, and applied to several benchmark programs for various N (See Section 4.3). Table 1 shows the number of machine instructions and basic blocks in each program, and the number of instructions in the largest basic block. `compress(UNIX)` is a UNIX application and the other programs are selected from [7]. The SPARC instruction set architecture is used as a target. Since each SPARC instruction has 4-byte width, we assume that the data bus has a width of 4 bytes.

We assume two instruction cache models. One is a direct mapped, 128-byte instruction cache with 32-byte lines, and another is a direct mapped, 128-byte instruction cache with 64-byte lines. Experimental results are shown in Table 2 and Table 3, respectively.

[2] N is large sufficiently if N is equal to or larger than the number of instructions in the basic block.

Table 1: Benchmark programs

	instructions	basic blocks	largest block size
`compress`	41	6	16
`laplace`	79	6	37
`linear`	48	10	15
`lowpass`	78	6	49
`sor`	100	6	43
`wavelet`	101	9	37
`compress(UNIX)`	1,805	471	35

In each table, the second column labeled "naive" gives the number of transitions without the optimization. The "$N = 1$" column gives the number of transitions on the data bus and the CPU time required for scheduling when N is set to 1. The "best" column shows the case when N is large sufficiently, but, in any case in our experiments, the best schedule is also obtained when $N = 5$. The scheduling time is measured on SPARCstation 5 (microSPARC-II, 85MHz, 32MB, Solaris 2.5).

Experimental results show that the algorithm achieves a reduction of up to 28% in transitions on the data bus. Significant reduction in power consumed for off-chip driving is expected even though the technique does not reduce transitions at data cache miss time.

The computation time for scheduling is short enough, within 1.3 seconds in any case. It shows that the algorithm proposed in Section 4 is efficient though it is very simple.

6 Conclusions and Future Works

In this paper, we have proposed an instruction scheduling technique to reduce energy per instruction cache miss, which is consumed for off-chip driving. The scheduling technique reduces transitions on a data bus between an on-chip cache and a main memory of the system. The approach can reduce power consumption without extra hardware cost and performance loss. The scheduling problem is formulated and a scheduling algorithm is proposed. Experimental results show that the proposed scheduling algorithm achieves significant reduction in transitions on the data bus, up to 28% of reduction, and runs efficiently.

There remain a lot of works to further reduce off-chip power consumptions. Since the scheduling technique is compatible with most of compilation techniques for cache miss reduction such as [13], to apply the technique proposed in this paper with these techniques will achieve greater power reduction.

The power consumption for off-chip driving depends on code assignment of machine instructions. In design of application specific processors, more reduction in bus transitions is expected by encoding instructions in such a way that the operation fields of instructions which are frequently executed consecutively

Table 2: Experimental results (32-byte cache lines)

	# transitions and CPU time			reduction	instruction cache miss rate
	naive	our proposed algorithm			
		$N = 1$	best		
`compress`	17,002	16,174 (0.03 sec)	16,152 (0.04 sec)	5.0 %	1.3 %
`laplace`	4,282,378	4,101,352 (0.07 sec)	3,901,726 (0.08 sec)	8.9 %	10.8 %
`linear`	514,364	466,346 (0.04 sec)	428,346 (0.07 sec)	16.7 %	0.1 %
`lowpass`	7,761,708	5,941,252 (0.05 sec)	5,558,046 (1.21 sec)	28.4 %	12.3 %
`sor`	2,432,938	2,047,392 (0.06 sec)	1,976,212 (1.37 sec)	18.8 %	10.9 %
`wavelet`	13,750	10,312 (0.08 sec)	10,084 (0.33 sec)	26.7 %	10.5 %
`compress(UNIX)`	1,090,232	1,049,472 (0.30 sec)	1,032,682 (0.48 sec)	5.3 %	14.2 %

Table 3: Experimental results (64-byte cache lines)

	# transitions and CPU time			reduction	instruction cache miss rate
	naive	our proposed algorithm			
		$N = 1$	best		
`compress`	36,134	35,104 (0.03 sec)	33,892 (0.06 sec)	6.3 %	1.3 %
`laplace`	7,081,276	6,741,208 (0.05 sec)	6,541,176 (0.10 sec)	7.6 %	10.8 %
`linear`	446,440	392,384 (0.04 sec)	362,368 (0.06 sec)	18.8 %	0.1 %
`lowpass`	8,520,594	6,520,556 (0.08 sec)	6,120,538 (1.23 sec)	28.2 %	8.1 %
`sor`	3,724,530	3,293,460 (0.07 sec)	2,930,224 (1.10 sec)	21.3 %	10.8 %
`wavelet`	23,040	17,528 (0.06 sec)	16,574 (0.34 sec)	28.1 %	10.2 %
`compress(UNIX)`	1,087,742	1,050,040 (0.33 sec)	1,031,182 (0.49 sec)	5.2 %	8.3 %

are assigned closely. Similarly, it may be effective for compilers to assign registers in such a way that the operand fields of instructions less switches. To optimize instruction encoding, register assignment, and instruction scheduling simultaneously is one of our future works.

This paper concentrates on bus transitions at instruction cache misses and does not consider data cache misses. Reducing off-chip power at data cache misses is also an interesting research avenue for us.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addition-Wesley, 1986.

[2] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addition-Wesley, 1990.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.

[4] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and minimization techniques for embedded DSP software," *IEEE Trans. VLSI Systems*, vol. 5, no. 1, pp. 123–135, March 1997.

[5] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Code optimization techniques for embedded DSP microprocessor," In *Proc. of 32nd Design Automation Conf.*, 1995.

[6] D. Liu and C. Svensson, "Power consumption estimation in CMOS VLSI chips," *IEEE J. Solid-State Circuits*, vol. 29, no. 6, pp. 663–670, June 1994.

[7] P. R. Panda and N. Dutt, "1995 high level synthesis design repository," In *Proc. of 8th Int'l Symp. System Synthesis*, 1995.

[8] P. R. Panda and N. D. Dutt, "Reducing address bus transitions for low power memory mapping," In *Proc. of ED&TC96*, pp. 63–67, 1996.

[9] SPARC International, Inc., *The SPARC Architecture Manual Version 8*, 1992.

[10] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Low power architecture design and compilation techniques for high-performance processors," In *Proc. of COMPCON'94*, 1994.

[11] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," In *Proc. of Symp. Low-Power Electronics*, 1994.

[12] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Trans. VLSI Systems*, vol. 2, no. 4, pp. 437–445, 1994.

[13] H. Tomiyama and H. Yasuura, "Optimal code placement of embedded software for instruction caches," In *Proc. of ED&TC96*, pp. 96–101, 1996.