An Operation Rearrangement Technique for Low-Power VLIW Instruction Fetch*

Dongkun Shin and Jihong Kim
School of Computer Science and Engineering
Seoul National University
E-mail: {sdk, jihong}@davinci.snu.ac.kr

Abstract

As mobile applications are required to handle more computing-intensive tasks, many mobile devices are designed using VLIW processors for high performance. In VLIW machines where a single instruction contains multiple operations, the power consumption during instruction fetches varies significantly depending on how the operations are arranged within the instruction. In this paper, we describe a post-pass optimal operation rearrangement method for low-power VLIW instruction fetch. The proposed method modifies operation placement orders within VLIW instructions so that the switching activity between successive instruction fetches is minimized. Our experiment shows that the switching activity can be reduced by 34% on average for benchmark programs.

I. Introduction

As mobile applications are required to handle more computing-intensive tasks (such as video decoding), many mobile devices are designed using VLIW processors for high performance. For example, the Crusoe processors [9] from Transmeta (which were developed for mobile Internet computing market) are based on 64 bits or 128 bits VLIW CPU cores. Fujitsu Microelectronics' FR300 [5] (whose main application area is in wireless cellular phones) also has a VLIW architecture. In addition, there are many VLIW digital signal processors such as Texas Instruments' TMS320C6x series that can be used for wireless devices [4, 6].

While VLIW CPU-based mobile devices generally provide enough computing power to handle many computing intensive applications, they usually consume a large amount of power. For example, TMS320C620x processors consume between 1.2W and 2.3W at 1.8V while high-end embedded microprocessors such as StrongArm 110 consume between 100mW and 1W at 3V [8, 13]. Therefore, in designing VLIW CPU-based mobile devices, low power consumption is a dominant design constraint.

In digital CMOS circuits (that use well-designed logic gates), switching activity accounts for over 90% of total power consumption [1]. Therefore, many techniques have been proposed and developed to reduce the amount of switching activity in multiple levels of design abstraction [3]. For example, bus-invert coding [14] reduces a significant number of bit changes from bus lines by dynamically inverting the bus lines when the number of switched bus lines is more than half the number of bitlines. Register relabeling [11] assigns register numbers of instructions so that more frequently consecutive register numbers have a smaller Hamming distance, thus reducing the switching activity from the instruction fetch and decode logic.

In this paper, we propose a post-pass optimization technique that can significantly reduce switching activity during the instruction fetch phase in VLIW processors. The proposed method takes advantage of a VLIW machine's instruction encoding characteristic: VLIW CPUs can place the same operation in multiple operation slots within the VLIW instruction.[1] Since a single instruction generally contains multiple operations in a VLIW CPU, the power consumption during instruction fetches varies significantly depending on how the operations are arranged within the instruction. We reduce switching activity by modifying operation placement orders within VLIW instructions so that the switching activity between successive instruction fetches is minimized.

The organization of the rest of the paper is as follows. Before presenting the proposed operation rearrangement technique, we review prior works on low-power instruction scheduling in Section II. In Section III, we describe a target VLIW machine model and define several terms. An operation rearrangement technique applicable to a single basic block is explained in Section IV while the complete solution is discussed in Section V. Experimental results are presented in Section VI followed by conclusions in Section VII.

*This work was supported in part by BK21 Information Technology program.

[1] We distinguish between an operation and an instruction in a VLIW CPU. A VLIW *instruction* is assumed to consist of several *operations*.

II. Related Works

The low-power instruction scheduling problem has been recently investigated by several research groups.

Su *et al.* proposed an instruction scheduling technique, called cold scheduling, to reduce the amount of switching activity in the control path [15]. Used in conjunction with a traditional list scheduling algorithm, cold scheduling schedules instructions in the ready list based on the power cost of an instruction. The power cost of an instruction is determined by the number of bit changes when the instruction in question is scheduled following the last instruction. Su *et al.* show that the combination of Gray code addressing and cold scheduling results in a 20-30% reduction in the switching activity from the control path.

Tiwari *et al.* show that conventional compiler optimization techniques targeting high performance are also effective for low-power software [16, 17]. Their experiments indicate that an optimal register allocation technique is effective in reducing power consumption.

Lee *et al.* have investigated the low-power scheduling problem for DSP-based systems [10]. They take into consideration what they term circuit-state overhead which is the switching activity between a pair of specific instructions. Through the code rescheduling based on circuit-state overhead, energy savings up to 40% were achieved on the benchmarks used.

Since off-chip driving and bus consume a significant amount of power in microprocessor-based systems, low-power instruction scheduling was studied to reduce the switching activity on system bus. Tomiyama *et al.* proposed an instruction scheduling technique which reduces transitions on an instruction bus between an on-chip cache and a main memory when instruction cache misses occur [19]. This scheduling technique schedules instructions in each basic block in a way that binary representations of consecutive two machine instructions are less different while maintaining the control/data dependencies of the original program.

Most of existing low-power instruction scheduling techniques (including the techniques described above), however, assume that processors can issue at most one instruction at each cycle. Therefore, these techniques can not be directly applied to multiple-issue machines such as a VLIW CPU. In a VLIW CPU, since multiple operations are packed into a single instruction, two levels of scheduling decisions should be made to reduce power consumption. In the first level, we have to decide that which operations are packed into which instructions. Once the first level scheduling decision is made, in the second level, we have to decide which orders the selected operations are placed in specific instructions. The technique proposed in this paper solves the second-level low-power scheduling problem for a VLIW CPU assuming that the decision for the first-level scheduling problem was already made.

One recent study investigated a low-power instruction scheduling technique for a VLIW CPU [18]. However, the goal of [18] was to reduce the *peak* power dissipation. The scheduling algorithm described in [18] schedules an operation in the current instruction as long as the power dissipation of the current instruction does not exceed the given threshold value. Although effective in reducing the peak power dissipation, this algorithm does not take account of the inter-instruction effect and inter-operation effect during the scheduling process. Our scheduling algorithm proposed in this paper considers both effects in arranging the operations within the instruction, thus resulting in a better solution.

III. VLIW Machine Model and Definitions

A. Target VLIW Machine Model

VLIW architectures use long instruction words to execute multiple operations simultaneously. In specifying multiple operations within a single VLIW instruction, two encoding methods are typically used: uncompressed encoding and compressed encoding [2]. In a VLIW machine with an uncompressed encoding, each operation slot of a VLIW instruction corresponds to a particular functional unit. The operation specified in a particular operation slot, therefore, is executed only in the corresponding functional unit. If a functional unit is not scheduled to execute an operation at the given cycle, NOP should be specified in the corresponding operation slot. Under this encoding method, the number of candidate operation slots for an operation is limited to the number of corresponding functional units that can execute the operation.

On the other hand, in a VLIW machine with a compressed encoding, the position of operation slots within a VLIW instruction does not directly correspond to a particular functional unit. The assignment of a particular functional unit to an operation is generally decided by the functional unit subfield of the operation encoding. The functional unit subfield specifies which functional unit should be assigned to the operation. In addition, in order to increase memory utilization, NOP operations are not explicitly encoded in the VLIW instruction. In this type of VLIW machines, an operation can be placed in any operation slot within the same VLIW instruction.

Figures 1 and 2 compare two types of encoding methods using a sample VLIW program sequence S. In the program sequence S, three VLIW instructions are shown where "||" specifies parallel operations that are executed simultaneously. As shown in Figures 1.(b) and 1.(c), in an uncompressed VLIW instruction encoding, the operation rearrangement is rather limited. For example, in the first VLIW instruction, IADD and NOP, FADD and NOP, and LOAD and STORE can be exchanged. For a compressed VLIW instruction encoding shown in Figures 2.(b) and 2.(c), there are more chances for operation rearrangements because there is no direct correspondence between the position of an operation slot and a corresponding functional unit. For example, for the first VLIW

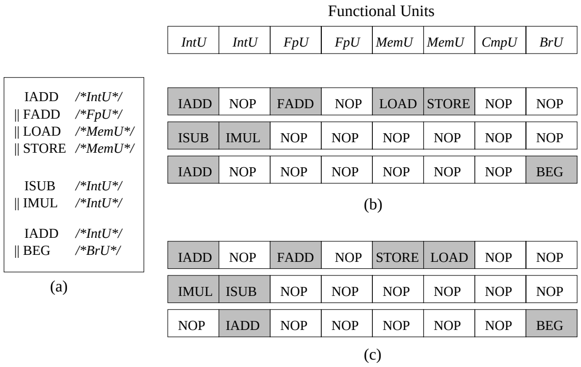

Functional Units
IntU IntU FpU FpU MemU MemU CmpU BrU
IADD /*IntU*/
|| FADD /*FpU*/
|| LOAD /*MemU*/
|| STORE /*MemU*/
ISUB /*IntU*/
|| IMUL /*IntU*/
IADD /*IntU*/
|| BEG /*BrU*/
(a)
IADD NOP FADD NOP LOAD STORE NOP NOP
ISUB IMUL NOP NOP NOP NOP NOP NOP
IADD NOP NOP NOP NOP NOP NOP BEG
(b)
IADD NOP FADD NOP STORE LOAD NOP NOP
IMUL ISUB NOP NOP NOP NOP NOP NOP
NOP IADD NOP NOP NOP NOP NOP BEG
(c)

Fig. 1. Uncompressed VLIW instruction encoding; (a) a sample instruction sequence S, (b) one uncompressed encoding of S and (c) an alternative encoding of S.

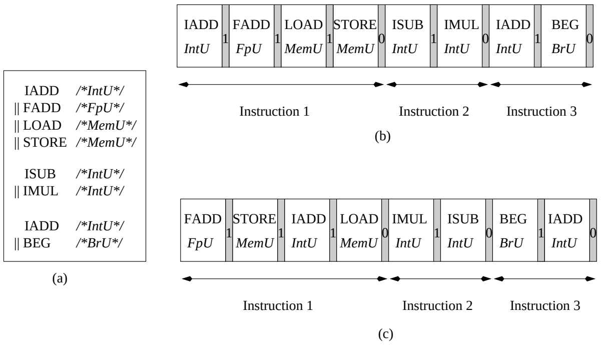


Fig. 2. Compressed VLIW instruction encoding; (a) a sample instruction sequence S, (b) one compressed encoding of S and (c) an alternative encoding of S.

instruction of S, 4! different operation rearrangements are all possible.[2] Although the proposed operation rearrangement technique is equally effective for a VLIW machine with an uncompressed encoding, we assume that a target VLIW CPU was encoded using a compressed encoding method.

Throughout this paper, we consider a target system with an architectural organization shown in Figure 3. The VLIW processor with a compressed encoding has an on-chip instruction cache. The VLIW instructions are fetched through the b_{cache}-bit width instruction bus. If the instruction is not found in the on-chip instruction cache, the corresponding memory block is fetched from the main memory through the b_{mem}-bit width instruction bus. Because of the compressed encoding format, several VLIW instructions can be fetched together in a single fetch from the instruction cache. We call these instructions *a fetch packet* as a group. For a description purpose, we make the following assumptions on the target system:

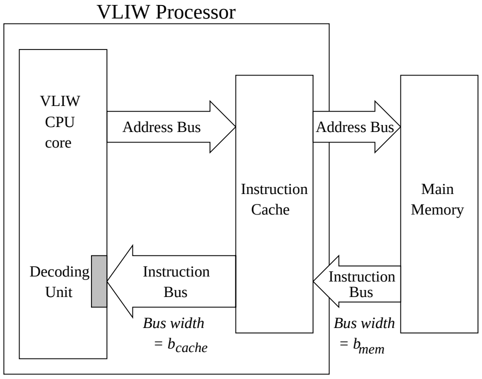


Fig. 3. Target system architecture.

- In a single b_{cache}-bit fetch packet, exactly N operations are included. (That is, the width of a single operation slot is exactly b_{cache}/N.)
- No instruction crosses the fetch packet boundary.
- b_{mem} is equal to the operation width. (That is, b_{mem} = b_{cache}/N.)
- When the external instruction bus is not used, each line in the external bus is assumed to hold a logic 1 value to prevent from the high impedance condition.

B. Definitions

In explaining the operation rearrangement technique, we use the following definitions:

Definition 1 A permutation $\sigma : \{1, \cdots, n\} \rightarrow \{1, \cdots, n\}$ is said to be an operation rearrangement function.

Definition 2 Two VLIW instructions $I_1 = (OP_1^1, OP_2^1, \cdots, OP_n^1)$ and $I_2 = (OP_1^2, OP_2^2, \cdots, OP_n^2)$ are said to be equivalent under operation rearrangement if there exists an operation rearrangement function σ such that $OP_{\sigma(i)}^1 = OP_i^2$ for all $1 \leq i \leq n$.

Definition 3 Two fetch packets $FP_1 = (I_1^1, I_2^1, \cdots, I_n^1)$ and $FP_2 = (I_1^2, I_2^2, \cdots, I_n^2)$ are said to be equivalent under operation rearrangement if there exist operation rearrangement functions $(\sigma_1, \sigma_2, \cdots, \sigma_n)$ such that I_i^1 is equivalent to I_i^2 under σ_i for all $1 \leq i \leq n$. $EQ(FP_i)$ is used to represent the set of equivalent fetch packets for a given FP_i.

Definition 4 Two basic blocks $bb_1 = ($ FP_1^1, FP_2^1, $\cdots$, FP_n^1 $)$ and $bb_2 = (FP_1^2, FP_2^2, \cdots, FP_n^2)$ are said to be equivalent under operation rearrangement if FP_i^1 is equivalent to FP_i^2 under operation rearrangement for all $1 \leq i \leq n$. $EQ(bb)$ is used to represent the set of equivalent basic blocks for a given basic block bb.

Definition 5 Two programs $S_1 = (bb_1^1, bb_2^1, \cdots, bb_n^1)$ and $S_2 = (bb_1^2, bb_2^2, \cdots, bb_n^2)$ are said to be equivalent under operation rearrangement if bb_i^1 is equivalent to bb_i^2

[2] In Figures 2.(b) and 2.(c), parallel operations within the same VLIW instruction is specified using tail bits (shown in the shaded boxes). If a tail bit of an operation O is 1, the operation O is executed in parallel with the next operation. Otherwise, the next operation is executed after the current instruction is executed.

under operation rearrangement for all $1 \leq i \leq n$. $EQ(S)$ is used to represent the set of equivalent programs for a given program S.

In the rest of paper, we use "equivalent" to mean "equivalent under operation rearrangement" where no confusion arises.

IV. Local Operation Rearrangement Problem

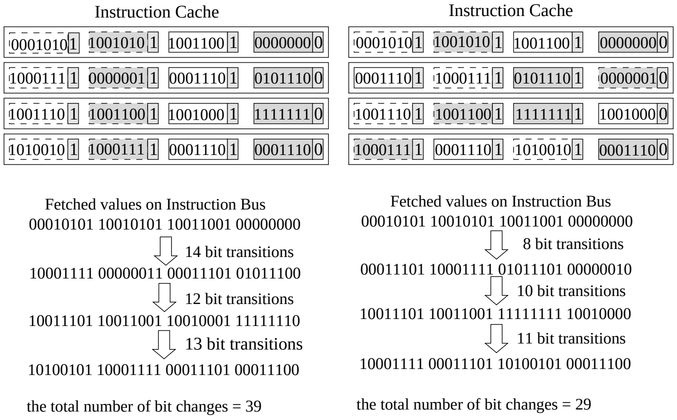
In this section, we consider a simpler operation rearrangement problem that we call *local operation rearrangement problem (LOR)*. In the LOR problem, each basic block is independently considered and assumed that the basic block is fetched from the main memory and executed only once. Since the basic block is fetched from the main memory, there are cache misses associated during the instruction fetch. A complete operation rearrangement problem that we call *global operation rearrangement problem (GOR)* is discussed in the next section. In the GOR problem, all the basic blocks are simultaneously considered.

A. Basic Idea

In order to reduce the switching activity during the instruction fetch phase in a target system, we reduce the number of bit transitions between successive instruction fetches, because switching activity is directly proportional to the number of bit changes. Since, in a VLIW machine with a compressed encoding, an operation can be placed in any operation slot within the instruction boundary, the number of bit transitions between successive instruction fetches can be reduced by reordering given VLIW instructions to equivalent instructions that have less switching activity. Consider an example shown in Figure 4. There are four fetch packets each of which is 32-bit wide (that is, $b_{cache} = 32$). In the example, each fetch packet consists of a single VLIW instruction which in turn consists of four operations. Figure 4.(b) shows the instruction sequence after an operation placement order was modified to reduce the bit transitions in the instruction bus. When the four instructions are executed sequentially only once, the rearranged instruction sequence shown in Figure 4.(b) reduces the total number of bit changes by about 25% from 39 to 29, while maintaining the same semantics of the original sequence.

B. LOR Problem Formulation

If a given basic block B is executed only once, in our target architecture shown in Figure 3, the number of bit changes SW^B during the instruction fetch phase is given by the sum of two terms, SW^B_{cache} and SW^B_{mem}. SW^B_{cache} represents the number of bit changes at the internal instruction bus and SW_{mem} indicates the number of bit changes at the external instruction bus. Using the notations explained in Table 1, SW^B_{cache} and SW^B_{mem} are computed as follows.



(a) Before operation rearrangement

(b) After operation rearrangement

Fig. 4. An operation rearrangement example.

SW^B_{cache} is the sum of all the bit changes incurred during successive fetches of fetch packets from the instruction cache and calculated as follows:

$$SW^B_{cache} = \sum_{i=1}^{N_{fp}(B)-1} d_{fp}(FP^B_i, FP^B_{i+1}) \qquad (1)$$

SW^B_{mem} is the sum of all the bit changes between adjacent operation fetches from the main memory. Since we assumed that b_{mem} is equal to b_{cache}/N_{op} in Section III, if we assume that there is only one cache miss for each memory block and basic blocks are aligned by the cache memory block size, SW^B_{mem} is calculated as follows:

$$\begin{aligned} SW^B_{mem} &= \sum_{i=1}^{N_{fp}(B)} \sum_{n=1}^{N_{op}-1} d_{op}(OP_n^{FP^B_i}, OP_{n+1}^{FP^B_i}) \\ &+ \sum_{i=1}^{N_{fp}(B)-1} d_{op}(OP_{N_{op}}^{FP^B_i}, OP_1^{FP^B_{i+1}}) \\ &+ d_{op}(\mathbf{1}, OP_1^{FP^B_1}) + d_{op}(OP_{N_{op}}^{FP^B_{N_{fp}(B)}}, \mathbf{1}) \qquad (2) \end{aligned}$$

Assuming the load capacitance ratio of the internal instruction bus to the external instruction bus is $\frac{1}{\alpha}$, SW^B is computed as follows using the Equations (1) and (2):

$$\begin{aligned} SW^B &= SW^B_{cache} + \alpha \cdot SW^B_{mem} \\ &= \sum_{i=1}^{N_{fp}(B)-1} SW^{inter}_{FP}(FP^B_i, FP^B_{i+1}) \\ &+ \sum_{i=1}^{N_{fp}(B)} SW^{intra}_{FP}(FP^B_i) \qquad (3) \end{aligned}$$

Symbol	Meaning
$N_{fp}(B)$	The number of fetch packets in a basic block B.
N_{op}	The number of operations in a fetch packet. (This is a fixed value regardless of B.)
$\mathbf{1}$	The bit vector where every bit is 1 and whose length is b_{mem}.
FP_i^B	The i-th fetch packet of a basic block B.
$OP_n^{FP_i^B}$	The n-th operation of FP_i^B. (Within a fetch packet FP_i^B, the first operation is $OP_1^{FP_i^B}$ and the last one is $OP_{N_{op}}^{FP_i^B}$.)
$d_{fp}(FP_i^B, FP_j^B)$	The Hamming distance between the fetch packets FP_i^B and FP_j^B.
$d_{op}(OP_n^{FP_i^B}, OP_m^{FP_j^B})$	The Hamming distance between the operations $OP_n^{FP_i^B}$ and $OP_m^{FP_j^B}$.

TABLE 1
NOTATIONS USED IN SECTION IV.B

where

$$SW_{FP}^{inter}(FP_i^B, FP_{i+1}^B) = d_{fp}(FP_i^B, FP_{i+1}^B) + \alpha \cdot d_{op}(OP_{N_{op}}^{FP_i^B}, OP_1^{FP_{i+1}^B}) \quad (4)$$

$$SW_{FP}^{intra}(FP_i^B) = \begin{cases} \alpha \cdot d_{op}(\mathbf{1}, OP_1^{FP_i^B}) + S_{op} & \text{if } i = 1 \\ \alpha \cdot d_{op}(OP_{N_{op}}^{FP_i^B}, \mathbf{1}) + S_{op} & \text{if } i = N_{fp}(B) \\ S_{op} & \text{otherwise} \end{cases} \quad (5)$$

$$\text{(where } S_{op} = \alpha \cdot \sum_{n=1}^{N_{op}-1} d_{op}(OP_n^{FP_i^B}, OP_{n+1}^{FP_i^B})\text{)}$$

Given a basic block B, the LOR problem is to find an equivalent basic block B' such that $SW^{B'} \leq SW^{B''}$ for all $B'' \in EQ(B)$. If operations are rearranged, $d_{fp}(FP_i^B, FP_{i+1}^B)$, $d_{op}(OP_{N_{op}}^{FP_i^B}, OP_1^{FP_{i+1}^B})$ and $d_{op}(OP_n^{FP_i^B}, OP_{n+1}^{FP_i^B})$ in Equations (4) and (5) are changed.

C. Optimal Solution for LOR

We compute an optimal solution for the LOR problem by converting the LOR problem to the shortest path problem between two special nodes, START and END. Using the notations described in Table 2, given a basic block B, we construct a weighted directed graph $G_B = \{V, E, W_{node}, W_{edge}\}$, where

$$\begin{aligned} V &= \{\mathsf{START}, \mathsf{END}\} \cup \bigcup_{i=1}^{N_{fp}(B)} EQ(FP_i^B) \\ &= \{\mathsf{START}, \mathsf{END}\} \cup \bigcup_{i=1}^{N_{fp}(B)} \{FP_{i,1}^B, \cdots, FP_{i,N_{eq}(FP_i^B)}^B\}, \\ E &= \{(v,w) \mid v = \mathsf{START}, w \in EQ(FP_1^B)\} \cup \\ &\quad \{(v,w) \mid w = \mathsf{END}, v \in EQ(FP_{N_{fp}(B)}^B)\} \cup \\ &\quad \{(v,w) \mid v \in EQ(FP_i^B), w \in EQ(FP_{i+1}^B) \\ &\quad\quad \text{for } 1 \leq i < N_{fp}(B)\}, \end{aligned}$$

$$W_{node}(v) = \begin{cases} SW_{FP}^{intra}(v) & \text{if } v \in V - \{\mathsf{START}, \mathsf{END}\} \\ 0 & \text{otherwise} \end{cases}, \text{ and}$$

$$W_{edge}(v,w) = \begin{cases} SW_{FP}^{inter}(v,w) & \text{if } v, w \in V - \{\mathsf{START}, \mathsf{END}\} \\ 0 & \text{otherwise.} \end{cases}$$

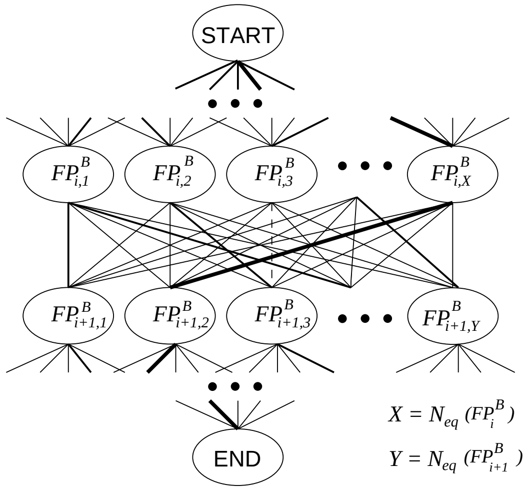


Fig. 5. A shortest path problem formulation of the LOR problem (with node and edge weights omitted).

Figure 5 shows an example graph constructed by transforming the LOR problem to the shortest path problem. For each fetch packet FP_i^B, $N_{eq}(FP_i^B)$ vertices are created in G_B, and for successive fetch packets, FP_i^B and FP_{i+1}^B, every pair of $(FP_{i,k}^B, FP_{i+1,k'}^B)$ is connected by an edge. We call the $N_{eq}(FP_i^B)$ vertices created from the fetch packet FP_i^B to be in the level i. In the graph G_B, the distance of a path $P = (\mathsf{START}, v_1, \cdots, v_k, \mathsf{END})$ is given by $\sum_{i=1}^{k} W_{node}(v_i) + \sum_{i=1}^{k-1} W_{edge}(v_i, v_{i+1})$. The distance of path P is equal to SW^B when each fetch packet FP_i^B is reordered to v_i for $1 \leq i \leq k$.

An optimal solution of the shortest path problem described above can be found by using a modified shortest path algorithm shown in Figure 6. The modified shortest path algorithm is based on the following theorem whose proof is trivial.

Symbol	Meaning
$N_{ins}(FP_i^B)$	The number of instructions in FP_i^B.
$I_j^{FP_i^B}$	The j-th instruction of $FP_i^B \quad (1 \leq j \leq N_{ins}(FP_i^B))$.
$N_{op}(I_j^{FP_i^B})$	The number of operations in $I_j^{FP_i^B}$.
$N_{eq}(I_j^{FP_i^B})$	The number of instructions that are equivalent to $I_j^{FP_i^B}$ $(N_{eq}(I_j^{FP_i^B}) = (N_{op}(I_j^{FP_i^B}))!)$.
$N_{eq}(FP_i^B)$	The number of fetch packets that are equivalent to FP_i^B $(N_{eq}(FP_i^B) = \prod_{j=1}^{N_{ins}(FP_i^B)} N_{eq}(I_j^{FP_i^B}))$.
$FP_{i,n}^B$	The n-th fetch packet in $EQ(FP_i^B)$ $(1 \leq n \leq N_{eq}(FP_i^B))$.

TABLE 2
Notations used in Section IV.C

```
 1: for $i \leftarrow 0$ to $N_{fp}(B)$ {
 2:     /* for each vertex in the level $i+1$ */
 3:     for $k \leftarrow 1$ to $N_{eq}(FP_{i+1}^B)$ {
 4:         $SW_{min} := \infty$;
 5:         /* for each vertex in the level $i$ */
 6:         for $j \leftarrow 1$ to $N_{eq}(FP_i^B)$ {
 7:             $SW_{cur} := d_{P(FP_{i,j}^B)} + W_{edge}(FP_{i,j}^B, FP_{i+1,k}^B)$
 8:                         $+ W_{node}(FP_{i+1,k}^B)$;
 9:             /* find the minimum value */
10:             if ( $SW_{min} > SW_{cur}$ ) {
11:                 $SW_{min} := SW_{cur}$;
12:                 MinNode := j;
13:             }
14:         }
15:         $d_{P(FP_{i+1,k}^B)} := SW_{min}$;
16:         /* store MinNode for the final path construction */
17:         MinPath[$FP_{i+1,k}^B$] := $FP_{i,MinNode}^B$;
18:     }
19: }
```

Fig. 6. A modified shortest path algorithm.

Theorem 1 *Let a path $P(FP_{i,j}^B) = (\mathsf{START}, v_1, \cdots, v_{i-1}, FP_{i,j}^B)$ be the shortest path from* START *to $FP_{i,j}^B \in EQ(FP_i^B)$ and the distance of the path $P(FP_{i,j}^B)$ be $d_{P(FP_{i,j}^B)}$. Then the minimum distance of the path $P(FP_{i+1,k}^B)$=(START$, v_1, \cdots, v_i, FP_{i+1,k}^B)$, $d_{P(FP_{i+1,k}^B)}$, is given by*

$$min_{1 \leq j \leq N_{eq}(FP_i^B)}[d_{P(FP_{i,j}^B)} + W_{edge}(FP_{i,j}, FP_{i+1,k}) + W_{node}(FP_{i+1,k})]. \qquad (6)$$

In Figure 6, SW_{min} is a variable to store the minimum distance of a path from START to $FP_{i+1,k}^B$ (in Line 15) and SW_{cur} is a variable to store the minimum distance of a path from START to $FP_{i+1,k}^B$ that passes through $FP_{i,j}^B$. The shortest path is constructed by visiting $MinPath$ in reverse order. The complexity of the modified shortest path algorithm is given by $O(N_{fp}(B) \cdot (\overline{N_{eq}^{FP^B}})^2)$ where $\overline{N_{eq}^{FP^B}} = \frac{1}{N_{fp}(B)} \sum_{i=1}^{N_{fp}(B)} N_{eq}(FP_i^B)$. $\overline{N_{eq}^{FP^B}}$ is bounded by $N_{op}!$.

V. Global Operation Rearrangement Problem

In the GOR problem, all the basic blocks in a program are simultaneously considered to find a global optimal solution. Since the LOR problem does not take account of inter-block switching activity, simply solving the LOR problem for each basic block does not minimize the number of bit changes for a complete program. In order to compute an optimal solution for the GOR problem, we need additional information on the dynamic behavior of program execution. For example, we should know how many times each basic block is executed, how often each basic block experiences cache misses, and how basic blocks are related each other, etc.

A. GOR Problem Formulation

If a program S is composed of basic blocks $bb_1, bb_2, \cdots, bb_{N_{bb}(S)}$, then the total number of bit changes SW^S from instruction fetches while executing the program S is given as follows, using the notations described in Table 3:

$$SW^S = \sum_{i=1}^{N_{bb}(S)} \sum_{j=1}^{N_{bb}(S)} SW_{BB}^{inter}(bb_i, bb_j) + \sum_{i=1}^{N_{bb}(S)} SW_{BB}^{intra}(bb_i) \qquad (7)$$

where

$$\begin{aligned} &SW_{BB}^{inter}(bb_i, bb_j) \\ &\quad = w(bb_i, bb_j) \cdot SW_{FP}^{inter}(FP_{N_{fp}(bb_i)}^{bb_i}, FP_1^{bb_j}) && (8) \\ &SW_{BB}^{intra}(bb_i) \\ &\quad = w(bb_i) \cdot (\sum_{i=1}^{N_{fp}(bb_i)-1} SW_{FP}^{inter}(FP_i^{bb_i}, FP_{i+1}^{bb_i}) \\ &\quad + \sum_{i=1}^{N_{fp}(bb_i)} SW_{FP}^{intra}(FP_i^{bb_i})) && (9) \end{aligned}$$

$$SW_{FP}^{inter}(FP_n^{bb_i}, FP_m^{bb_j}) = d_{fp}(FP_n^{bb_i}, FP_m^{bb_j}) \qquad (10)$$

$$SW_{FP}^{intra}(FP_n^{bb_i})$$

Symbol	Meaning
$N_{bb}(S)$	The number of basic blocks in a program S.
$w(bb_i, bb_j)$	The expected number of times that a basic block bb_j is executed right after a basic block bb_i.
$w(bb_i)$	The expected number of times that a basic block bb_i is executed.
$MB(FP_n^{bb_i})$	The memory block that contains $FP_n^{bb_i}$.
$FP_j^{MB(FP_n^{bb_i})}$	The j-th fetch packet in the memory block that contains $FP_n^{bb_i}$.
$R_{miss}^{MB(FP_n^{bb_i})}$	The cache miss rate of the memory block $MB(FP_n^{bb_i})$.

TABLE 3
Notations used in Section V.A

Symbol	Meaning
$N_{eq}(bb_i^S)$	The number of basic blocks that are equivalent to bb_i^S ($N_{eq}(bb_i^S) = \prod_{j=1}^{N_{fp}(bb_i^S)} N_{eq}(FP_j^{bb_i^S})$).
$bb_{i,n}^S$	The n-th basic block in $EQ(bb_i^S)$ ($1 \leq n \leq N_{eq}(bb_i^S)$).

TABLE 4
Notations used in Section V.B

$$\begin{aligned} &= \alpha \cdot R_{miss}^{mb} \cdot (\sum_{j=1}^{N_{fp}(MB)} \sum_{k=1}^{N_{op}-1} d_{op}(OP_k^{FP_j^{mb}}, OP_{k+1}^{FP_j^{mb}}) \\ &+ \sum_{j=1}^{N_{fp}(MB)-1} d_{op}(OP_{N_{op}}^{FP_j^{mb}}, OP_1^{FP_{j+1}^{mb}}) \\ &+ d_{op}(\mathbf{1}, OP_1^{FP_1^{mb}}) + d_{op}(OP_{N_{op}}^{FP_{N_{fp}(MB)}^{mb}}, \mathbf{1})) \qquad (11) \\ &\quad (\text{where } mb = MB(FP_n^{bb_i})) \end{aligned}$$

In Equations (8), (9), and (11), $w(bb_i, bb_j)$, $w(bb_i)$ and R_{miss}^{mb} can be calculated by analyzing program execution traces. When a cache miss occurs for a fetch packet F, all the fetch packets in the missed memory block that contains F are fetched from the main memory. Therefore, in Equation (11), all the fetch packets in $MB(FP_n^{bb_i})$ are considered in computing the bit changes at the external instruction bus. We assume that basic blocks are aligned by the cache memory block size. In Equation (10), the hamming distance between the last operation of $FP_n^{bb_i}$ and the first operation of $FP_m^{bb_j}$ is omitted because it is included in Equation (11). Given a program S, the GOR problem is to find an equivalent program S' such that $SW^{S'} \leq SW^{S''}$ for all $S'' \in EQ(S)$.

B. Optimal Solution for GOR

We solve the GOR problem in a similar fashion as the LOR problem by transforming the GOR problem to the shortest path problem. The main difference from the LOR problem is that since a program generally contains branches and loops, a constructed graph may span multiple paths from a given node. In order to utilize the same shortest path algorithm used in solving the LOR problem, we transform the constructed graph so that the new graph has no branches and loops.

Using the notations described in Table 4, given a program S, we construct a weighted directed graph $G_S = \{V, E, W_{node}, W_{edge},\}$, where

$$\begin{aligned} V &= \{\mathsf{START}, \mathsf{END}\} \cup \bigcup_{i=1}^{N_{bb}(S)} EQ(bb_i^S) \\ &= \{\mathsf{START}, \mathsf{END}\} \cup \bigcup_{i=1}^{N_{bb}(S)} \{bb_{i,1}^S, \cdots, bb_{i,N_{eq}(bb_i^S)}^S\}, \\ E &= \{(v,w) \bigm| v = \mathsf{START}, w \in EQ(bb_{Entry}^S)\} \cup \\ &\quad \{(v,w) \bigm| w = \mathsf{END}, v \in \bigcup_{i \in Exit^S} EQ(bb_i^S)\} \cup \\ &\quad \{(v,w) \bigm| v \in EQ(bb_i^S), w \in EQ(bb_j^S) \text{ for } 1 \leq i,j \leq N_{bb}(S) \\ &\qquad \text{where } bb_j^S \text{ is an immediate successor of } bb_i^S \\ &\qquad \text{in a control flow graph}\}, \end{aligned}$$

$$W_{node}(v) = \begin{cases} SW_{BB}^{intra}(v) & \text{if } v \in V - \{\mathsf{START}, \mathsf{END}\} \\ 0 & \text{otherwise} \end{cases} \quad \text{, and}$$

$$W_{edge}(v,w) = \begin{cases} SW_{BB}^{inter}(v,w) & \text{if } v, w \in V - \{\mathsf{START}, \mathsf{END}\} \\ 0 & \text{otherwise} \end{cases}$$

bb_{Entry}^S is the entry point of the program S and the $Exit^S$ contains the indices of basic blocks that are the exit points of the program S.

In order to reduce the computational complexity of the GOR problem, we eliminate $bb_{i,k}^S$ from G_S if there exists $bb_{i,j}^S$ (where $j \neq k$) such that

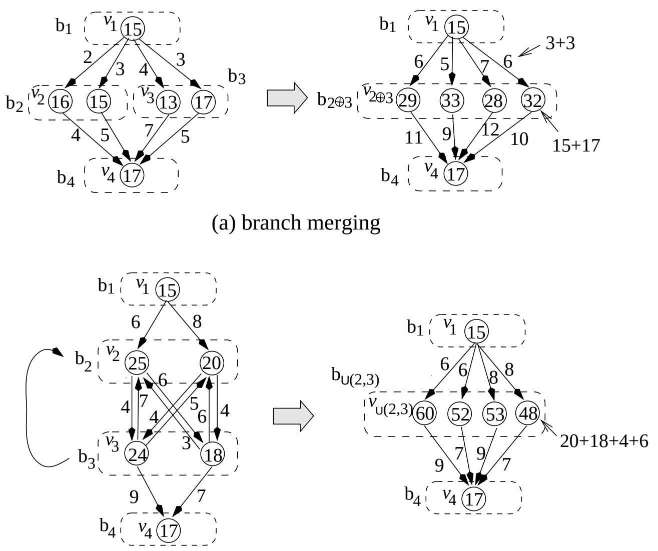
$$FP_1^{bb_{i,j}^S} = FP_1^{bb_{i,k}^S}, \qquad (12)$$

$$FP_{N_{fp}(bb_i^S)}^{bb_{i,j}^S} = FP_{N_{fp}(bb_i^S)}^{bb_{i,k}^S}, \text{ and} \qquad (13)$$

$$SW_{BB}^{intra}(bb_{i,j}^S) \leq SW_{BB}^{intra}(bb_{i,k}^S) \qquad (14)$$

If $bb^S_{i,k}$ satisfies Equations (12), (13), and (14), $bb^S_{i,k}$ cannot be a part of an optimal GOR solution because both $bb^S_{i,j}$ and $bb^S_{i,k}$ have the same SW^{inter}_{BB} value. For each basic block bb^S_i, applying a modified LOR algorithm (with the first and last fetch packets fixed) $N_{eq}(FP^{bb^S_i}_1) \times N_{eq}(FP^{bb^S_i}_{N_{fp}(bb^S_i)})$ times, we can construct a simplified G_S with no redundant $bb^S_{i,k}$'s. Eliminating redundant $bb^S_{i,k}$ from G_S, $N_{eq}(bb^S_i)$ is effectively reduced to $N_{eq}(FP^{bb^S_i}_1) \times N_{eq}(FP^{bb^S_i}_{N_{fp}(bb^S_i)})$. (For the rest of the paper, we use N_i to represent $N_{eq}(FP^{bb^S_i}_1) \times N_{eq}(FP^{bb^S_i}_{N_{fp}(bb^S_i)})$ for a simpler description.)

Once a simplified G_S is constructed, it is further converted to remove branching nodes and looping nodes so that the shortest path algorithm for the LOR problem can be reused. Figure 7 illustrates how the branch merging and loop rolling operations work on the G_S graph using the example control structures.



(a) branch merging

(b) loop rolling

Fig. 7. Effects of branch merging and loop rolling on the G_S graph.

Branch merging replaces two branch successor nodes v_i and v_j with a new node $v_{i\oplus j}$. For the new node $v_{i\oplus j}$, $N_{eq}(bb_{i\oplus j})$ is set to $N_{eq}(bb_i) \times N_{eq}(bb_j)$. For example, in Figure 7.(a), the basic blocks b_2 and b_3 have 2 equivalent basic blocks respectively. After the branch merging operation is applied, $v_{2\oplus 3}$ has 4 equivalent basic blocks. The node v_2 (with $W_{node}(v_2) = 16$) and the node v_3 ($W_{node}(v_3) = 13$) are merged into the node $v_{2\oplus 3}$ whose W_{node} value is 29 (= 16 + 13). The node $v_{2\oplus 3}$ has an edge with v_1 and its edge weight is 6 (= 2 + 4).

After branch merging in Figure 7.(a), three basic blocks b_1,$b_{2\oplus 3}$, and b_4 can be merged into a single basic block using the LOR algorithm. The resulting node has the W_{node} value of 78. We call this extra merging *sequential merging*. If the basic blocks $b_i, \cdots, b_j$ are merged into a single basic block by a sequential merging operation, the merged basic block has $N_{eq}(FP^{bb^S_i}_1) \times N_{eq}(FP^{bb^S_j}_{N_{fp}(bb^S_j)})$ equivalent nodes.

Loop rolling works in a similar fashion as sequential merging. It merges loop body nodes $v_i, \cdots, v_j$ into a new node $v_{\cup(i,\cdots,j)}$ as with sequential merging. The difference is that loop rolling adds the weights of back edges in computing the weight of the merged node. For example, in Figure 7.(b), consider the basic blocks b_2 and b_3 that have two equivalent basic blocks respectively. The nodes v_2 and v_3 are merged into the new node $v_{\cup(2,3)}$ whose W_{node} value is 60 (= 25 + 24 + 4 + 7). The node $v_{\cup(2,3)}$ has an edge with the node v_1, and its edge weight is 6 that is the value of $W_{edge}(v_1, v_2)$.

When nodes v_i, v_j are merged into a new node v' by branch merging or loop rolling, the following changes are made to the G_S graph:

$$
\begin{aligned}
V &= V \cup EQ(v') - EQ(v_i) - EQ(v_j) \\
E &= E \cup \{(v,w) | v \in EQ(v'), w \in EQ(v_k)\} \\
&\quad \cup \{(v,w) | v \in EQ(v_h), w \in EQ(v')\} \\
&\quad - \{(v,w) | v \in EQ(v_i) \cup EQ(v_j), w \in EQ(v_k)\} \\
&\quad - \{(v,w) | v \in EQ(v_h), w \in EQ(v_i) \cup EQ(v_j)\} \\
&\quad - \{(v,w) | v \in EQ(v_i), w \in EQ(v_j)\} \\
&\quad - \{(v,w) | v \in EQ(v_j), w \in EQ(v_i)\} \\
&\quad \text{for all } k \text{ such that } k \neq i, k \neq j, \\
&\quad\ \text{and } ((v_i, v_k) \in E \text{ or } (v_j, v_k) \in E), \text{ and} \\
&\quad \text{for all } h \text{ such that } h \neq i, h \neq j, \\
&\quad\ \text{and } ((v_h, v_i) \in E \text{ or } (v_h, v_j) \in E)
\end{aligned}
$$

$$
\begin{aligned}
W_{node}(v') &= W_{node}(v_i) + W_{node}(v_j) \\
&\quad + W_{edge}(v_i, v_j) + W_{edge}(v_j, v_i) \\
W_{edge}(v', v_k) &= W_{edge}(v_i, v_k) + W_{edge}(v_j, v_k) \\
&\quad \text{for all } k \text{ such that } k \neq i, k \neq j, \\
&\quad\ \text{and } ((v_i, v_k) \in E \text{ or } (v_j, v_k) \in E) \\
W_{edge}(v_h, v') &= W_{edge}(v_h, v_i) + W_{edge}(v_h, v_j) \\
&\quad \text{for all } h \text{ such that } h \neq i, h \neq j, \\
&\quad\ \text{and } ((v_h, v_i) \in E \text{ or } (v_h, v_j) \in E)
\end{aligned}
$$

Once the G_S is converted to a graph with no branches and loops, the shortest path algorithm used for the LOR problem can compute the optimal solution.

C. Heuristic Solution for GOR

Finding an optimal GOR solution using the G_S graph constructed in the previous section may require an excessive amount of memory and cycles. For example, for each basic block b_i, N_i node structures are required. Furthermore, when two basic blocks b_i and b_j are merged using a branch merging operation, the required number of node structures for the merged node increases to $N_i \times N_j$. In this section, we propose a heuristic solution for the GOR problem which we call the GOR-H algorithm.

The GOR-H algorithm reduces the memory requirement and computing cycles significantly by two heuristic rules. First, all the basic blocks are not equally treated. For each basic block bb_i, we associate $FR(bb_i)$ which is defined as follows:

$$FR(bb_i) = \frac{w(bb_i) \cdot N_{fp}(bb_i)}{\sum_{j=1}^{N_{bb}(S)} w(bb_j) \cdot N_{fp}(bb_j)}$$

$FR(bb_i)$ represents an effective fetch rate of the fetch packets in the basic block bb_i over all the basic blocks of a program. Since a basic block with a larger $FR(bb_i)$ value has a bigger effect on the total switching activity during the instruction fetch phase, basic blocks with large $FR(bb_i)$ values are more thoroughly reordered than ones with small $FR(bb_i)$ values.

Second, for each basic block bb_i, not all the equivalent basic blocks in $EQ(bb_i)$ are tried to find an optimal solution. Only N_{cand} equivalent basic blocks are created and included in G_S. These N_{cand} equivalent basic blocks are ones with up to the $N_{cand} - th$ smallest switching activity value among all the basic blocks in $EQ(bb_i)$.

Once the G_S graph is constructed by the two rules above, the rest of processing steps (that is, branch merging, loop rolling and sequential merging) are same as done in the previous section. From the transformed G_S, we can solve the GOR problem using the LOR algorithm.

VI. Experiments

In order to evaluate how well the proposed operation rearrangement technique works on application programs, we have performed experiments using a VLIW digital signal processor, TMS320C6201 [7], from Texas Instruments. The TMS320C6201 is a fixed-point DSP that can specify eight 32-bit operations in a single 256-bit instruction. The TMS320C6201 uses a compressed encoding with b_{cache} = 256. As benchmark programs, various DSP programs were used. The proposed technique was implemented as a separate post-pass tool, which takes as an input an executable file produced by the TI's TMS320C6x optimizing C compiler and produces as an output the rearranged low-power version of the same program.

We have measured the number of bit transitions during the instruction fetch phase for each benchmark program using a switching activity counter. Given an executable file with appropriate input data, a switching activity counter program computes the number of bit transitions from both the internal and external busses during the program execution using instruction address traces. Instruction address traces for benchmark programs were collected by a manual analysis of benchmark source programs.

Table 5 summaries the experimental results with selected DSP benchmark programs. For each benchmark program, the average number of bit transitions per instruction fetch (BT/IF) is computed. For α, we have used 100 [12]. We have compared BT/IF's among TI compiler generated programs (the default column in Table 5), rearranged programs by the proposed LOR technique (the LOR column in Table 5) and the GOR heuristic technique (the GOR-H column in Table 5). We have used 100 for N_{cand} in the GOR heuristic technique.

As shown in Table 5, our operation rearrangement technique reduces the number of bit transitions during the instruction fetch phase on an average by 34.3% compared with the programs generated by the TI compiler. The GOR heuristic technique outperformed the LOR technique by 2.9% more reduction in the switching activity. For many benchmark programs, however, the LOR technique was quite effective, resulting in the almost equivalent switching activity reduction as in the GOR heuristic technique.

VII. Conclusions

In this paper we have described and evaluated an operation rearrangement method during instruction fetches in VLIW machines. The proposed method, which works as a post-pass tool for compiled programs, reorganizes the operation placement orders within VLIW instructions such that the resulting program has the minimum number of bit transitions during instruction fetches. The experimental results show that the proposed rearrangement technique can significantly reduce the switching activity during the instruction fetch phase in VLIW machines. For our benchmark programs, the switching activity was reduced by 34% on an average.

In this paper, we considered the problem of modifying operation orders for *pre-compiled* VLIW programs. However, optimization decisions made during the compilation process can affect the outcome of operation rearrangement. For example, depending on how instructions are scheduled, the number of bit changes during the instruction fetch phase can vary significantly. We plan to investigate the phase-ordering problem between the operation rearrangement and other optimization steps as a next topic.

References

[1] A. Chandrakasan, T. Shung, and R. W. Broderson. Low power CMOS digital design. *IEEE Journal of Solid State Circuits*, 27(4):473–484, 1992.

[2] T. Conte, S. Banerjia, S. Larin, K. N. Menezes, and S. W. Sathaye. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proc. of the 29th IEEE/ACM Int. Symp. on Microarchitecture*, pages 201–211, 1996.

[3] S. Devadas and S. Malik. A survey of optimization techniques targeting low power VLSI circuits. In

Benchmark	Bit transitions/IF			Reduction	
Program	default	LOR	GOR-H	LOR	GOR-H
vector multiply	68.6	46.0	43.7	33.0%	36.3%
FIR8	86.8	59.3	56.7	31.6%	34.6%
FIRcx	79.5	60.6	60.5	23.9%	24.0%
IIR	71.7	52.1	51.7	27.4%	28.0%
lattice analysis	88.4	63.4	58.2	28.3%	34.2%
W_vec	89.5	62.9	57.1	30.0%	36.3%
dotp_sqr	79.2	44.5	44.3	43.9%	44.1%
minerror	50.6	33.2	31.3	34.3%	38.1%
biquad	78.1	54.6	52.3	30.0%	33.0%
Average	76.9	53.0	50.6	31.4%	34.3%

TABLE 5

Experimental results

Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED'97), pages 239–242, 1997.

[4] P. Faraboschi, G. Desoli, and J. A. Fisher. The latest word in digital and media processing. *IEEE Signal Processing Magazine*, 15(2):59–85, 1998.

[5] Fujitsu Microelectronics, Inc. *Fujitsu's new high-performance VLIW processor cores.* http://www.fujitsumicro.com/.

[6] R. Henning and C. Chakrabarti. High-level design synthesis of a low power, VLIW processor for the IS-54 VSELP speech encoder. In *Proc. of Int. Conf. on Computer Design (ICCD'97)*, pages 571–576, 1997.

[7] Texas Instruments. *TMS320C62xx CPU and Instruction Set*, 1997.

[8] Texas Instruments. *TMS320C6000 Power Consumption Summary*, 1999.

[9] A. Klaiber. *The technology behind the Crusoe processor.* Transmeta Corporation White Paper, 2000.

[10] M. T. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans. VLSI Systems*, 5(1):123–135, 1997.

[11] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED'97)*, pages 72–75, 1997.

[12] E. Musoll, T. Lang, and L. Cortadella. Exploiting the locality of memory references to reduce the address bus energy. In *Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED'97)*, pages 202–207, 1997.

[13] J.-M. Puiatti, J. Llosa, C. Piguet, and E. Sanchez. Low-power VLIW processors: A high-level evaluation. In *Proc. of Int. Workshop - Power and Timing Modeling, Optimization and Simulation (PATMOS '98)*, pages 399–408, 1998.

[14] M. R. Stan and W. P. Burleson. Bus-invert coding for low power I/O. *IEEE Trans. on VLSI Systems*, 3:49–58, Mar. 1995.

[15] C. L. Su, C. Y. Tsui, and A. Despain. Low power architectural design and compilation techniques for high-performance processor. In *Proc. of COMPCON94*, pages 489–498, 1994.

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

[17] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. VLSI Systems*, 2(4):437–445, 1994.

[18] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. In *Proc. of Power Driven Microarchitecture Workshop in conjunction with the 25th International Symposium on Computer Architecture (ISCA'98)*, 1998.

[19] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction scheduling for power reduction in processor-based system design. In *Proc. of the 1998 Design Automation and Test in Europe (DATE '98)*, pages 855–860, 1998.