An Operation Rearrangement Technique for Power Optimization in VLIW Instruction Fetch

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 operation rearrangement method that reduces the power consumption from the instruction-fetch datapath. 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.

1 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 [6] from Transmeta (which were developed for mobile Internet computing market) are based on 64 bits or 128 bits VLIW CPU cores. Fujitsu Microelectronics' FR300 [4] (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 [3, 5].

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 [15, 9]. Therefore, in designing VLIW CPU-based mobile devices, low power consumption becomes an important design constraint.

In digital CMOS circuits (that use well-designed logic gates), dynamic power consumption accounts for over 90% of total power consumption [1]. Since the dynamic power consumption is proportionate to the switching activity $\times$ capacitance term, to reduce the overall dynamic power consumption, the switching activity should be reduced from the components having large capacitances. For example, system-level off-chip busses are such components. The power consumption from off-chip driving can reach up to 70% of the total chip power, where bus transitions are the most dominant factor due to the large capacitances of the bus lines [7].

Many techniques have been proposed and developed to reduce the frequency of bit transitions in a system-level off-chip bus [12, 11, 13, 16]. For example, various bus encoding techniques [12, 11] convert bus data representations, exploiting the characteristics of bus access patterns, so that the power consumption from the off-chip bus can be reduced. On the other hand, low-power instruction scheduling techniques [13, 16] modify instruction placement orders to reduce the bit changes from the successive instruction fetches.

In this paper, we propose a post-pass optimization technique that can significantly reduce switching activity from both the internal and external instruction busses of 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] We reduce switching activity by modifying operation placement orders within VLIW instructions so that the switching activity in the instruction-fetch datapath is minimized. The proposed technique also takes into account of the inter-block switching activity, which was ignored in the existing low-power instruction scheduling techniques, in scheduling instructions for low-power fetches.

The main contribution of this paper is two-fold. First, as far as we know, our work is the *first* attempt to reduce the power consumption from the VLIW instruction-

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

fetch datapath[2] by instruction scheduling. Second, the proposed technique, unlike the existing low-power instruction scheduling such as [13, 16], tries to reduce the overall power consumption from both the on-chip and off-chip instruction busses. In reordering instructions for low-power instruction fetch, existing low-power instruction scheduling techniques do not consider simultaneously both the on-chip bus consumption and the off-chip bus consumption. For example, the instruction scheduling technique [16] by Tomiyama *et al.* does not take account of the switching activity at the on-chip instruction bus. For their target processors, since off-chip bus accesses are the dominating power consumer, the authors did not consider to reduce the power consumption from the on-chip bus, which contributes much less to the overall power consumption than the off-chip bus.

On the other hand, in VLIW processors, an instruction scheduling technique must weigh the switching activity from the on-chip instruction bus as well as the off-chip instruction bus. Since the width of the on-chip instruction bus is generally much larger than that of the off-chip instruction bus in VLIW architectures, if an instruction schedule were produced considering only the bit changes from the off-chip bus, it might be a bad (thus more bit-changing) schedule for the on-chip instruction bus. Furthermore, since the on-chip instruction bus has the large wire capacitance as well as the large output load capacitance [17] and it is used every cycle in a high speed, the impact on the total power consumption of the switching activity at the on-chip instruction bus cannot be ignored.

The organization of the rest of the paper is as follows. Before presenting the proposed operation rearrangement technique, we review prior work on low-power techniques for instruction fetch in Section 2. In Section 3, we describe a target VLIW machine model and define several terms. An operation rearrangement technique is explained in Section 4. Experimental results are presented in Section 5 followed by conclusions in Section 6.

2 Related Work

The research to reduce the bit transitions of bus can be classified by three approaches.

The first approach is the bus encoding. Bus-invert coding [12] 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. Shin *et al.* advanced the bus-invert coding by selecting a sub-group of bus lines involved in bus encoding to avoid unnecessary inversion of bus lines not in the sub-group [11].

[2]This includes an external memory, an off-chip bus, an instruction cache and an on-chip bus.

The second approach is the instruction scheduling. Su *et al.* proposed an instruction scheduling technique, called cold scheduling, to reduce the amount of switching activity in the control path [13]. 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. 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 [16]. 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.

The third approach is the instruction encoding. Register relabeling [8] assigns register numbers of instructions so that more frequently consecutive register numbers have a smaller Hamming distance, thus reducing the switching activity of the instruction bus and decode logic. The instruction scheduling and instruction encoding techniques also reduce the switching activity of the instruction fetch and decoding logic.

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 cannot 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.

3 VLIW Machine Model and Definitions

3.1 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 shows compressed encoding method 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. For a compressed VLIW instruction encoding shown in Figures 1.(b) and 1.(c), there are many 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 instruction of *S*, 4! different operation rearrangements are all possible.[3] 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 2. 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:

[3] In Figures 1.(b) and 1.(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.

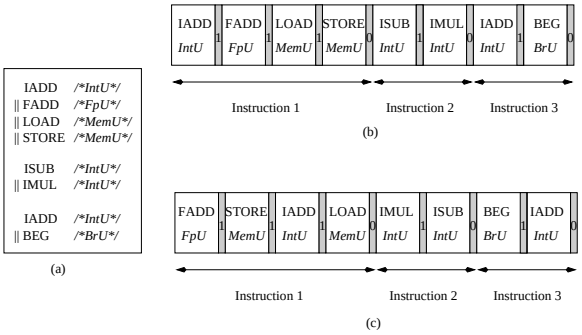

Figure 1. Compressed VLIW instruction encoding; (a) a sample instruction sequence *S*, (b) one compressed encoding of *S* and (c) an alternative encoding of *S*.

- 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.

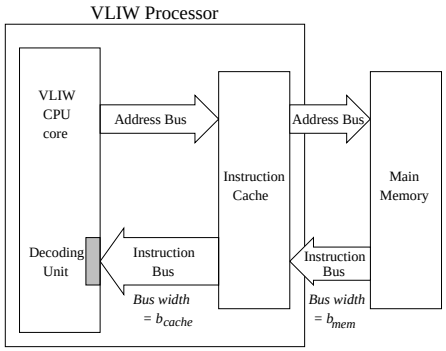

Figure 2. Target system architecture.

3.2 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 $OP^1_{\sigma(i)} = OP^2_i$ for all $1 \leq i \leq n$.

Definition 3 Two fetch packets $FP_1 = (I^1_1, I^1_2, \cdots, I^1_n)$ and $FP_2 = (I^2_1, I^2_2, \cdots, I^2_n)$ are said to be equivalent under operation rearrangement if there exist operation rearrangement functions $(\sigma_1, \sigma_2, \cdots, \sigma_n)$ such that I^1_i is equivalent to I^2_i 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^1_2, $\cdots$, FP^1_n) and $bb_2 = (FP^2_1, FP^2_2, \cdots, FP^2_n)$ are said to be equivalent under operation rearrangement if FP^1_i is equivalent to FP^2_i 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 $S_1 = (bb^1_1, bb^1_2, \cdots, bb^1_n)$ and $S_2 = (bb^2_1, bb^2_2, \cdots, bb^2_n)$ are said to be equivalent under operation rearrangement if bb^1_i is equivalent to bb^2_i 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.

4 Operation Rearrangement Problem

In this section, we introduce the operation rearrangement problem and present its solution, formulating the problem into a shortest path problem.

4.1 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 3. 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 3.(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 3.(b) reduces the total number of bit changes by about 25% from 39 to 29, while maintaining the same semantics of the original sequence.

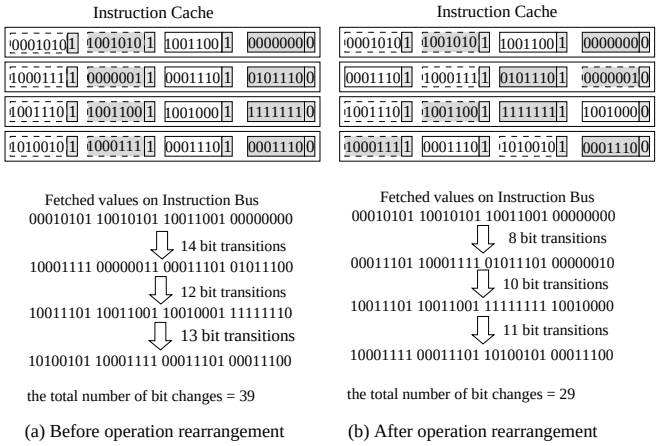

Figure 3. An operation rearrangement example.

4.2 Problem Formulation and Solution

We first consider the operation rearrangement problem for a single basic block where each basic block is assumed to be independent. We call this problem the single basic block (SBB) problem. A complete operation rearrangement problem is solved by extending the solution for the single basic block problem.

4.2.1 Single Basic Block (SBB) Problem

For a given execution of a program P, the total number of bit changes SW^B from a basic block 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. (In the explanations below, we use the basic block B_{eg} shown in Figure 4 as an example. The basic block B_{eg} consists of three cache memory blocks, MB_1, MB_2 and MB_3, and each cache memory block consists of three fetch packets. Each fetch packet consists of four operations.)

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:

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

Symbol	Meaning
$w(B)$	The number of times that a basic block B is executed.
$N_{fp}(B)$	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 $OP_m^{FP_j^B}$.
$MB(FP_i^B)$	The memory block that contains FP_i^B.
N_{miss}^{MB}	The number of cache misses of the memory block MB.

Table 1. Notations used in Section 4.2.1

where $w(B)$ is the number of times that a basic block B is executed. The upper portion of Figure 4 shows how Equation (1) is calculated for the example basic block B_{eg}.

SW_{mem}^B is equal to the sum of all the bit changes between adjacent operation fetches from the main memory because we assumed that $b_{mem} = b_{cache}/N_{op}$ in Section 3.1. For the description purpose, if we assume that basic blocks are aligned by the cache memory block size, and their sizes are the multiple of cache memory block size, SW_{mem}^B can be computed by adding the bit changes of all the memory blocks that consist of the basic block B. For such a memory block MB, if we assume that the memory block has K fetch packets, the number of bit changes SW_{mem}^{MB} from the memory block MB at the external instruction bus is given

$$SW_{mem}^{MB} = \sum_{i=1}^{K} N_{miss}^{MB} \cdot intra(i) + \sum_{i=1}^{K-1} N_{miss}^{MB} \cdot inter(i) \qquad (2)$$

where N_{miss}^{MB} is the number of cache misses of the memory block MB, and

$$intra(i) = \begin{cases} d_{op}(\mathbf{1}, OP_1^{FP_i^{MB}}) + S_{op}(i) & \text{if } (i\%K) = 1 \\ d_{op}(OP_{N_{op}}^{FP_i^{MB}}, \mathbf{1}) + S_{op}(i) & \text{if } (i\%K) = 0 \\ S_{op}(i) & \text{otherwise} \end{cases} \qquad (3)$$

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

$$inter(i) = \begin{cases} 0 & \text{if } (i\%K) = 0 \\ d_{op}(OP_{N_{op}}^{FP_i^{MB}}, OP_1^{FP_{i+1}^{MB}}) & \text{otherwise} \end{cases} \qquad (4)$$

In Equations (3) and (4), FP_j^{MB} represents the i-th fetch packet of the memory block MB. In Equation (3), the number of bit transitions between the **1** vector and the first operation of the memory block and the number of bit transitions between the last operation of the memory block and **1** vector are included in the calculation. This is because we assumed that in Section 3.1, each bus line of the external instruction bus holds a logic 1 value when the bus is not used. The $intra(i)$ and $inter(i)$ terms above can be easily understood with an example. For example, for the first memory block MB_1 of the Figure 4 that consists of three fetch packets, the $intra(i)$ and $inter(i)$ terms are as follows:

$$\begin{aligned} intra(1) &= d_{op}(\mathbf{1}, A) + d_{op}(A, B) + d_{op}(B, C) + d_{op}(C, D) \\ intra(2) &= d_{op}(E, F) + d_{op}(F, G) + d_{op}(G, H) \\ intra(3) &= d_{op}(I, J) + d_{op}(J, K) + d_{op}(K, L) + d_{op}(L, \mathbf{1}) \end{aligned}$$

$$inter(1) = d_{op}(D, E), \quad inter(2) = d_{op}(H, I), \quad inter(3) = 0$$

Since SW_{mem}^B can be computed by summing SW_{mem}^{MB} over

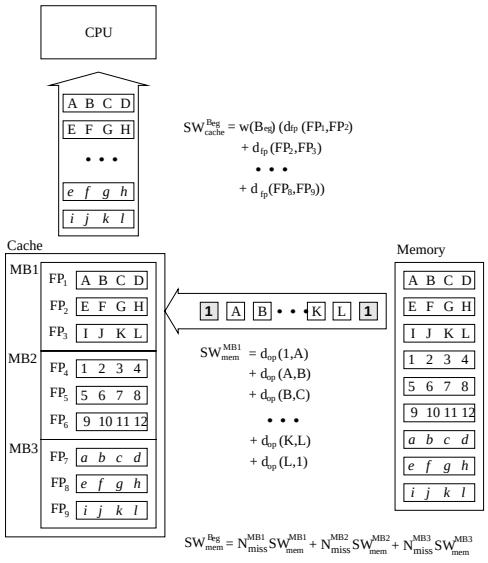

Figure 4. An example calculation of bit transitions at the instruction busses during the execution of a basic block B_{eg}.

all the memory blocks of B, SW_{mem}^B is calculated as follows:

$$\begin{aligned} SW_{mem}^B &= \sum_{i=1}^{N_{fp}(B)} N_{miss}^{MB(FP_i^B)} \cdot intra(i) \\ &+ \sum_{i=1}^{N_{fp}(B)-1} N_{miss}^{MB(FP_i^B)} \cdot inter(i) \end{aligned} \qquad (5)$$

Assuming the load capacitance ratio of the external instruction bus to the internal instruction bus is α, SW^B, in the

number of bit transitions at the internal bus, is computed as follows using the Equations (1) and (2):

$$\begin{aligned} SW^B &= SW^B_{cache} + \alpha \cdot SW^B_{mem} \\ &= \sum_{i=1}^{N_{fp}(B)-1} SW^{inter}_{FP}(FP^B_i, FP^B_{i+1}) \\ &+ \sum_{i=1}^{N_{fp}(B)} SW^{intra}_{FP}(FP^B_i) \end{aligned} \quad (6)$$

where

$$SW^{inter}_{FP}(FP^B_i, FP^B_{i+1}) = w(B) \cdot d_{fp}(FP^B_i, FP^B_{i+1}) + \alpha \cdot N^{MB(FP^B_i)}_{miss} \cdot inter(i) \quad (7)$$

$$SW^{intra}_{FP}(FP^B_i) = \alpha \cdot N^{MB(FP^B_i)}_{miss} \cdot intra(i) \quad (8)$$

Given a basic block B, the SBB 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, $d_{fp}(FP^B_i, FP^B_{i+1})$, $d_{op}(OP^{FP^B_j}_n, OP^{FP^B_j}_{n+1})$ and $d_{op}(OP^{FP^B_j}_{N_{op}}, OP^{FP^B_{j+1}}_1)$ in Equations (1), (3) and (4) are changed.

4.2.2 Solution for the SBB Problem

We compute an optimal solution for the SBB problem by converting the 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

$$\begin{aligned} V &= \{\mathsf{START}, \mathsf{END}\} \cup \bigcup_{i=1}^{N_{fp}(B)} EQ(FP^B_i) \\ &= \{\mathsf{START}, \mathsf{END}\} \cup \bigcup_{i=1}^{N_{fp}(B)} \{FP^B_{i,1}, \cdots, FP^B_{i,N_{eq}(FP^B_i)}\}, \\ E &= \{(v,w) \mid v = \mathsf{START}, w \in EQ(FP^B_1)\} \cup \\ &\quad \{(v,w) \mid w = \mathsf{END}, v \in EQ(FP^B_{N_{fp}(B)})\} \cup \\ &\quad \{(v,w) \mid v \in EQ(FP^B_i), w \in EQ(FP^B_{i+1}) \\ &\qquad \text{for } 1 \leq i < N_{fp}(B)\}, \end{aligned}$$

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

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

Figure 5 shows an example graph G_B constructed by transforming the operation rearrangement problem to the shortest path problem. For each fetch packet FP^B_i, $N_{eq}(FP^B_i)$ vertices are created in G_B, and for successive fetch packets, FP^B_i and FP^B_{i+1}, every pair of $(FP^B_{i,k}, FP^B_{i+1,k'})$ is connected by an edge. We call the $N_{eq}(FP^B_i)$ vertices created from the fetch packet FP^B_i 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^B_i is reordered to v_i for $1 \leq i \leq k$.

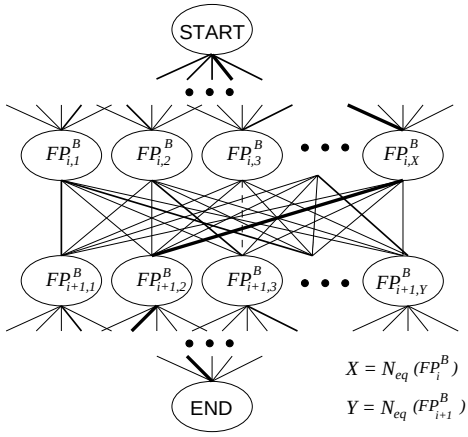

Figure 5. A shortest path problem formulation of the operation rearrangement problem (with node and edge weights omitted).

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.

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

$$\begin{aligned} min_{1 \leq j \leq N_{eq}(FP^B_i)} [d_{P(FP^B_{i,j})} + W_{edge}(FP_{i,j}, FP_{i+1,k}) \\ + W_{node}(FP_{i+1,k})]. \end{aligned} \quad (9)$$

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

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 $(1 \leq j \leq N_{ins}(FP_i^B))$.
$N_{op}(I_j^{FP_i^B})$	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 4.2.2

```
1: for i ← 0 to N_fp(B) {
2:   /* for each vertex in the level i + 1 */
3:   for k ← 1 to N_eq(FP^B_{i+1}) {
4:     SW_min := ∞;
5:     /* for each vertex in the level i */
6:     for j ← 1 to N_eq(FP^B_i) {
7:       SW_cur := d_{P(FP^B_{i,j})} + W_edge( FP^B_{i,j}, FP^B_{i+1,k})
8:                 + W_node(FP^B_{i+1,k});
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^B_{i+1,k})} := SW_min;
16:    /* store MinNode for the final path construction */
17:    MinPath[FP^B_{i+1,k}] := FP^B_{i,MinNode};
18:  }
19: }
```

Figure 6. A modified shortest path algorithm.

4.2.3 Operation Rearrangement Problem for Whole Program

The operation rearrangement solution for the SBB problem described above does not take account of inter-block switching activity. Therefore, simply solving the SBB problem for each basic block does not minimize the number of bit changes for a whole program. In order to find a global (thus better) solution for the complete program, we need additional information on the dynamic behavior of program execution as well as ones required for the SBB problem. For example, we should know how branches are resolved in run time to compute the relative adjacency frequency between two basic blocks.

The solution of the operation rearrangement problem for a whole program can be solved in a similar fashion on the SBB problem by transforming the problem to the shortest path problem. The main difference is that in the whole program, because of branches and loops, nodes in a constructed graph for a shortest path problem formulation may span multiple paths from a given node. We use two techniques, branch merging and loop rolling [10] to convert the graph with no branches and loops. Once the graph is converted to have no branches and loops, the shortest path algorithm for the SBB problem can be used to find a global solution [10].

5 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 [14], 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 global solution 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.

Table 3 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 between TI compiler generated programs (the default column in Table 3), and rearranged programs by the proposed operation rearrangement technique (the ORT column in Table 3).

As shown in Table 3, 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.

Benchmark Program	Bit transitions/IF		
	default	ORT	Reduction
vector multiply	68.6	43.7	36.3%
FIR8	86.8	56.7	34.6%
FIRcx	79.5	60.5	24.0%
IIR	71.7	51.7	28.0%
lattice analysis	88.4	58.2	34.2%
W_vec	89.5	57.1	36.3%
dotp_sqr	79.2	44.3	44.1%
minerror	50.6	31.3	38.1%
biquad	78.1	52.3	33.0%
Average	76.9	50.6	34.3%

Table 3. Experimental results

6 Conclusions

In this paper we have described and evaluated an operation rearrangement method for power optimization in instruction fetches of 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 reduce the switching activity significantly from the complete instruction-fetch datapath of VLIW machines. For our benchmark programs, the switching activity was reduced by 34% on an average.

The work described in this paper can be extended in several directions. One of important future tasks is to quantify the *real* energy gains, not the simulated ones, from using the proposed technique. We are currently building a cycle-accurate measurement-based power profiling tool for an embedded microprocessor. If this tool works as expected, we plan to extend it for VLIW processors in the future.

Although we focused on VLIW processors in this paper, a similar operation rearrangement technique can be effective for low-power instruction fetches in superscalar processors. The preliminary result using a four-way superscalar processor suggests that the *total* processor energy can be reduced by about 7%. We are currently implementing a modified operation rearrangement algorithm for a superscalar processor.

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 research 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] 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.

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

[5] 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*, pages 571–576, 1997.

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

[7] D. Liu and C. Svensson. Power consumption estimation in CMOS VLSI chips. *IEEE Journal of Solid State Circuits*, 29(6):663–670, 1994.

[8] 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*, pages 72–75, 1997.

[9] 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*, pages 399–408, 1998.

[10] D. Shin and J. Kim. A global operation rearrangement technique for low-power instruction fetch. Technical Report SNU-CSE-AE-99-001, Computer Architecture and Embedded Systems Laboratory, Seoul National University, 1999.

[11] Y. Shin, S. Chae, and K. Choi. Partial bus-invert coding for power optimization of system level bus. In *Proc. of Int. Symp. on Low Power Electronics and Design*, pages 127–129, 1998.

[12] 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.

[13] 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.

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

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

[16] 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*, pages 855–860, 1998.

[17] Y. Zhang, R. Y. Chen, W. Ye, and M. J. Irwin. System level interconnect power modeling. In *Proc. of the 11th international ASIC Conference*, pages 289–293, 1998.