Switching-Activity Minimization on Instruction-level Loop Scheduling for VLIW DSP Applications *

Zili Shao Qingfeng Zhuge Meilin Liu Bin Xiao Edwin H.-M. Sha
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083, USA
{zxs015000, qfzhuge, mxl024100, bxiao, edsha}@utdallas.edu

Abstract

This paper develops an instruction-level loop scheduling technique to reduce both execution time and bus switching activities for applications with loops on VLIW architectures. We propose an algorithm, SAMLS (Switching-Activity Minimization Loop Scheduling), to minimize both schedule length and switching activities for applications with loops. In the algorithm, we obtain the best schedule from the ones that are generated from an initial schedule by repeatedly rescheduling the nodes with schedule length and switching activities minimization based on rotation scheduling and bipartite matching. The experimental results show that our algorithm can greatly reduce both schedule length and bus switching activities compared with the previous work.

1: Introduction

In order to satisfy ever-growing requirements for high performance DSP (Digital Signal Processing), VLIW (Very Long Instruction Word) architecture is widely adapted in high-end DSP processors. A VLIW processor has multiple functional units (FUs) and can process several instructions simultaneously. While this multiple-FU architecture can be exploited to increase instruction-level parallelism and improve time performance, it causes more power consumption. In embedded systems, high performance DSP needs to be performed not only with high data throughput but also with low power consumption. Therefore, it becomes an important problem to reduce the power consumption of a DSP application with the optimization of time performance on VLIW processors. Since loops are usually the most critical sections and consume a significant amount of power and time in a DSP application, in this paper, we address loop optimization problem and develop an instruction-level loop scheduling technique to minimize both power consumption and execution time of an application on VLIW processors.

We focus on reducing the power consumption of applications on VLIW architectures by reducing transition activities on the instruction bus. Due to large capacitance and high transition activities, buses consume a significant fraction of total power dissipation in a processor. For example, buses in DEC Alpha 21064 processor dissipate more than 15% of the total power consumption, and buses in Intel 80386 processor dissipate more than 30% of the total[5]. In this paper, we study bus switching-activity reduction problem from compiler level by instruction-level scheduling. Using instruction-level scheduling to reduce bus switching activities can be considered as an extension of the low

*This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, NSF CCR-0309461, USA, and HK POLYU A-PF86 and COMP 4-Z077, HK.

power bus coding techniques [14, 15] at compiler level. In a VLIW processor, an instruction word that is fetched onto the instruction bus consists of several instructions. So we can "encode" each long instruction word to reduce bus switching activities by arranging the locations and sequence of instructions of an application. A VLIW processor usually has a big number of instruction bus wires so that it can fetch several instructions simultaneously. Therefore, we can greatly reduce power consumption by reducing switching activities on the instruction bus.

In recent years, people have addressed the issue to reduce power consumption by software arrangement at instruction level[17, 7]. Most of work in instruction scheduling for low power focuses on DAG (Directed Acyclic Graph) scheduling. They study the minimization of switching activities considering different problems such as register assignment problem[1], I-cache data bus[19], etc.

For VLIW architectures, low-power related instruction scheduling techniques have been proposed in [7, 10]. In most of these work, the scheduling techniques are based on traditional list scheduling in which applications are modeled as Directed Acyclic Graph and only intra-iteration dependencies are considered. In this paper, we show we can significantly improve both the power consumption and time performance for applications with loops on VLIW architectures by carefully exploiting inter-iteration dependencies.

Several loop optimization techniques have been proposed to reduce power variations of applications. Yun and Kim [20] propose a power-aware modulo scheduling algorithm to reduce both the step power and peak power for VLIW processors. Yang et al. [18] propose an instruction scheduling compilation technique to minimize power variation in software pipelined loops. A schedule with the minimum power variation may not be the schedule with the minimum total energy consumption nor a schedule with the minimum length. This paper focuses on developing efficient loop scheduling techniques to reduce both schedule length and switching activities so as to reduce the energy consumption of an application.

Our work is related to [12, 13, 7]. In [12, 13], Shin et al. propose a post-pass optimal operation rearrangement method for VLIW instruction fetch to reduce bus switching activities by converting the problem to the shortest path problem. In [7], Lee et al. propose an instruction scheduling technique to further reduce bus switching activities on VLIW architectures by horizontally rearranging operations using bipartite matching and vertically rescheduling operations using a heuristic algorithm. In these work, applications are represented as Directed Acyclic Graph (DAG). This paper shows that we can further significantly reduce both bus switching activities and schedule length for applications with loops on VLIW processors. Compared with the technique in[7] that optimizes the DAG part of a loop, our technique shows an average 21.1% reduction in swithing activities and an average 14.4% reduction in schedule length. One of our basic ideas is to exploit inter-iteration dependencies of a loop which is also known as software pipelining[6, 3]. However, the traditional software pipelining such as rotation scheduling[3], etc., is performance-oriented and does not consider switching activities reduction. Therefore, we propose a loop scheduling approach that optimizes both the schedule length and bus switching activities based on rotation scheduling.

In our previous work [11], we prove that the loop scheduling problem with minimum latency and minimum switching activities is NP-complete with or without resource constraints, and propose a heuristic algorithm to reduce switching activities and schedule length based on a greedy strategy in which each node in a rotation node set is considered separately and re-scheduled to the best location one by one. The algorithm based on this strategy may not give good results for some applications. Thus, this paper proposes a better algorithm, SAMLS (Switching-Activity Minimization Loop Scheduling), based on rotation scheduling and bipartite matching.

In SAMLS, we use a strategy in which all nodes in a rotation node set are considered together and re-scheduled base on a best matching obtained by constructing a weighted bipartite matching

between nodes and empty locations. And we select the best schedule among the ones that are generated from a given initial schedule by repeatedly rescheduling the nodes with schedule length and switching activities minimization. SAMLS can be applied to various VLIW architectures. We experiment with our algorithm on a set of benchmarks. The experiments are performed on a VLIW architecture similar to that in [7] using the real TI C6000 instructions. The experimental results show that our algorithm can greatly reduce both bus switching activities and schedule length compared with the previous work.

The remainder of this paper is organized as follows. In Section 2, we give the basic definitions and concepts used in the rest of the paper. The SAMLS algorithm is presented in Section 3. Experimental results and concluding remarks are provided in Section 4 and Section 5, respectively.

2: Basic Models and Concepts

In this section, we introduce basic models and concepts that will be used in the later sections. We first introduce the target VLIW architecture and cost model. Then we explain how to use cyclic DFG to model loops. Next we introduce the static schedule and define the switching activities of a schedule. Finally, we introduce the basic concepts of rotation scheduling.

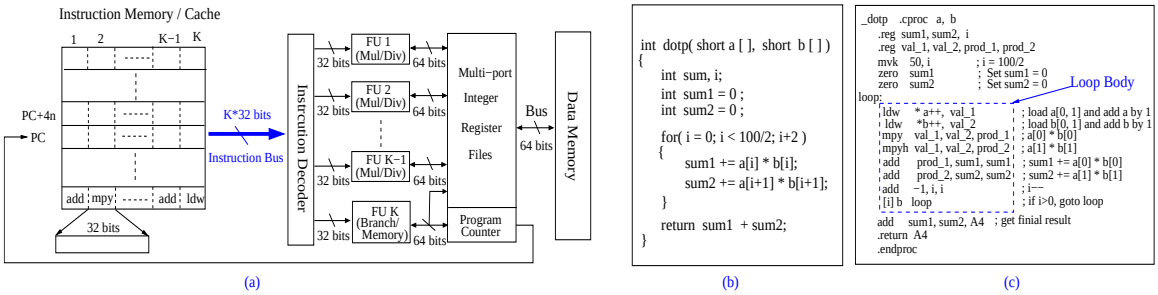


Figure 1. (a) The target VLIW architecture and bus models. (b-c) A dot-product C code and its corresponding assembly code from TI C6000[16].

The abstract VLIW machine is shown in Figure 1(a) that has the similar architecture as that in [7]. In this VLIW architecture, a long instruction word consists of K instructions and each instruction is 32-bit long. In each clock cycle, a long instruction word is fetched to the instruction decoder through a 32×K-bit instruction bus and correspondingly executed by K FUs. The first (K-1) FUs, FU_1 through FU_{K-1}, are integer ALUs that can do integer addition, multiplication, division, and logic operation. The Kth FU, FU_K, can do branch/flow control and load/store operation in addition to all the other operations. A long instruction word can contain one load/store instruction (or branch/flow control) and $K-1$ integer arithmetic/logic instructions. Or it can contain K integer arithmetic/logic instructions. The program that consists of long instruction words is stored in the instruction memory or cache. Memory addressing is byte-based. This architecture is used in our experiments. We obtain the results when K equals to 4.

Hamming distance is used to to estimate switching activities in the instruction bus. Given two binary strings, hamming distance is the number of bit difference between them. Let X=$(x_1, x_2, \cdots, x_K)$ and Y=$(y_1, y_2, \cdots, y_k)$ be two consecutive instruction words in which x_i and y_i denote the instructions at location i of X and Y, respectively. Then the bus switching activities caused by fetching Y immediately after X in the instruction bus is:

$$H(X,Y) = \sum_{i=1}^{K} h(Binary\_String(x_i), Binary\_String(y_i)), \quad (1)$$

where $Binary\_String(x_i)$ is the function to map instruction x_i to its binary code, and h is the hamming distance between $Binary\_String(x_i)$ and $Binary\_String(y_i)$.

Cyclic Data Flow Graph (DFG) is used to model loops. A *cyclic DFG* $G = \langle V, E, d, t,$ Binary_String$\rangle$ is a node-weighted and edge-weighted directed graph, where V is the set of nodes and each node denotes an instruction of a loop, $E \subseteq V \times V$ is the edge set that defines the data dependency relations for all nodes in V, $d(e)$ represents the number of delays for any edge $e \in E$, $t(u)$ represents the computation time for any node $u \in V$, and $Binary\_String(u)$ is a function to map any node $u \in V$ to its binary code. Nodes in V are instructions in a loop. The computation time of each node is the computation time of the corresponding instruction. The edge without delay represents the intra-iteration data dependency; the edge with delays represents the inter-iteration data dependency and the number of delays represents the number of iterations related to the dependency. The *cycle period* of a DFG corresponds to the minimum schedule length of one iteration of the loop when there is no resource constraint.

We use a real loop application, a dot-product program, to show how to use cyclic DFG to model a loop. A program to compute the dot-product of two integer arrays is shown in Figure 1-(b) and its corresponding assembly code from TI C6000 [16] is shown in Figure 1-(c). Our focus is the loop body. Basically, in the loop body in Figure 1-(c), 64-bit data are first loaded into registers by instruction LDW. Then the multiplications are done by instruction MPY and $MPYH$ for low 32 bits and high 32 bits, respectively. Finally, the summations are done by instruction ADD. To model the loop body in Figure 1-(c), the mapping between the node and instruction is shown in Figure 2-(a) and the corresponding cyclic DFG is shown in Figure 2-(b).

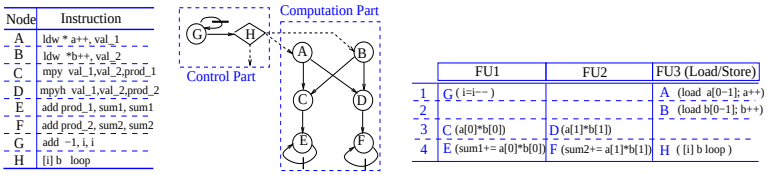


Figure 2. (a) The nodes and their corresponding instructions. (b) The cyclic DFG that represents the loop body in Figure 1-(c). (c) The schedule generated by the list scheduling.

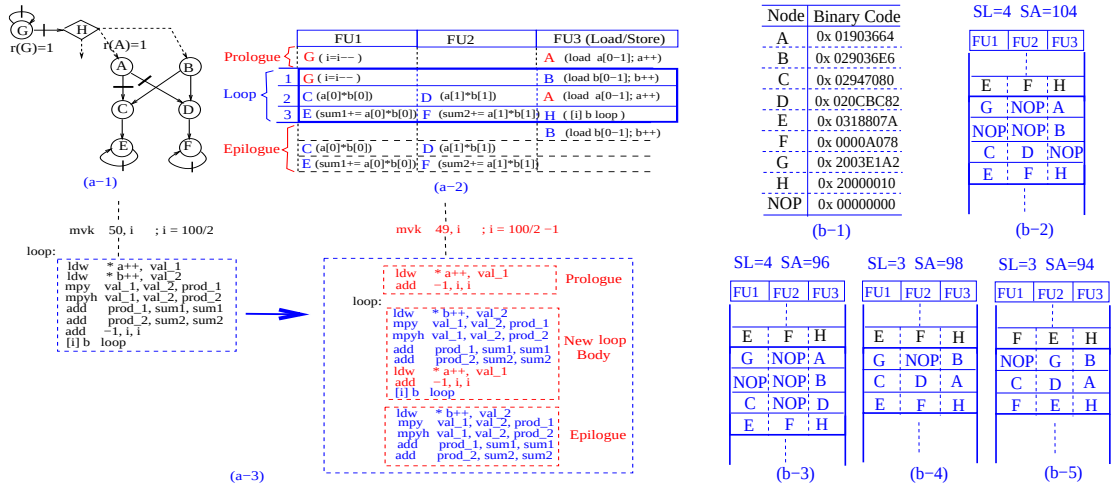
A *static* schedule of a cyclic DFG is a repeated pattern of an execution of the corresponding loop. In our work, a schedule implies both control step assignment and allocation. A static schedule must obey the dependency relations of the DAG portion of the DFG. The DAG is obtained by removing all edges with delays in the DFG. Assume that we want to schedule the DFG in Figure 2-(b) to the target VLIW architecture with 3 FUs ($K = 3$). And let functional unit FU_1 and FU_2 be the integer-ALU, and FU_3 be the load/store/branch Unit. The static schedule generated by the list scheduling is shown in Figure 2-(c). We use (i, j) to denote the location of a node in a schedule, where i is the row and j is the column. For example, the location of node B is $(2, 3)$ in the schedule shown in Figure 2-(c).

The switching activities of a static schedule for a DFG are defined as the summation of the switching activities caused by all long instruction words of a schedule in one iteration on the instruction bus. It can be calculated by counting bit switches on the instruction bus when one iteration of a loop is executed. Since the static schedule is repeatedly executed for a loop, the binary code of the last long instruction word fetched onto the instruction bus in the previous iteration is set as the initial value of the instruction bus in the current iteration. The switching activities caused in the first iteration may be different from other iterations, since the initial state on the instruction bus for the first iteration may be different. A loop is usually executed many times so the influence of the first iteration is very small to the average switching activities of a schedule. Therefore, we use

the switching activities of any iteration except the first one to denote the switching activities of a schedule.

Considering inter-iteration dependencies, retiming and rotation are two optimization techniques for the scheduling of cyclic DFGs. *Retiming* [8] can be used to optimize the cycle period of a cyclic DFG by evenly distributing the delays in it. It generates the optimal schedule for a cyclic DFG when there is no resource constraints. Given a cyclic DFG $G = \langle V, E, d, t \rangle$, retiming r of G is a function from V to integers. For a node $u \in V$, the value of $r(u)$ is the number of delays drawn from each of incoming edges of node u and pushed to all of the outgoing edges. Let $G_r = \langle V, E, d_r, t \rangle$ denote the retimed graph of G with retiming r, then $d_r(e) = d(e) + r(u) - r(v)$ for every edge $e(u \rightarrow v) \in V$ in G_r.

Rotation Scheduling [3] is a scheduling technique used to optimize a loop schedule with resource constraints. It transforms a schedule to a more compact one iteratively. In most cases, the minimal schedule length can be obtained in polynomial time by rotation scheduling. In each rotation, the nodes in the first row of the schedule are rotated down to the earliest possible available locations. In this way, the schedule length can be reduced. From the retiming point of view, these nodes get retimed once by drawing one delay from each of incoming edges of the node and adding one delay to each of its outgoing edges in the DFG. The new locations of the nodes in the schedule must also obey the dependency relations in the new retimed graph.



(a) (b)

Figure 3. a.(1) The retimed graph. (2) The schedule after the first Rotation (3) The corresponding transformation for the loop body. **b.**(1) The nodes and TI C6000 machine code. (2) The schedule generated by the list scheduling. (3) The schedule generated by the algorithms in[7]. (4) The schedule generated by rotation scheduling. (5) The schedule generated by our technique.

Using the schedule generated by the list scheduling in Figure 2-(b) as an initial schedule, we give an example in Figure 3(a) to show how to rotate the nodes in the first row (node G and A) to generate a more compact schedule. The retimed graph is shown in Figure 3(a)-(1) and the schedule after the first rotation is shown in Figure 3(a)-(2). The schedule length is reduced to 3 after the first row is rotated. From the program point of view, rotation scheduling regroups a loop body and attempts to reduce intra-dependencies among nodes. For example, after the first rotation is performed, a new loop is obtained by the transformation as shown in Figure 3(a)-(3), in which the corresponding instructions for node G and A are rotated and put at the end of the new loop body above the branch instruction H. And one iteration of the old loop is separated and put outside the

new loop body: the instructions for G and A are put in the prologue and those for the other nodes are put in the epilogue. In the new loop body, G and A perform the computation for the $(i+1)$th iteration when the other nodes do the computation of the i_{th}. The transformed loop body after the rotation scheduling can be obtained based on the retiming values of nodes[2]. The code size is increased by introducing the prologue and epilogue after the rotation is performed. This problem can be solved by the code size reduction technique proposed in [21].

We use the real machine code from TI C6000 instruction set for this dot-product program and compare schedule length and bus switching activities of the schedules generated by various techniques. The nodes and their corresponding binary code are shown in Figure 3(b)-(1), and the schedules are shown in Figure 3(b)-(2-5) in which "SA" denotes the switching activities of the schedule and "SL" denotes the schedule length. Among them, the schedule generated by our algorithm shown in Figure 3(b)-(5) has the minimal bus switching activities and the minimal schedule length.

3: Switching-Activity Minimization Loop Scheduling

In [11], the loop scheduling problem with minimum latency and minimum switching activities is proved to be NP-complete with or without resource constraints. In this section, we propose a heuristic algorithm, SAMLS (Switching-Activity Minimization Loop Scheduling), to reduce both schedule length and switching activities for applications with loops. The SAMLS algorithm is shown in Figure 4 and its two key functions are discussed in Figure 5 and 8, respectively.

Input: DFG $G = \langle V, E, d, t, Binary\_String \rangle$, the retiming function r of G, an initial schedule S of G, the rotation times N.
Output: A new schedule S' and a new retiming function r'.
Algorithm:

1. **for** i=1 to N {
   (a) Put all nodes in the first row in S into a set R. Retiming each node $u \in R$ by $r(u) \leftarrow r(u) + 1$. Delete the first row from S and shift S up by one control step.
   (b) Reschedule the nodes in R by calling function BipartiteMatching_NodesSchedule(G,r,S,R) (Figure 5).
   (c) Minimize the switching activities of S row by row by calling function RowByRow_BipartiteMatching(S) (Figure 8).
   (d) Store the obtained schedule and retiming function by $S_i \leftarrow S$ and $r_i \leftarrow r$.
   }
2. Select S_j from $S_1, S_2, \cdots, S_N$ such that S_j has the minimum switching activities among all minimum-latency schedules. Output results: $S' \leftarrow S_j$ and $r' \leftarrow r_j$.

Figure 4. Algorithm SAMLS.

The SAMLS Algorithm The basic idea is to obtain a better schedule by repeatedly rescheduling the nodes based on the rotation scheduling with schedule length and switching activities minimization. As shown in Figure 4, in SAMLS, we first generate N schedules based on a given initial schedule and then select the one with the minimum switching activities among all minimum-latency schedules, where N is an input integer to determine the rotation times. These N schedules are obtained by repeatedly rescheduling the nodes in the first row to new locations based on the rotation scheduling with schedule length and switching activities minimization. Two functions, BipartiteMatching_NodesSchedule() and RowByRow_BipartiteMatching(), are used to generate a new schedule. BipartiteMatching_NodesSchedule() is used to reschedule the nodes in the first row to new

locations to minimize schedule length and switching activities. RowByRow_BipartiteMatching() is then used to further minimize the switching activities of a schedule by performing a row-by-row scheduling. The implementation of these two key functions are shown in Figure 5 and Figure 8 below.

BipartiteMatching_NodesSchedule() BipartiteMatching_NodesSchedule() is shown in Figure 5. The basic idea is to construct a weighted bipartite graph between the nodes and the empty locations and reschedule the nodes based on the obtained minimum cost matching.

Input: DFG $G = \langle V, E, d, t, Binary\_String \rangle$, the retiming r of G, a schedule S, and a node set R.
Output: The revised schedule S.
Algorithm:

1. $Len \leftarrow$ the schedule length of S.
2. **while** (R is not empty) **do** {
   (a) Group all empty locations of S into blocks and let B be the set of all blocks. If B is empty, then let $Len \leftarrow Len + 1$; **Continue**.
   (b) Construct a weighted bipartite graph G_{BM} between node set R and block set B. $G_{BM} = \langle V_{BM}, E_{BM}, W \rangle$ in which: $V_{BM} = R \cup B$; for each $u \in R$ and $b_i \in B$, if u can be put into Block b_i, then $e(u, b_i)$ is added into E_{BM} with weight $W(e(u, b_i)) = Switch\_Block(u, b_i)$.
   (c) If E_{BM} is empty, then let $Len \leftarrow L + 1$; **Continue.**
   (d) Get the minimum cost maximum match M by calling function Min_Cost_Bipartite_Matching(G_{BM}).
   (e) Find edge $e(u, b_i)$ in M that has the minimal weight among all edges in M.
   (f) Assign u into the earliest possible location in Block b_i and remove u from set R.
   }

Figure 5. Algorithm BipartiteMatching_NodesSchedule().

In BipartiteMatching_NodesSchedule(), we construct a weighed bipartite graph between the nodes and the blocks. A block is a set that contains the consecutive empty locations in a column of a schedule. For example, for the schedule in Figure 6, there are 2 blocks: $Block_1 = \{(2,1),(3,1),(4,1)\}$ and $Block_2 = \{(1,2),(5,2)\}$. Location (1,2) and (5,2) are consecutive when we consider that the schedule is repeatedly executed as shown in Figure 6-(b). We do not construct a bipartite graph directly between the nodes and the empty locations, since the matching obtained from such bipartite graph may not be a good one in terms of minimizing switching activities. For example, in Figure 6, assume two nodes X and Y are matched to two consecutive locations (2,1) and (3,1) in a best matching that is obtained from a weighted bipartite graph constructed directly between the nodes and the empty locations. Since the switching activities caused by X and Y (they are next to each other) are not considered, the actual switching activities may be more than the number we expect and the matching may not be the best. Instead, we construct the bipartite graph between the nodes and the blocks. In such a way, we can obtain a matching shown below in which at most one node can be put into a block.

The weighted bipartite graph between the nodes and the blocks, $G_{BM} = \langle V_{BM}, E_{BM}, W \rangle$, is

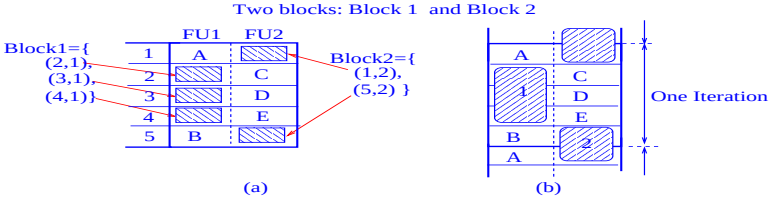


Figure 6. (a) A given schedule. (b) Two blocks that contain consecutive empty locations in a column.

constructed as follows: $V_{BM} = R \cup B$ where R is the rotated node set and B is the set of all blocks. For each node $u \in R$ and each block $b_i \in B$, if u can be put into at least one location in block b_i, an edge $e(u, b_i)$ is added into E_{BM} and $W(e(u, b_i)) = Switch\_Block(u, b_i)$. Function $Switch\_Block(u, b_i)$ computes the switching activities when u is put into b_i. Assume that u' and u'' are the corresponding nodes in the locations immediately above and below the earliest location that u can be put in b_i in the same column, then $Switch\_Block(u, b_i)$ is computed by:

$$Switch\_Block(u, b_i) = H(u, u') + H(u, u'') - H(NOP, u') - H(NOP, u'') \quad (2)$$

$Switch\_Block(u, b_i)$ is the switching activities caused by replacing NOP with u.

After G_{BM} is constructed, $Min\_Cost\_Bipartite\_Matching$ is called to obtain a minimum weight maximum bipartite matching M of G_{BM}. Since we set the switching activities as the weight of edges in G_{BM}, the schedule based on M will cause the minimum switching activities. We find the edge $e(u, b_i)$ that has the minimum weight in the matching and schedule u to the earliest location in b_i. We only schedule one node from the obtained matching each time. Since more blocks may be generated after u is scheduled, other nodes may find better locations in the next matching. In this way, we also put the nodes into the empty locations as many as possible without increasing the schedule length. Therefore, both the schedule length and switching activities can be reduced by this strategy.

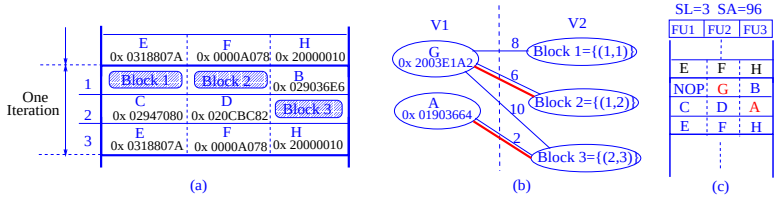


Figure 7. (a) The schedule with the first row removed. (b) The weighted bipartite graph. (c) The obtained schedule.

Using the schedule generated by the list scheduling in Figure 2-(c) as an initial schedule, we give an example in Figure 7 to show how to reschedule the nodes in the first row by SAMLS. The schedule with the first row removed is shown in Figure 7-(a), and the constructed weighted bipartite graph is shown in Figure 7-(b). The weights of edges in Figure 7 are obtained using Equation 2 shown above. For example, the weight of the edge between G and $Block\_1$ is calculated by: H(G,E)+H(G,C)-H(NOP,E)-H(NOP,C)=14+12-10-8=8. The obtained matching is M=$\{(G, Block\_2), (A, Block\_3)\}$. Based on SAMLS, node A is scheduled to location (2,3) since $e(A, Block\_3)$ has the minimal weight in the matching. Similarly, node G is scheduled to location (1,2) in the second iteration. The final schedule is shown in Figure 7-(c).

RowByRow_BipartiteMatching() In RowByRow_BipartiteMatching(), we horizontally schedule nodes in each row to further reduce switching activities. The algorithm is similar to the horizontal scheduling in[7]. However, two differences need to be considered. First, every row in the schedule can be regarded as the initial row in terms of minimizing switching activities, since we deal with cyclic DFG and the static schedule can be regarded as a repeatedly-executed cycle. Second, when processing the last row, we need to not only consider the second to the last row but also the first row in the next iteration, since both of them are fixed at that time. RowByRow_BipartiteMatching() is shown in Figure 8. After running RowByRow_BipartiteMatching() with the schedule shown in Figure 7-(c) as the input, we obtain the finial schedule shown in Figure 3(b)-(5) in Section 2.

Discussion and Analysis As we show in the algorithm, SAMLS can be applied to various VLIW architectures if architecture-related constraints are considered in constructing the weighted bipartite

Input: A schedule S.
Output: The revised schedule S with switching activities minimization.
Algorithm:

1. $Len \leftarrow$ the schedule length of S and $Col \leftarrow$ the number of columns of S.
2. Let BS[Col] be a binary string array and BS[Col]={BS[1],BS[2],$\cdots$,BS[Col]}. And let Init_BS[Col] be another binary string array and Init_BS[Col]={Init_BS[1],Init_BS[2],$\cdots$,Init_BS[Col]}.
3. **for** i=1 to Len {
   (a) $S_i \leftarrow S$.
   (b) Set BS[k]=Init_BS[k]=Binary_String($S_i(1,k)$) for $k = 1, 2, \cdots, Col$, where $S_i(1,k)$ denotes the node at location (1,k) in schedule S_i.
   (c) **for** j=2 to Len {
   - $R \leftarrow$ All nodes in Row j in S_i.
   - Construct a weighted bipartite graph G_{BM} between node set R and location set $\{1, 2, \cdots, Col\}$. $G_{BM} = \langle V_{BM}, E_{BM}, W \rangle$ in which: $V_{BM} = R \cup \{1, 2, \cdots, Col\}$; for each $u \in R$ and $k \in \{1, 2, \cdots, Col\}$, $e(u,k)$ is added into E_{BM} and $W(e(u,k))$ is set as follows: If j<Len, then W(e(u,k))=h(Binary_String(u),BS[k]); otherwise,W(e(u,k))=h(Binary_String(u),BS[k])+h(Binary_string(u),Init_BS[k]).
   - $M \leftarrow$ Min_Cost_Bipartite_Matching(G_{BM}).
   - Put u into location (j,k) in S for each edge $e(u,k) \in M$.
   - Set BS[k]=Binary_String($S_i(j,k)$) for $k = 1, 2, \cdots, Col$.

   }
   (d) Rotate down the first row of S by putting it into the last row.

   }
4. Select S_j from $S_1, S_2, \cdots, S_{Len}$ where S_j has the minimum switching activities. Output S_j.

Figure 8. Algorithm RowByRow_BipartiteMatching.

graphs. In the algorithm, we select the best schedule from the generated N schedules. N should be selected to satisfy that max_r is less than the given loop count where $max_r = \max_{u \in V} r(u)$ in a rotated graph[2]. In the experiments, we found that the rotation times to generate the best schedules for various benchmarks are around $1 * Sch\_Len$, where Sch_Len is the schedule length of the corresponding initial schedule. Loops are usually executed many times in computation-intensive DSP applications, so N can be selected as $(5 \sim 10) * Sch\_Len$ to guarantee that a good result can be obtained while the requirement for max_r can still be satisfied.

Fredman and Tarjan [4] show that it takes $O(n^2 \log n + nm)$ to find a min-cost maximum bipartite matching for a bipartite graph G, where n is the number of nodes in G and m is the number of edges in G. Let C be the number of instructions in a long instruction word (that is also the number of columns in the given initial schedule). In BipartiteMatching_NodesSchedule() (Figure 5) the number of nodes in a row is at most C and the number of blocks is at most $C*|V|$. To construct each edge in the bipartite graph, we need $O(|E|)$ time to go through the graph to check dependencies and decide whether we can put a node into an empty location. The constructed bipartite graph has at most $(C + C * |V|)$ nodes and at most $C^2 * |V|$ edges. So it takes $O(|E| + |V|^2 * \log |V|)$ to finish the rotation in BipartiteMatching_NodesSchedule(). In RowByRow_BipartiteMatching() (Figure 8), it takes $O((2C)^2 \log 2C + 2C * (2C)^2)$ to reschedule one row. So it takes $O(|V|^2)$ to finish the rescheduling row by row in RowByRow_BipartiteMatching() considering C is a constant. Therefore, the complexity of SAMLS is $O(N * (|E| + |V|^2 \log |V|))$, where N is the rotation times, $|E|$ is the number of edges, and $|V|$ is the number of nodes.

4: Experiments

In this section, we experiment with the SAMLS algorithm on a set of benchmarks including 4-stage lattice filter, 8-stage lattice filter, differential equation solver, elliptic filter and voltera filter. In the experiments, we select the rotation times, N, as $10 * Sch\_Len$ where Sch_Len is the schedule length of the given initial schedule. That means each node is rotated about 10 times on average. The experimental results show that the rotation times to generate the best schedules are around $1 * Sch\_Len$, which is the time when all nodes have been rotated one time.

In the experiments, we first obtain the linear assembly code based on TI C6000 for various benchmarks. Then we model them as the cyclic DFGs. We compare the schedules for each benchmark by various techniques: the list scheduling, the algorithm in [7], rotation scheduling, the PRRS algorithm in [11] and our SAMLS algorithm. In the list scheduling, the priority of a node is set as the longest path from this node to a leaf node [9]. The running time of SAMLS on each benchmark is less than one minute.

Bench.	**List**		**Rotation**		**PRRS [11]**		**SAMLS**			
	SA	**SL**	**SA**	**SL**	**SA**	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Lattice	68	9	72	7	38	7	**34**	**50.0%**	**7**	**22.2%**
8-Lattice	108	17	118	11	68	11	**56**	**48.1%**	**11**	**35.3%**
DEQ	30	5	32	4	14	4	**12**	**60.0%**	**4**	**20.0%**
Elliptic	136	14	136	14	86	14	**72**	**47.1%**	**14**	**0.0%**
Voltera	70	12	68	12	38	12	**32**	**54.3%**	**12**	**0.0%**
Average Reduction (%) over List Scheduling								**51.9%**	–	**15.5%**

Table 1. The bus switching activities and schedule length for list scheduling, rotation scheduling, PRRS and SAMLS when FUs=4.

The experimental results for the list scheduling, the rotation scheduling, the PRRS algorithm from [11] and our SAMLS algorithm, are shown in Table 1 when the number of FUs is 4. Column "SA" presents the switching activity of the static schedule and Column "SL" presents the schedule length obtained from three different scheduling algorithms. Column "SL(%)" and "SA(%)" under "SAMLS" present the percentage of reduction in schedule length and switching activities respectively compared to the list scheduling algorithm. The average reduction is shown in the last row of the table. SAMLS shows an average 15.5% reduction in schedule length and 51.9% reduction in bus switching activities compared with the list scheduling.

Bench.	**HV_Schedule ([7])**		**SAMLS**			
	SA	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Lattice	46	9	**34**	**26.1%**	**7**	**0.0%**
8-Lattice	64	17	**56**	**12.5%**	**11**	**17.6%**
DEQ	26	5	**12**	**53.8%**	**4**	**20.0%**
Elliptic	74	14	**72**	**2.7%**	**14**	**0.0%**
Voltera	42	12	**32**	**23.8%**	**12**	**0.0%**
Average Reduction (%)				**23.8%**	–	**15.5%**

Table 2. The bus switching activities and schedule length for SAMLS and the algorithms in[7] when FUs=4.

To compare the performance between SAMLS and the algorithms in [7], we implement their horizontal scheduling and vertical scheduling and do experiments with window size 8. The experimental results for the various benchmarks are shown in Table 2 when the number of FUs is 4. In the table, "HV_Schedule" presents the algorithms in [7]. SAMLS shows an average 15.5% reduction in schedule length and 23.8% reduction in bus switching activity compared with the algorithms in [7].

5: Conclusion

This paper studied the scheduling problem that minimizes both schedule length and switching activities for applications with loops on VLIW architectures. An algorithm, SAMLS (Switching-Activity Minimization Loop Scheduling), was proposed. The algorithm attempted to minimize both switching activities and schedule length by rescheduling nodes repeatedly based on rotation scheduling and bipartite matching. The experimental results showed that our algorithm produces a schedule with a great reduction in switching activities and schedule length for high performance DSP applications.

References

[1] J.M. Chang and M. Pedram. Register allocation and binding for low power. In *Proc. of the 32nd ACM/IEEE Design Automation Conference*, pages 29–35, June 1995.

[2] L.-F. Chao. *Scheduling and Behavioral Transformations for Parallel Systems*. PhD thesis, Dept. of Computer Science, Princeton University, 1993.

[3] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Trans. on Computer-Aided Design*, 16(3):229–239, March 1997.

[4] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[5] M. J. Irwin. Tutorial: Power reduction techniques in SoC bus interconnects. In *1999 IEEE International ASIC/SOC Conference*, 1999.

[6] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, June 1988.

[7] C. Lee, J.-K. Lee, T. Hwang, and Shi-Chun Tsai. Compiler optimization on VLIW instruction scheduling for low power. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):252–268, Apr. 2003.

[8] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[9] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[10] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Instruction scheduling based on energy and performance constraints. In *IEEE Computer Society Annual Workshop on VLSI*, pages 37–42, Apr. 2000.

[11] Z. Shao, Q. Zhuge, E. H.-M. Sha, and C. Chantrapornchai. Loop scheduling for minimizing schedule length and switching activities. In *the 2003 IEEE International Symposium on Circuits and Systems*, volume V, pages 109–112, May 2003.

[12] Dongkun Shin and Jihong Kim. An operation rearrangement technique for low-power vliw instruction fetch. In *Workshop on Complexity-Effective Design*, June 2000.

[13] Dongkun Shin, Jihong Kim, and Naehyuck Chang. An operation rearrangement technique for power optimization in vliw instruction fetch. In *Design Automation and Test in Europe Conference*, page 809, March 2001.

[14] Mircea R. Stan and Wayne P. Burleson. Bus-invert coding for low-power i/o. *IEEE Trans. on VLSI Syst.*, 3(1):49–58, March 1995.

[15] V. Sundararajan and K. K. Parhi. Reducing bus transition activity by limited weight coding with codeword slimming. In *2000 Great Lakes Symposium on VLSI*, pages 13–16, March 2000.

[16] Texas Instruments, Inc. *TMS320C6000 Optimizing Compiler User's Guide*, 2001.

[17] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *the Symposium on Low Power Electronics*, pages 38–39, 1994.

[18] Hongbo Yang, , Guang R. Gao, and Clement Leung. On achieving balanced power consumption in software pipelined loops. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 210–217, 2002.

[19] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *the 37th Design automation Conference*, pages 340–345, June 2000.

[20] Han-Saem Yun and Jihong Kim. Power-aware modulo scheduling for high-performance vliw processors. In *the 2001 international symposium on Low power electronics and design*, pages 40–45, August 2001.

[21] Q. Zhuge, B. Xiao, and E. H.-M. Sha. Code size reduction technique and implementation for software-pipelined dsp applications. *ACM Transactions on Embedded Computing Systems*, 2(4):1–24, Nov. 2003.