Loop Scheduling with Timing and Switching-Activity Minimization for VLIW DSP *

Zili Shao, Chun Xue, Qingfeng Zhuge, Bin Xiao, Edwin H.-M. Sha †

Abstract

In embedded systems, high performance DSP needs to be performed not only with high data throughput but also with low power consumption. 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 reduce both schedule length and bus switching activities. Compared with the work in [10], SAMLS shows an average 11.5% reduction in schedule length and an average 19.4% reduction in bus switching activities.

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

†Z. Shao, C. Xue, Q. Zhuge and E. H.-M. are with the Department of Computer Science, the University of Texas at Dallas, Richardson, TX 75083, USA. Email: {zxs015000, cxx016000, cxx016000, edsha}@utdallas.edu. Bin Xiao is with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. Email: csbxiao@comp.polyu.edu.hk.

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.

In CMOS circuits, there are three major sources of power dissipation: switching, direct-path short circuit current and leakage current [2]. Among them, the dynamic power caused by switching is the dominant part and can be represented as [21]:

$$P_{chip} \propto \sum_{i=1}^{N} C_{load_i} \times V_{dd}^2 \times f \times p_{t_i} \quad (1)$$

where: C_{load_i} is the load capacitance at node i, V_{dd} is the power supply voltage, f is the frequency, p_{t_i} is the activity factor at node i, and the power consumption of a circuit is the summation of power consumption over all N nodes of this circuit. From this equation, reducing switching activities (lowering the activity factor p_{t_i}) can effectively decrease the power consumption. Therefore, in VLSI system designs, various techniques have been proposed to reduce power consumption by reducing switching activities [2, 16, 21]. Due to large capacitance and high transition activities, buses including instruction bus, data bus, address bus, etc. consume a significant fraction of total power dissipation in a processor [13]. 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 [8]. In this paper, we focus on reducing the power consumption of applications on VLIW architectures by reducing transition activities on the instruction bus. 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.

We study this 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 power bus coding techniques [14,21,23] 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 carefully arranging the instructions of an application.

In recent years, people have addressed the issue to reduce power consumption by software arrangement at instruction level [11, 22, 26]. 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 at the address lines [22], register assignment problem [3], operand swapping and dual memory [11], data bus between cache and main memory [27], and I-cache data bus [29].

For VLIW architectures, low-power related instruction scheduling techniques have been proposed in [9,10,17,31]. In most of these work, the scheduling techniques are based on traditional list scheduling in which applications are modeled as DAG 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 [30] propose a power-aware modulo scheduling algorithm to reduce both the step power and peak power for VLIW processors. Yang et al. [28] 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.

Lee et al. [10] propose an instruction scheduling technique to produce a schedule with bus switching activities minimization on VLIW architectures for applications represented as DAGs. In their work, the problem is categorized into horizontal scheduling problem and vertical scheduling problem. A greedy bipartite-matching scheme is proposed to optimally solve horizontal scheduling problem. And vertical scheduling problem is proved to be NP-hard problem and a heuristic algorithm is proposed to solve it.

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 [10] that optimizes the DAG part of a loop, our technique shows an average 19.4% reduction in swithing activities and an average 11.5% 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 [5, 18]. By exploiting inter-iteration dependencies, we provide more opportunities to reschedule nodes to the best locations so the switching activities can be minimized. However, the traditional software pipelining such as modulo scheduling [18], rotation scheduling [5], 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.

We propose an algorithm, SAMLS (Switching-Activity Minimization Loop Scheduling), to minimize both schedule length and switching activities for loops. Our loop scheduling scheme reduces the energy of a program by reducing both schedule length and bus switching activities. The energy E consumed by a program can be calculated by $E = P \times T$, where T is the execution time of the program and P is the average power [11, 26]. The execution time of a program is reduced by reducing schedule length. As shown in Equation 1, the real capacitance loading is reduced by reducing switching activities. So the average power consumption is reduced by minimizing switching activities in the instruction bus.

In SAMLS, 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 based on rotation scheduling and bipartite matching. Our algorithm exploits inter-iteration dependencies of a loop and changes intra-iteration dependencies of nodes using rotation scheduling. When a node is rotated, it can be rescheduled into more locations compared with the case only considering the DAG part of a loop. Therefore, more opportunities are provided to optimize switching activities. 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 [10]. In the experiments, the real TI C6000 instructions [24] are used. The experimental results show that our algorithm can reduce both bus switching activities and schedule length. Compared with the list scheduling, SAMLS shows an average 11.5% reduction in schedule length and 45.7% reduction in bus switching activities. Compared with the technique in [10]

that combines horizontal scheduling and vertical scheduling with window size eight, SAMLS shows an average 11.5% reduction in schedule length and an average 19.4% reduction in bus switching activities.

The remainder of this paper is organized as follows. In Section 2, we give the basic models and concepts used in the rest of the paper. The 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 lower bounds of schedule length for cyclic DFGs and the basic concepts of rotation scheduling.

2.1 The Target VLIW Architecture and Cost Model

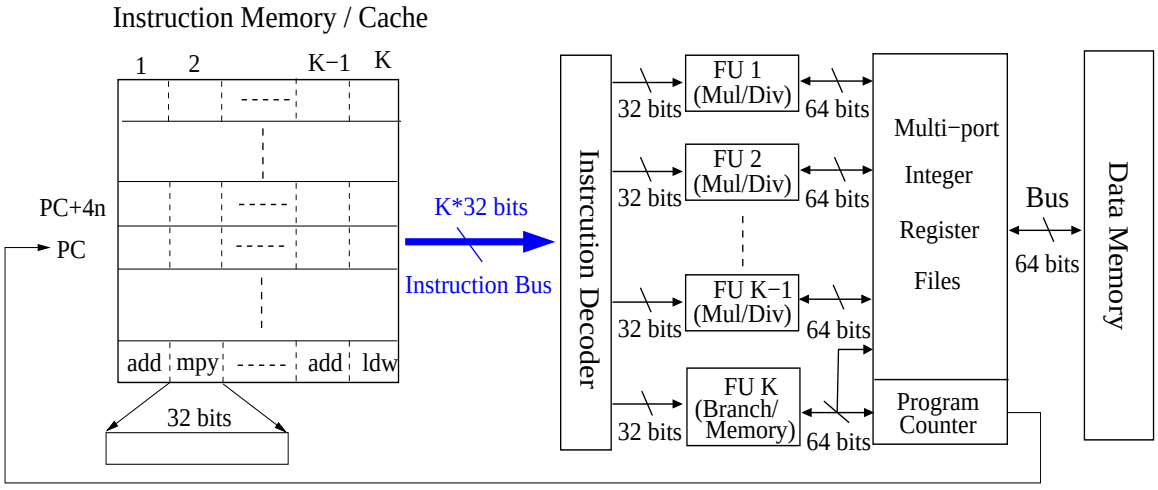


Figure 1: The target VLIW architecture and bus models.

The abstract VLIW machine is shown in Figure 1 that has the similar architecture as that in [10]. 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 operations 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 do experiments on a set of benchmarks with the real TI C6000 instructions and obtain the results when K equals 3, 4 and 5, respectively.

We use the same cost model as used in [10]. *Hamming Distance* is used 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 on the instruction bus is:

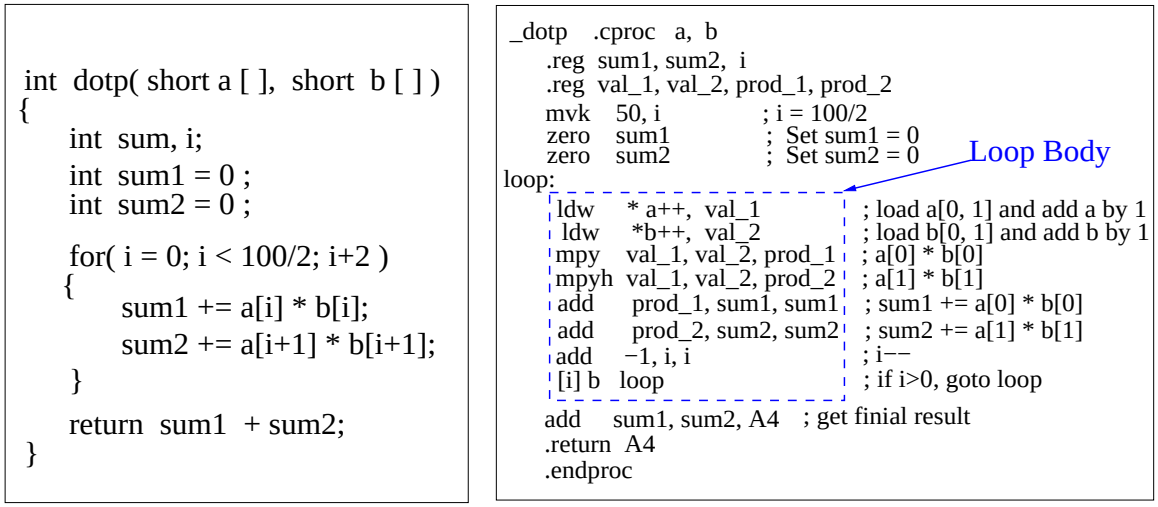
$$H(X,Y) = \sum_{i=1}^{K} h(Binary\_String(x_i), Binary\_String(y_i)), \tag{2}$$

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 $Binary\_String(y_i)$.

2.2 Loops and Cyclic Data-Flow-Graph (DFG)

We use cyclic DFG to model loops. A *cyclic Data Flow Graph (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 involved.

We use a real loop application, a dot-product program, to show how to use cyclic DFG to model



(a) C Code for Dot Product.

(b) Assemly Code for Dot Product.

Figure 2: A dot-product C code and its corresponding assembly code from TI C6000 [25].

a loop. A program to compute the dot-product of two integer arrays is shown in Figure 2(a) and its corresponding assembly code from TI C6000 [25] is shown in Figure 2(b). Our focus is the loop body. Basically, in the loop body in Figure 2(b), 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 2, the mapping between the node and instruction is shown in Figure 3(a) and the corresponding cyclic DFG is shown in Figure 3(b).

2.3 A Static Schedule and Its Switching Activities

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 3(b) to the target VLIW architecture with 3 FUs ($K = 3$) (discussed in Section 2.1). And let functional unit FU_1 and FU_2 be integer ALUs, and FU_3 be the load/store/branch Unit. The static schedule generated by the list scheduling is shown in Figure 3(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

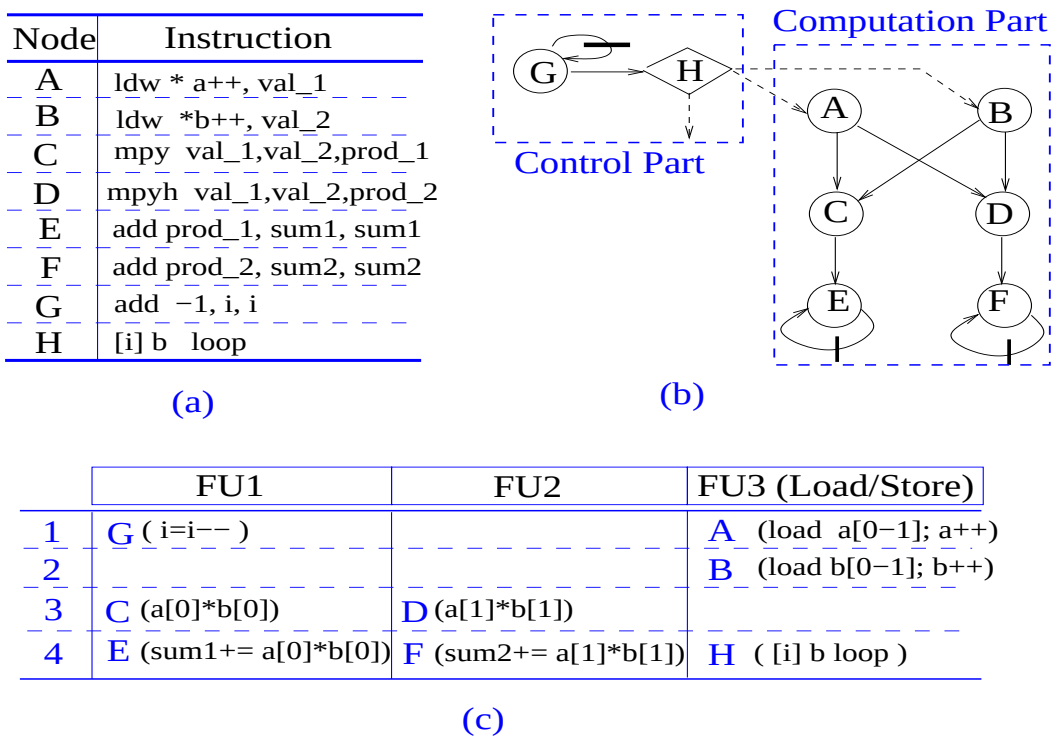


Figure 3: (a) The nodes and their corresponding instructions. (b) The cyclic DFG that represents the loop body in Figure 2(b). (c) The schedule generated by the list scheduling.

Figure 3(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 in the instruction bus. Since the static schedule is repeatedly executed for a loop, when switching activities are calculated, 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 of a schedule can be calculated from the second iteration by summing up all switching activities caused by each long instruction word in the instruction bus. The bus switching activities caused for each iteration except the first one are equal to the switching activities obtained from the second iteration. For the first iteration, a different initial state may exist in the instruction bus when the first instruction word is scheduled. However, since a loop is usually executed for many times, 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.

2.4 Lower Bounds of Schedule Length for Cyclic DFGs

The lower bound of schedule length for a cyclic DFG denotes the smallest possible value of the schedule length for which a schedule exists. The lower bound for a DFG under resource constraints can be derived from either the structure of the DFG or the resource availability. The lower bound from the structure of the DFG is called as *iteration bound* [19]. The iteration bound of DFG G, denoted by $IB(G)$, is defined to be the maximum-time-to-delay ratio over all cycles of the DFG, i.e.

$$IB(G) = \max_{\forall \text{ cycle } l \text{ in } G} \frac{Time(l)}{Delay(l)},$$

where $Time(l)$ is the sum of computation time in cycle l, and $Delay(l)$ is the sum of delay counts in cycle l. The iteration bound of a cyclic DFG can be obtained in polynomial time by the longest path matrix algorithm [7]. We implement the longest path matrix algorithm and calculate the iteration bound of each benchmark in the experiments.

The lower bound from resource availability for DFG G, denoted by $RB(G)$, is defined as the maximum ratio of number of operations to number of FUs over all FU types, i.e.,

$$RB(G) = \max_{\forall \text{ FU type } A} \frac{N(A)}{F(A)},$$

where $N(A)$ is the number of operations using type-A FUs in the DFG, and $F(A)$ is the number of type-A FUs available. After IB and RB are obtained, then the lower bound of DFG G, denoted by $LB(G)$, can be obtained by taking the maximum value of IB and RB, i.e.

$$LB(G) = \max\{IB(G), RB(G)\}.$$

2.5 Retiming and Rotation Scheduling

Considering inter-iteration dependencies, retiming and rotation are two optimization techniques for the scheduling of cyclic DFGs. *Retiming* [12] 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 [5] 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 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.

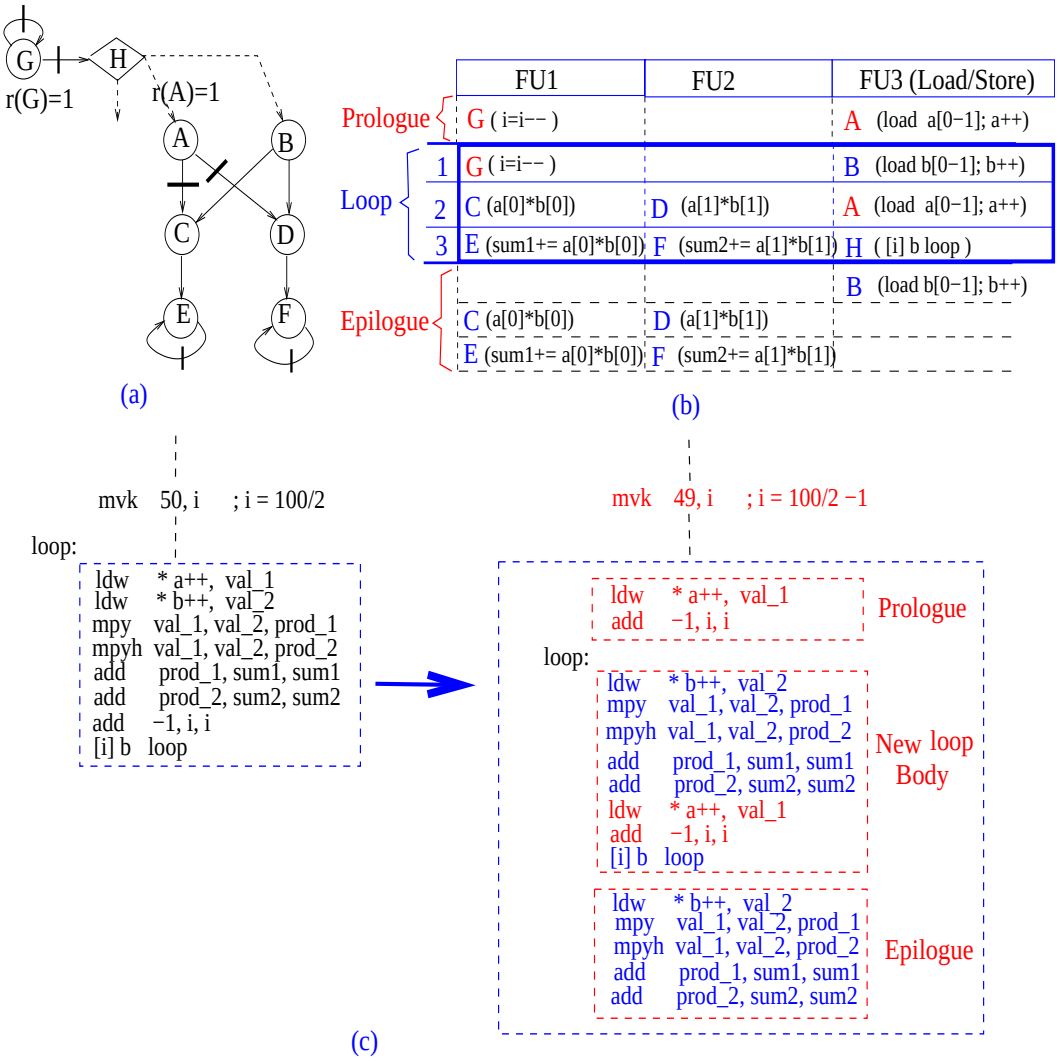


Figure 4: (a) The retimed graph. (b) The schedule after the first Rotation. (c) The corresponding transformation for the loop body.

Using the schedule generated by the list scheduling in Figure 3(b) as an initial schedule, we give an example in Figure 4 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 4(a) and the schedule after the first rotation is shown in Figure 4(b). 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 4(c), 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 [4]. 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 [32].

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 5(a), and the schedules are shown in Figures 5(b)-(e) 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 5(e) has the minimal bus switching activities and the minimal schedule length.

3 Switching-Activity Minimization Loop Scheduling

The loop scheduling problem with minimum latency and minimum switching activities is NP-complete with or without resource constraints [20]. In this section, we propose an algorithm, SAMLS (Switching-Activity Minimization Loop Scheduling), to reduce both schedule length and switching activities for applications with loops. We first present the SAMLS algorithm in Section 3.1 and then discuss its two key functions in Section 3.2 and 3.3. Finally, we analyze properties and complexity of the SAMLS algorithm in Section 3.4.

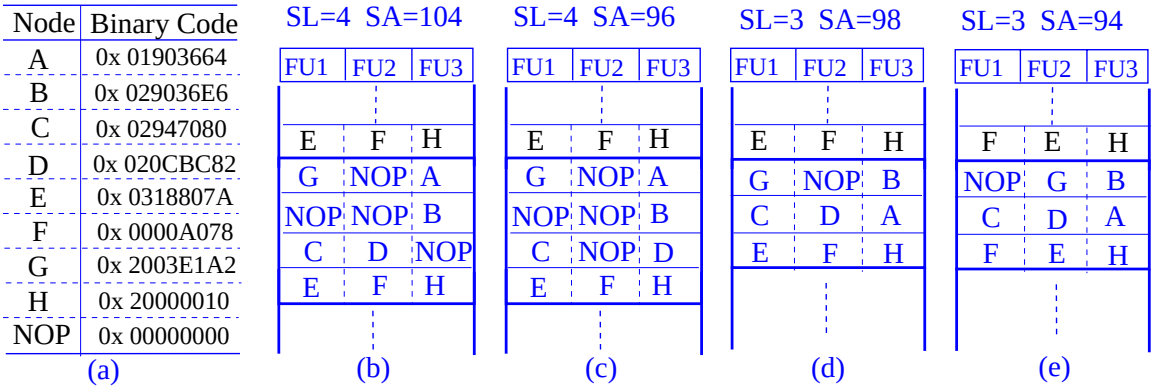


Figure 5: (a)The nodes and TI C6000 machine code. (b) The schedule generated by the list scheduling. (c) The schedule generated by the algorithms in [10]. (d) The schedule generated by rotation scheduling. (e) The schedule generated by our technique.

3.1 The SAMLS Algorithm

The SAMLS algorithm is designed to reduce both schedule length and switching activities for a cyclic DFG based on a given initial schedule. 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. SAMLS is shown as follows:

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) Call function BipartiteMatching_NodesSchedule(G,r,S,R) to reschedule the nodes in R.

   (c) Call function RowByRow_BipartiteMatching(S) to Minimize the switching activities of S row by row.

   (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 $r' \leftarrow r_j$.

In Algorithm 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. Then RowByRow_BipartiteMatching() is 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 Section 3.2 and Section 3.3 below.

3.2 BipartiteMatching_NodesSchedule()

In rotation scheduling [5], in order to minimize schedule length, the nodes in the first row of the schedule are rotated down and put into the earliest locations based on the dependency relations in G_r (the retimed graph obtained from G with retiming function r). In our case, we also need to consider switching activities minimization. We solve this problem by constructing a weighted bipartite graph between the nodes and the empty locations and rescheduling the nodes based on the obtained minimum cost matching. BipartiteMatching_NodesSchedule() is shown as follows:

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

Output: The revised schedule S.

Algorithm:

1. $\mathtt{Len} \leftarrow$ 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 $\mathtt{Len} \leftarrow \mathtt{Len} + 1$; **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.

}

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 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

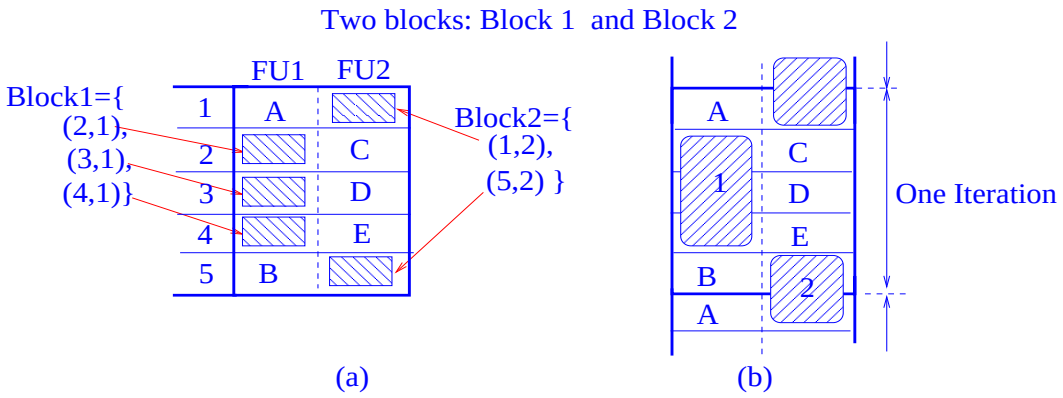


Figure 6: (a) A given schedule. (b) Two blocks that contain consecutive empty locations in a column.

same column, then $Switch\_Block(u, b_i)$ is computed by:

$$Switch\_Block(u, b_i) = H(u, u') + H(u, u'') - H(NOP, u') - H(NOP, u'') \quad (3)$$

$Switch\_Block(u, b_i)$ 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.

Using the schedule generated by the list scheduling in Figure 3(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 3 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).

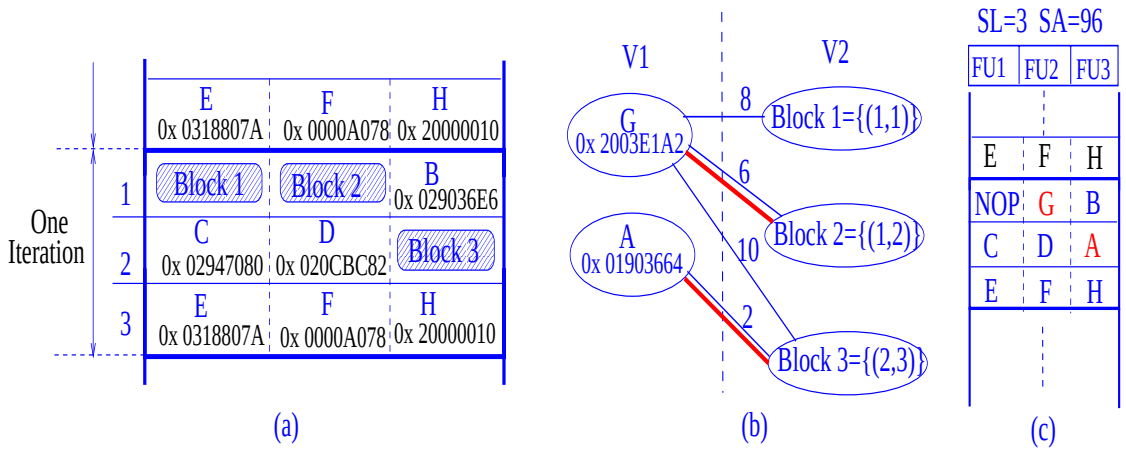


Figure 7: (a) The schedule with the first row removed. (b) The weighted bipartite graph. (c) The obtained schedule.

3.3 RowByRow_BipartiteMatching()

After rescheduling the nodes by function BipartiteMatching_NodesSchedule(), we horizontally schedule nodes in each row to further reduce switching activities by function RowByRow_BipartiteMatching(). The algorithm is similar to the horizontal scheduling in [10]. 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 as follows:

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:

$$W(e(u,k)) = \begin{cases} \text{h(Binary\_String(u),BS[k])} & \text{j<Len,} \\ \text{h(Binary\_String(u),BS[k])+h(Binary\_string(u),Init\_BS[k])} & \text{Otherwise} \end{cases}$$

- $M \leftarrow$ 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.

In RowByRow_BipartiteMatching(), we generate Len schedules by repeatedly rotating down the first row to the last, where Len is the schedule length. For each schedule, we fix the first row to record the binary string of the node at (1,k) into $BS[k]$ and $Init\_BS[k]$ for each $k = 1, 2, \cdots, Col$. Then we construct a weighted bipartite graph between the nodes and the locations in the current row, and reschedule the nodes row by row based on the obtained minimum cost matching. When constructing the weighted bipartite graph for row j, we has two cases:

1. When row j is not the last row, we set the weight of edge $e(u,k)$ (node u matches to location (j,k)) as the hamming distance between the binary string of u and $BS[k]$, where $BS[k]$ records the binary string of the node located immediately above (j,k);

2. When row j is the last row, we set the weight of edge $e(u, k)$ as the summation of two hamming distances: one is from u and the node immediately above (j,k) that is the binary string recorded in $BS[k]$, and the other is from u and the node immediately below (j,k) that is the binary string recorded in $Init\_BS[k]$.

In such a way, we consider the influence from both the second to the last row and the first row of the next iteration when rescheduling nodes in the last row. The schedule with minimal switching activities is selected from these Len schedules.

An example is given in Figure 8 to show that we need to consider three cases in order to horizontally minimize switching activities of the schedule given in Figure 8(a). As shown in Figures 8(b)-(d), in each case, one row is fixed and set as the initial row, and the other rows are rescheduled based it; when processing the last row, the influence from the previous row and the first row in the next iteration are considered together. After running RowByRow_BipartiteMatching(), we obtain the finial schedule shown in Figure 5(e) in Section 2.5.

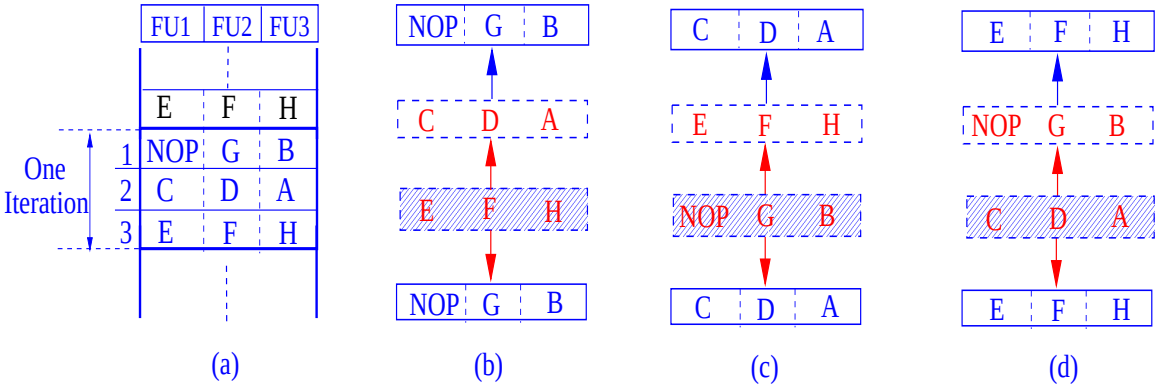


Figure 8: (a) The schedule obtained from BipartiteMatching_NodesSchedule() (Figure 7(c)). (b) Fix the first row. (c) Fix the second row. (d) Fix the third row.

3.4 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 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 [4]. 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 [6] 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(), 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(), 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 (4-Stage), 8-stage Lattice Filter (8-Stage), UF2-8-stage Lattice Filter (uf2-8Stage), Differential Equation Solver (DEQ), UF2-Differential Equation Solver (uf2-DEQ), Fifth Order Elliptic Filter (Elliptic), Voltera Filter (Voltera), UF2-Voltera Filter (uf2-Voltera), 2-cascaded Biquad Filter (Biquad) and RLS-laguerre Lattice Filter (RLS-Laguerre). In the benchmarks, UF2-8-stage Lattice Filter, UF2-Differential Equation Filter and UF2-Voltera Filter are obtained by unfolding 8-stage Lattice Filter, Differential Equation Solver (DEQ) and Voltera Filter (Voltera), respectively, with unfolding factor 2. The numbers of nodes and edges for each benchmark are shown in Table 1. In the experiments, we select 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.

	4-Stage	8-Stage	uf2-8Stage	DEQ	uf2-DEQ
The Number of Nodes	26	42	84	11	22
The Number of Edges	35	59	118	20	40
	Elliptic	Voltera	uf2-Voltera	Biquad	RLS-Laguerre
The Number of Nodes	34	27	54	16	19
The Number of Edges	58	34	68	23	43

Table 1: The numbers of nodes and edges for each benchmark.

In our experiments, the instructions are obtained from TI TMS320C 6000 Instruction Set. The VLIW architecture in Section 2.1 is used as the test platform. 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 [10], rotation scheduling 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 [15]. In the experiments, we use LP_SOLVE [1] to obtain a min-cost maximum bipartite matching based on ILP form (integer linear program) of weighted bipartite graph. The experiments are performed on a Dell PC with a P4 2.1 G processor and 512 MB memory running Red Hat Linux 9.0. Every experiment is finished within one minute.

The experimental results for the list scheduling, rotation scheduling, and our SAMLS algorithm, are shown in Table 2-4 when the number of FUs is 3, 4 and 5, respectively. Column “LB” presents the lower bound of schedule length obtained using the approach in Section 2.4. Column “SA” presents the switching activity of the static schedule and Column “SL” presents the schedule length obtained from three different scheduling algorithms: the list scheduling (Field “List”), the traditional rotation scheduling (Field “Rotation”), and our SAMLS algorithm (Field “SAMLS”). 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. Totally, SAMLS shows an average 11.5% reduction in schedule length and 45.7% reduction in bus switching activities compared with the list scheduling. SAMLS achieves the lower bounds of

The number of FUs = 3									
Bench.	**LB**	**List**		**Rotation**		**SAMLS**			
		SA	**SL**	**SA**	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Stage	**9**	66	9	66	9	**40**	**39.4%**	**9**	**0.0%**
8-Stage	**14**	106	17	118	14	**84**	**20.8%**	**14**	**17.6%**
uf2-8Stage	**28**	242	28	242	28	**172**	**28.9%**	**28**	**0.0%**
DEQ	**4**	30	5	34	4	**24**	**20.0%**	**4**	**20.0%**
uf2-DEQ	**8**	66	9	66	8	**42**	**36.4%**	**8**	**11.1%**
Elliptic	**13**	140	15	140	15	**102**	**27.1%**	**15**	**0.0%**
Voltera	**12**	70	12	66	12	**40**	**42.9 %**	**12**	**0.0%**
uf2-Voltera	**24**	106	24	110	24	**68**	**35.8 %**	**24**	**0.0%**
Biquad	**6**	54	7	44	6	**26**	**51.9 %**	**6**	**14.3%**
RLS-Laguerre	**7**	54	7	60	7	**36**	**33.3 %**	**7**	**0.0%**
Average Reduction (%) over List Scheduling							**33.6 %**	–	**6.3%**

Table 2: The comparison of bus switching activities and schedule length for list scheduling, rotation scheduling and SAMLS.

The number of FUs = 4									
Bench.	**LB**	**List**		**Rotation**		**SAMLS**			
		SA	**SL**	**SA**	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Stage	**7**	68	9	72	7	**34**	**50.0%**	**7**	**22.2%**
8-Stage	**11**	108	17	118	11	**56**	**48.1%**	**11**	**35.3%**
uf2-8Stage	**21**	230	21	230	21	**186**	**19.1%**	**21**	**0.0%**
DEQ	**4**	30	5	32	4	**12**	**60.0%**	**4**	**20.0%**
uf2-DEQ	**8**	72	9	72	8	**30**	**58.3%**	**8**	**11.1%**
Elliptic	**13**	136	14	136	13	**66**	**51.5%**	**13**	**7.1%**
Voltera	**12**	70	12	68	12	**32**	**54.3%**	**12**	**0.0%**
uf2-Voltera	**24**	106	24	104	24	**56**	**47.2 %**	**24**	**0.0%**
Biquad	**4**	52	6	50	4	**32**	**38.5 %**	**4**	**33.3%**
RLS-Laguerre	**7**	64	7	64	7	**42**	**34.4 %**	**7**	**0.0%**
Average Reduction (%) over List							**46.1 %**	–	**12.9%**

Table 3: The comparison of bus switching activities and schedule length for list scheduling, rotation scheduling and SAMLS.

The number of FUs = 5									
Bench.	**LB**	**List**		**Rotation**		**SAMLS**			
		SA	**SL**	**SA**	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Stage	**6**	74	9	80	6	**22**	**70.3%**	**6**	**33.3%**
8-Stage	**9**	106	17	112	9	**50**	**52.8%**	**9**	**47.1%**
uf2-8Stage	**17**	218	17	218	17	**152**	**30.3%**	**17**	**0.0%**
DEQ	**4**	30	5	36	4	**8**	**73.3%**	**4**	**20.0%**
uf2-DEQ	**8**	60	9	64	8	**18**	**70.0%**	**8**	**11.1%**
Elliptic	**13**	136	14	136	13	**50**	**63.2%**	**13**	**7.1%**
Voltera	**12**	72	12	72	12	**24**	**66.7%**	**12**	**0.0%**
uf2-Voltera	**24**	112	24	112	24	**56**	**50.0 %**	**24**	**0.0%**
Biquad	**4**	54	6	50	4	**18**	**66.7 %**	**4**	**33.3%**
RLS-Laguerre	**7**	46	7	56	7	**32**	**30.4 %**	**7**	**0.0%**
Average Reduction (%) over List Scheduling							**57.4 %**	–	**15.2%**

Table 4: The comparison of bus switching activities and schedule length for list scheduling, rotation scheduling and SAMLS.

schedule length in all experiments except one for Elliptic Filter when the number of FUs equals 3, in which the schedule length obtained by SAMLS (15) is very close to the lower bound (13).

To compare the performance between SAMLS and the algorithms in [10], 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 5-7 when the number of FUs is 3, 4 and 5, respectively. In the table, "HV_Schedule" presents the algorithms in [10]. Totally, SAMLS shows an average 11.5% reduction in schedule length and 19.4% reduction in bus switching activity compared with the algorithms in [10].

Through the experimental results from Table 2 and Table7, we found that the traditional rotation scheduling can effectively reduce schedule length but not bus switching activities. The algorithms in [10] can reduce bus switching activities without timing performance optimization for applications with loops. Our SAMLS can significantly reduce both schedule length and switching activities.

The number of FUs = 3							
Bench.	**LB**	**HV_Schedule ([10])**		**SAMLS**			
		SA	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Stage	**9**	56	9	**40**	**28.6%**	**9**	**0.0%**
8-Stage	**14**	90	17	**84**	**6.7%**	**14**	**17.6%**
uf2-8Stage	**28**	174	28	**172**	**1.1%**	**28**	**0.0%**
DEQ	**4**	24	5	**24**	**0.0%**	**4**	**20.0%**
uf2-DEQ	**8**	50	9	**42**	**16.0%**	**8**	**11.1%**
Elliptic	**13**	108	15	**102**	**5.6%**	**15**	**0.0%**
Voltera	**12**	54	12	**40**	**25.9 %**	**12**	**0.0%**
uf2-Voltera	**24**	80	24	**68**	**15.0 %**	**24**	**0.0%**
Biquad	**6**	28	7	**26**	**7.1 %**	**6**	**14.3%**
RLS-Laguerre	**7**	46	7	**36**	**21.7 %**	**7**	**0.0%**
Average Reduction (%)					**12.8 %**	–	**6.3%**

Table 5: The comparison of bus switching activities and schedule length between SAMLS and the algorithms in [10].

The number of FUs = 4							
Bench.	**LB**	**HV_Schedule ([10])**		**SAMLS**			
		SA	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Stage	**7**	46	9	**34**	**26.1%**	**7**	**22.2%**
8-Stage	**11**	64	17	**56**	**12.5%**	**11**	**35.3%**
uf2-8Stage	**21**	190	21	**186**	**2.1%**	**21**	**0.0%**
DEQ	**4**	26	5	**12**	**53.8%**	**4**	**20.0%**
uf2-DEQ	**8**	58	9	**30**	**48.3%**	**8**	**11.1%**
Elliptic	**13**	74	14	**66**	**10.8%**	**13**	**7.1%**
Voltera	**12**	42	12	**32**	**23.8%**	**12**	**0.0%**
uf2-Voltera	**24**	76	24	**56**	**26.3 %**	**24**	**0.0%**
Biquad	**4**	36	6	**32**	**11.1 %**	**4**	**33.3%**
RLS-Laguerre	**7**	44	7	**42**	**4.5 %**	**7**	**0.0%**
Average Reduction (%)					**21.9 %**	–	**12.9%**

Table 6: The comparison of bus switching activities and schedule length between SAMLS and the algorithms in [10].

The number of FUs = 5							
Bench.	**LB**	**HV_Schedule ([10])**		**SAMLS**			
		SA	**SL**	**SA**	**SA(%)**	**SL**	**SL(%)**
4-Stage	**6**	36	9	**22**	**38.9%**	**6**	**33.3%**
8-Stage	**9**	52	17	**50**	**3.8%**	**9**	**47.1%**
uf2-8Stage	**17**	174	17	**152**	**12.6%**	**17**	**0.0%**
DEQ	**4**	12	5	**8**	**33.3%**	**4**	**20.0%**
uf2-DEQ	**8**	30	9	**18**	**40.0%**	**8**	**11.1%**
Elliptic	**13**	70	14	**50**	**28.6%**	**13**	**7.1%**
Voltera	**12**	34	12	**24**	**29.4%**	**12**	**0.0%**
uf2-Voltera	**24**	58	24	**56**	**3.4 %**	**24**	**0.0%**
Biquad	**4**	32	6	**18**	**43.8 %**	**4**	**33.3%**
RLS-Laguerre	**7**	32	7	**32**	**0.0 %**	**7**	**0.0%**
Average Reduction (%)					**23.4 %**	–	**15.2%**

Table 7: The comparison of bus switching activities and schedule length between SAMLS and the algorithms in [10].

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] M. Berkelaar. *Unix Manual of lp_solve*. Eindhoven University, 1992.

[2] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[3] J. 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.

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

[5] 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.

[6] 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.

[7] S. H. Gerez, S. H. de Groot, and O. Herrmann. A polynomial-time algorithm for the computation of the iteration-period bound in recursive data-flow graphs. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39(1):49–52, Jan. 1992.

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

[9] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Adapting instruction level parallelism for optimizing leakage in vliw architectures. In *LCTES 2003*, pages 275–283, June 2003.

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

[11] M. T.-C. Lee, M. Fujita, V. Tiwari, and S. Malik. Power analysis and minimization techniques for embedded dsp software. *IEEE Transactions on VLSI Systems*, 5(1):123–135, March 1997.

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

[13] D. Liu and C. Svensson. Power consumption estimation in cmos vlsi chips. *IEEE Journal of Solid State Circuits*, 29(6):663—670, 1994.

[14] M. Mamidipaka, D. Hirschberg, and N. Dutt. Adaptive low power encoding techniques using self organizing lists. *IEEE Trans. on VLSI Syst.*, 11(5):827–834, Oct. 2003.

[15] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[16] P. Panda and N. Dutt. Low power memory mapping through reducing address bus activity. *IEEE Trans. on VLSI Syst.*, 7(3):309–320, Sept. 1999.

[17] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Instruction scheduling for low power. *Journal of VLSI Signal Processing*, 37:129–149, 2004.

[18] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *the 25th annual international symposium on Microarchitecture*, pages 158–169, Dec. 1992.

[19] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Trans. Circuits Syst.*, CAS-28(3):196–202, March 1981.

[20] Z. Shao, Q. Zhuge, M. Liu, E. H.-M. Sha, and B. Xiao. Loop scheduling for real-time dsps with minimum switching activities on multiple-functional-unit architectures. In *The 2004 International Conference on Embedded And Ubiquitous Computing, LNCS*, pages 53–63, August 2004.

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

[22] C.-L. Su, C.-Y. Tsui, and A. M. Despain. Saving power in the control path of embedded processors. *IEEE Design & Test of Computers*, 11(4):24–30, Winter 1994.

[23] 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.

[24] Texas Instruments, Inc. *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000.

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

[26] V. Tiwari, S. Malik, and A. Wolfe. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2):1–18, Aug. 1996.

[27] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction scheduling to reduce switching activity of off-chip buses for low-power systems with caches. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E81-A(12):2621–2629, December 1998.

[28] H. Yang, , G. R. Gao, and C. 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.

[29] 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.

[30] H.-S. Yun and J. 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.

[31] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y. Tsai. Exploiting vliw schedule slacks for dynamic and leakage energy reduction. In *the 34th Annual International Symposium on Microarchitecture*, pages 102–113, Dec. 2001.

[32] 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.