Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 410–428
The Journal of Systems and Software
www.elsevier.com/locate/jss

An effective and efficient code generation algorithm for uniform loops on non-orthogonal DSP architecture

Yi-Hsuan Lee, Cheng Chen *

Department of Computer Science and Information Engineering, 1001 Ta Hsueh Road, Hsinchu, 30050, Taiwan, ROC

Received 20 October 2005; received in revised form 30 May 2006; accepted 4 June 2006
Available online 17 July 2006

Abstract

To meet ever-increasing demands for higher performance and lower power consumption, many high-end digital signal processors (DSPs) commonly employ non-orthogonal architecture. This architecture typically is characterized by irregular data paths, heterogeneous registers, and multiple memory banks. Moreover, sufficient compiler support is obviously important to harvest its benefits. However, usual compilation techniques do not adapt well to non-orthogonal architectures and the compiler design becomes much more difficult due to the complexity of these architectures. The entire code generation process for non-orthogonal architecture must include several phases. In this paper, we extend our previous study to propose a code generation algorithm *Rotation Scheduling with Spill Codes Avoiding* (*RSSA*), which is suitable for various DSPs with similar architectural features. As well as introducing detailed principles and algorithms of RSSA, we select several DSP applications and evaluate it under Motorola DSP56000 architectures. The evaluation results clearly demonstrate the effectiveness of RSSA, which can obtain scheduling results with minimum length and fewer spill codes compared to related work. In addition, in order to study the influence of different number of resources on the scheduling result, we also define a hypothetical machine model to represent a scalable non-orthogonal DSP architecture. After evaluating RSSA on various target architectures, we find that adding additional accumulators is the most efficient way to reduce spill codes. Meanwhile, for instruction-level parallelism exploration, numbers of data ALUs and accumulators have to be concurrently increased. Furthermore, based on our analysis, RSSA is not only effective but also quite efficient compared to related studies.

Keywords: DSP; Non-orthogonal architecture; Code generation

1. Introduction

Most scientific and digital signal processing (DSP) applications, such as image processing and weather forecasting, are iterative and usually represented by uniform nested loops (Hsu and Jeang, 1993; Kung, 1988; Madisetti, 1995). A digital signal processor (DSP) is a special-purpose microprocessor that is designed to achieve high performance in DSP applications (Eyer and Bier, 2000). In order to meet stringent speed and power requirements for embedded applications, DSPs commonly employ *non-orthogonal* architectures that are typically characterized by irregular data paths, heterogeneous register sets, and multiple memory banks (Cho et al., 2002). For the data path this architecture has multiple small register files dedicated to different sets of function units instead of a large number of centralized homogeneous registers. In addition, parallel access, enabled by multi-bank memory, is useful to explore the potential of higher memory bandwidth but gives rise to the problem of how to partition variables into the multiple memory banks (Cho et al., 2002; Lee and Chen, 2004; Leupers and Kotte, 2001; Saghir et al., 1994; Saghir et al., 1996; Shiue, 2001; Sudarsanam and Malik, 2000; Wang and Hu, 2004; Zhuge et al., 2001). Therefore, to harvest the benefits provided by this non-orthogonal architecture,

* Corresponding author. Tel.: +886 3 5712121x54734; fax: +886 3 5724176.
E-mail addresses: yslee@csie.nctu.edu.tw (Y.-H. Lee), cchen@csie.nctu.edu.tw (C. Chen).

0164-1212/$ - see front matter © 2006 Published by Elsevier Inc.
doi:10.1016/j.jss.2006.06.002

adequate compiler support is obviously essential (Lapsley et al., 1996; Madisetti, 1995).

Many researchers seek to design code generation algorithms for specific DSP architectures to use their features fully. The complete code generation process for non-orthogonal architecture must include several phases, such as intermediate representation, code compaction, instruction scheduling, memory bank assignment (or variable partition), and accumulator/register assignment (Sudarsanam and Malik, 2000). In our previous study, we proposed two scheduling methods for multi-bank memory architecture that cover all phases except accumulator/register assignment (Lee and Chen, 2004). Next, we also propose a code generation algorithm that contains all of the above phases (Lee and Chen, 2005). From our evaluation, although the algorithm proposed in Lee and Chen (2005) is relatively efficient and effective, it was not scalable and specifically designed for an embedded DSP Motorola DSP56000. Therefore, we want to extend it to a more general algorithm, which is suitable for various DSPs with similar architectural features.

Due to strict resource constraints of the non-orthogonal DSP architecture, accumulator/register spills will occur very often. If more spill codes are added to the final schedule, not only the scheduling length may be lengthened, but also costs more power consumption to execute those additional instructions. That is, in addition to increase the instruction-level parallelism, how to avoid generating too many spill codes is also an important issue of designing the code generation algorithm for non-orthogonal DSP architecture. Moreover, although using an effective code generation algorithm can obtain scheduling results with shorter length and less spill codes, increasing the number of resources is essentially a more direct way to achieve the same goal. Therefore, in this paper, we will propose an effective code generation method, and deep study the influence of differing number of resources on the scheduling result.

In order to do above studies, we need a parameterized architecture to model a scalable non-orthogonal DSP. Many parameterized architecture models have been developed to explore and investigate advanced compiler and architecture research (MESCAL; OptimoDE; ORC; Tensilica; Trimaran). However, none of them can faithfully represent the irregularity of non-orthogonal DSP architecture, especially its two main features multiple memory banks and heterogeneous register sets. Thus, we define a hypothetical machine model extended from the Motorola DSP56000, in which more resources will be included. Our proposed method is named *Rotation Scheduling with Spill Codes Avoiding* (*RSSA*). It is extended from our previous study (Lee and Chen, 2005), and its scheduling goal is to achieve shorter schedule length and avoid generating spill codes as far as possible. RSSA mainly contains five parts with following features. First, it contains a procedure to generate uncompacted codes directly from a high-level language. In most other related methods, this is not included and is done by existing tools (Cho et al., 2002; Sudarsanam and Malik, 2000; Shiue, 2001). Next, memory bank assignment is performed before code compaction as in Lee and Chen (2005). This execution sequence makes memory accesses be scheduled with information of variable partitioning, which can avoid extra cycles to fetch variables. Then, RSSA separately schedules ALU and memory load instructions in different parts. This strategy makes registers unfilled while dealing with accumulator spills, which is beneficial for temporarily storing overwritten ALU results. Compared to store and reload overwritten ALU results in memory, this mechanism requires less spill codes to resolve accumulator spills. It can be shown that using RSSA can obtain scheduling results with minimum length and fewer spill codes compared with related work. The reason is that it first generates the schedule without considering resource constraints and lengthens the schedule only when required.

After introducing the general algorithm, we selected several multi-dimensional data flow graphs (MDFGs) representing DSP applications for evaluation. Two metrics including schedule length and instruction count are used to evaluate the performance at the same time. From the evaluation results, our method actually can obtain shorter schedule lengths and less spill codes than those of related studies under the Motorola DSP56000 architecture. These results represent that our method is really effective on both evaluation metrics. In addition, we further analyze the effectiveness of RSSA itself. When the target architecture consists of more than one data ALU, RSSA can produce a schedule with length equal to or less than the critical path of the given MDFG. If there is only one data ALU, it still can produce the schedule with length equal to the number of ALU instructions of the given MDFG. As for the instruction count, RSSA also generates quite few spill codes. Meanwhile, these additional spill codes will be compacted with regular codes as far as possible, which can prevent lengthening the final schedule length and cost less power consumption. Then, the proposed method is evaluated on various target architecture to study the influence of differing number of resources on the scheduling result. It shows that accumulator is the most critical resource in non-orthogonal DSP architecture, because increasing the number of it is necessary to improve performance on both evaluation metrics. From our evaluation results, if the target architecture contains more than four accumulators, it is sufficient to keep most ALU results and eliminate almost all spill codes. Using more input registers or memory banks also can slightly reduce the instruction count. However, implementing additional memory banks and associated data buses requires heavy hardware costs. Thus, in view of their cost-performance, we recommend using additional accumulators to reduce the instruction count. As for instruction-level parallelism exploration, we conclude that numbers of data ALUs and accumulators must be concurrently increased. If only more data ALUs are added, accumulator spills will occur much frequently

and incur many spill codes. Besides, from evaluation results it also shows that two data ALUs in the target architecture is actually sufficient, since using our RSSA can generate shortest schedules in most MDFGs. Finally, we compare the efficiency among our method and some previous work. After analyzing execution complexities of scheduling phases for each method separately, it shows that RSSA is the most efficient one.

The remainder of this paper is organized as follows. Section 2 surveys the fundamental background and related studies. Our hypothetical target architecture and design motivations are also presented. Detailed principles and algorithms for the proposed method are introduced in Section 3. Section 4 contains our preliminary performance evaluations and brief description. Finally, conclusions and plans for future work are presented in Section 5.

2. Fundamental background

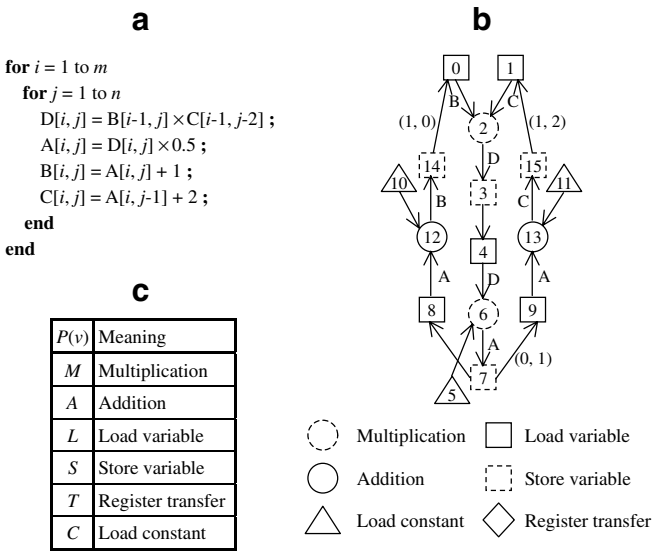
In this section, we describe some fundamentals, such as the program model, the retiming technique, and the target machine model. After surveying related studies, our design motivations are introduced.

2.1. Program model

Because multimedia and DSP applications are usually represented by uniform nested loops, they are commonly modeled by an MDFG. We define the MDFG to be the same as in Lee and Chen (2005), which is slightly different from previous studies (Lee and Chen, 2004; Zhuge et al., 2001).

Definition 2.1. An MDFG $G = (V, E, X, d, P)$ is a node-weighted and edge-weighted direct graph, where V is the set of computation nodes; $E \subseteq V \times V$ is the edge set that defines the precedence relations; $X(e)$ represents the variable accessed by an edge e; $d(e)$ is a function from E to Z^n representing the multi-dimensional delays between two nodes, where n is the number of dimensions; and $P(v)$ represents the node type (see Fig. 1(c)).

Fig. 1 shows an example of a nested loop and its corresponding MDFG. Nodes in the MDFG include ALU (multiplications and additions), memory access (load/store variable and load constant), and register transfer instructions. Note that an edge, e, that does not involve a memory access does not have a label $X(e)$. An MDFG is *realizable* if there exists a *schedule vector* s, such that $s \cdot d \geqslant 0$, where d are loop-carried dependencies (Lamport, 1974). An *iteration* is equivalent to executing each node in V exactly once. The *cycle period* is the period during which all nodes in an iteration are executed without resource constraints. It is also the maximum execution time among paths that have no delay, which will dominate the entire execution time of a nested loop. Note that many MDFGs can represent a single DSP application, depending on its representation by nested loops.



c

P(v)	Meaning
M	Multiplication
A	Addition
L	Load variable
S	Store variable
T	Register transfer
C	Load constant

Fig. 1. The MDFG example. (a) Nested loop in C code, (b) corresponding MDFG, (c) node types.

2.2. Retiming technique (Leiserson and Saxe, 1991)

Retiming is a popular technique used in loop scheduling that redistributes nodes in consecutive iterations to enhance the execution performance. The *retiming vector* $r(u)$, a function from V to Z^n, represents the offset between the original iteration and that after retiming. An MDFG $G_r = (V, E, X, d_r, P)$ is created after applying r, where the difference between G and G_r is only the delay vectors. Delay vectors will be changed accordingly to preserve the original dependencies.

A *prologue* is the instruction set that must be executed to provide necessary data for the iterative process. An *epilogue* is the complementary set that will be executed to complete the process. If the nested loop contains sufficient iterations, the time required for prologue and epilogue are negligible.

2.3. Hypothetical machine model

To conduct advanced compiler and architecture research, many parameterized architecture models are developed for simulation and evaluation (MESCAL; OptimoDE; ORC; Tensilica; Trimaran). Most of them are oriented towards *EPIC* (*explicitly parallel instruction computing*) or superscalar architectures, and support novel features such as prediction, control and data speculation, and memory hierarchy. However, none of them supports features like multiple memory banks and heterogeneous register sets. Therefore, in the following, we define a hypothetical machine model to represent a scalable non-orthogonal DSP architecture.

Fig. 2 shows the system overview of the Motorola DSP56000 (Motorola). This DSP is an example of non-orthogonal architecture, and is commonly used in practice

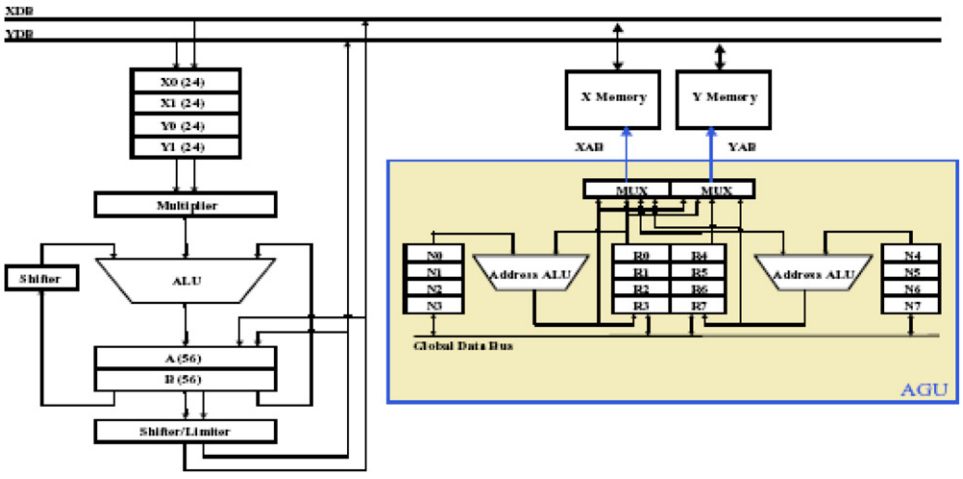


Fig. 2. Motorola DSP56000 architecture.

and research. As shown in Fig. 2, this processor consists of two memory banks, one data ALU, two accumulators, and four input registers (two for each memory bank). Our hypothetical machine model is generalized from the Motorola DSP56000. That is, in our design and experiments, we assume the target architecture contains N memory banks ($M_1 \ldots M_N$), k data ALUs ($ALU_1 \ldots ALU_k$), $k \times m$ accumulators ($acc_{11} \ldots acc_{1m}, \ldots, acc_{k1} \ldots acc_{km}$), and $N \times n$ input registers ($reg_{11} \ldots reg_{1n}, \ldots, reg_{N1} \ldots reg_{Nn}$). Similar to the Motorola DSP56000, the source operands for all ALU operations, except multiplication, must be input registers or accumulators and the destination operand must always be an accumulator. For multiplication, two source operands must always be input registers.

In the Motorola DSP56000 architecture, up to two move operations (including memory access, register transfer, and immediate load) and one data ALU instruction may be executed simultaneously in one cycle. Independent move operations executed in the same cycle are called *parallel moves*. However, due to the nature of the target architecture, only independent instructions satisfied *parallel move conditions* can be performed in parallel. In the following, we inherit these parallel move conditions and extend them to fit our assumed target architecture: (1) up to N independent move operations and k ALU instructions can be executed simultaneously in one cycle; (2) the N move operations reference data in different memory banks; (3) the N destination registers are different; (4) the M_i memory access load into restricted locations $reg_{i1} \ldots reg_{in}$, or all accumulators. Besides, about k ALU instructions being executed simultaneously, we define three conditions as follows: (1) $2k$ source operands must be read from different input registers or accumulators; (2) k destination operands must be stored in different accumulators; (3) the ALU instruction executed by ALU_i must store the generated ALU result in accumulators $acc_{i1} \ldots acc_{im}$. Note that we only list conditions specifically considered in the proposed method. Detailed parallel move conditions of the Motorola DSP56000 can be found in (Motorola).

2.4. Related work

A complete code generation algorithm for the architecture with multiple memory banks and a heterogeneous register set must include five phases: *intermediate representation*, *code compaction*, *instruction scheduling*, *memory bank assignment* (or *variable partition*), and *register/accumulator assignment* (Sudarsanam and Malik, 2000). These five phases can be performed in various sequences because they are logically independent. In addition, due to their extreme data dependences, more than one phase also can be considered simultaneously. However, code generation algorithms with tightly coupled phases are very time consuming, so we do not use this mechanism in our design method.

A number of papers have investigated the use of multi-bank memory to achieve maximum instruction level parallelism. (Leupers and Kotte, 2001; Saghir et al., 1994) focus on designing variable partitioning mechanisms, which try to evenly distribute memory accesses and explore the potential of higher memory bandwidth. For heterogeneous register sets, (Daveau et al., 2004; Scholz and Eckstein, 2002; Zhuang et al., 2004) present specific register allocation algorithms to fit their irregularity. Methods proposed in Lee and Chen (2004), Saghir et al. (1996), Wang and Hu (2004), Zhuge et al. (2001) solve both instruction scheduling and memory bank assignment problems, but do not consider the limitation of registers/accumulators. In addition, five methods (Cho et al., 2002; Kessler and Bednarski, 2002; Lee and Chen, 2005; Sudarsanam and Malik, 2000; Shiue, 2001) contain all above phases, and all except (Kessler and Bednarski, 2002) select Motorola DSP56000 as the target architecture. We describe three methods of them in some detail as follows.

In Cho et al. (2002), the main idea is applying the *graph coloring* approach to treat variable partition and register/accumulator assignment. For register/accumulator assignment, this phase is specially decoupled into two steps. It first classifies physical registers into a set of register

classes, and allocates each temporary variable to one of the register classes. Next, the graph coloring algorithm is applied to assign each temporary variable a physical register within the register class previously allocated to it. After generating compacted codes, a weighted undirected graph is constructed based on the sequence of variables referenced in these codes. Then, it identifies the *maximum spanning tree* (MST) of this graph, and assigns variables to each memory bank also using the graph coloring algorithm.

Sudarsanam and Malik (2000) is an example that simultaneously considers two phases. Because the Motorola DSP56000 has a heterogeneous register set, the memory access to each memory bank must load in a restricted set of locations. Therefore, (Sudarsanam and Malik, 2000) claims that variable partition and register/accumulator assignment phases should be performed simultaneously to maximally explore available parallelism among move operations. After generating compacted codes, an undirected graph is constructed representing constrained conditions on the register and memory bank assignments. Then, an algorithm based on *graph labeling* is used to determine both memory bank and register/accumulator assignments simultaneously.

Lee and Chen (2005) is our previous work, which contains features as follows. First, it has a procedure to generate uncompacted codes directly from a high-level language. In other related methods, this step is usually not included and performed using an existing tool. Second, it performs memory bank assignment before code compaction. In this case, memory accesses are scheduled with information of memory bank assignment. That is, the location conflict for parallel moves, in which two memory accesses are assumed to be executed in parallel but actually cannot be, will not occur. Third, it predicts the occurrence of accumulator spills and generates corresponding spill codes before code compaction. This feature forces spill codes to be scheduled in parallel with other instructions, which can prevent extension of the schedule length. Forth, it considers the limited number of registers during scheduling, so no extra action is required to check and handle the occurrences of register spills. Finally, it applies the retiming technique to reassign delays to explore the embedded parallelism of a loop.

2.5. Design motivation

After surveying related methods, we introduce our design motivations. In Lee and Chen (2005) we list two motivations: performing memory bank assignment before code compaction and predicting the occurrence of accumulator spills. We retain the former in the proposed method because it certainly can avoid the occurrence of spill codes. However, when the target architecture is not specific, using topological analysis of the input MDFG to predict accumulator spills becomes much more difficult and inaccurate. Therefore, in the new proposed method, we design another mechanism to solve accumulator spills and not predict their occurrence.

Next, we will consider the resolution of accumulator/register spills. When the register spill occurs, a variable currently residing in a register should be overwritten and reloaded again when required. If the overwritten variable is an ALU result transferred from an accumulator, it must be temporarily stored in the memory. As for the accumulator spill, except in memory, the overwritten ALU result can also be temporarily stored in a register before being used. Consider the following example. Assume the target architecture is the Motorola DSP56000, and we apply the method proposed in Cho et al. (2002) to schedule it. Fig. 3(a) and (b) show the sequence of assembly codes before and after doing code compaction. During the accumulator/register assignment phase, an accumulator spill is found at I_3. At that time, all variables residing in the four input registers will be used later (this algorithm schedules operations ASAP), so the overwritten ALU result, m, must be stored temporarily in memory. Later another memory access is added to reload m before I_8. Fig. 3(c) lists the final scheduling result using the method described in Cho et al. (2002). However, as shown in Fig. 3(d), if we move instruction i_{13} to I_5, variable m can be transferred to register Y0 instead of memory. In this case, one extra spill cost is eliminated and the schedule length reduces. From this example, we have found that it is preferable for an overwritten ALU result to be transferred to an input register when an accumulator spill occurs. In the new proposed method we will follow this principle to resolve accumulator spills.

In order to give an overwritten ALU result high priority to be temporarily stored in a register, input registers must be unfilled as far as possible while dealing with accumulator spills. However, according to the scheduling steps of (Cho et al., 2002; Sudarsanam and Malik, 2000; Shiue, 2001), accumulator spills are resolved after all operations been scheduled. In this case input registers will be occupied by source operands, which is unfavorable for inserting additional register transfers. Therefore, in the proposed method we divide the instruction scheduling phase into two steps, and let ALU operations be scheduled before memory accesses. Based on this mechanism, input registers can remain unfilled to store overwritten ALU results during dealing with accumulator spills, which is able to reduce additional spill costs.

Finally, we consider the time that register/accumulator been resolved during the entire code generation process. In Cho et al. (2002) and Sudarsanam and Malik (2000) they both perform this action at last, which means during instruction scheduling and code compaction the number of registers/accumulators are assumed to be unlimited. We think this case could have two flaws. One is the schedule length may be lengthened, because additional spill codes cannot be scheduled in parallel with other instructions. The scheduling result shown in Fig. 3(c) is such an example. Another is about its execution complexity. Although in Cho et al. (2002) and Sudarsanam and Malik

a

MOVE	a, reg0	(i_0)
MOVE	b, reg1	(i_1)
MPY	acc0, reg0, reg1	(i_2)
MOVE	c, reg2	(i_3)
ADD	acc1, acc0, reg2	(i_4)
MOVE	d, reg3	(i_5)
ADD	acc2, acc1, reg3	(i_6)
MOVE	acc2, p	(i_7)
MOVE	e, reg4	(i_8)
ADD	acc3, acc1, reg4	(i_9)
ADD	acc4, acc3, reg3	(i_{10})
MOVE	acc4, r	(i_{11})
ADD	acc5, acc3, reg0	(i_{12})
MOVE	f, reg5	(i_{13})
ADD	acc6, acc5, reg5	(i_{14})
ADD	acc7, acc0, acc6	(i_{15})
MOVE	acc7, t	(i_{16})

b

MOVE		X: a, reg0	Y: b, reg1	(I_0)
MPY	acc0, reg0, reg1	X: c, reg2	Y: d, reg3	(I_1)
ADD	acc1, acc0, reg2	X: e, reg4	Y: f, reg5	(I_2)
ADD	acc2, acc1, reg3			(I_3)
ADD	acc3, acc1, reg4	acc2, X: p		(I_4)
ADD	acc4, acc3, reg3			(I_5)
ADD	acc5, acc3, reg0		acc4, Y: r	(I_6)
ADD	acc6, acc5, reg4			(I_7)
ADD	acc7, acc0, acc6			(I_8)
MOVE		acc7, X: t		(I_9)

c

MOVE		X: a, X0	Y: b, Y0
MPY	A, X0, Y0	X: c, X1	Y: d, Y1
ADD	B, A, X1	X: e, X1	Y: f, Y0
MOVE		A, X: m	
ADD	A, B, Y1		
ADD	A, B, X1	A, X: p	
ADD	B, A, Y1		
ADD	B, A, X0		B, Y: r
ADD	A, B, Y0		
MOVE		X: m, X0	
ADD	B, A, X0		
MOVE		B, X: t	

d

MOVE		X: a, X0	Y: b, Y0
MPY	A, X0, Y0	X: c, X1	Y: d, Y1
ADD	B, A, X1	X: e, X1	
MOVE			A, Y0
ADD	A, B, Y1		
ADD	A, B, X1	A, X: p	
ADD	B, A, Y1		Y: f, Y1
ADD	B, A, X0		B, Y: r
ADD	A, B, Y1		
ADD	B, A, Y0		
MOVE		B, X: t	

Fig. 3. An example of code compaction. (a) Uncompacted code, (b) compacted code, (c, d) two scheduling results after resource assignment.

(2000) they do not present detailed mechanisms to insert spill codes, they definitely require an independent step to do this action. If the system contains strict resource constraints, this step may cost considerable time. Therefore, in the new proposed method, we want to not only design efficient mechanisms to determine and resolve register/accumulator spills, but also integrate them into the instruction scheduling and code compaction phases. That is, during instruction scheduling we have considered resource constraints already, and then no extra action is required to insert spill codes. Detailed descriptions of instruction scheduling and spill codes inserting will be listed in the next section.

3. Rotation scheduling with spill codes avoiding (RSSA)

In this section, we introduce our proposed method named *Rotation Scheduling with Spill Codes Avoiding* (*RSSA*). Section 3.1 contains some basic assumptions and scheduling principles. Detailed steps of RSSA are introduced in Section 3.2.

3.1. Preliminaries

To correctly execute an MDFG, its data dependencies cannot be violated. Because the non-orthogonal architecture has additional constraints on the use of resources, registers and accumulators must be used very carefully to preserve data dependencies. An operand residing in a register/accumulator obviously cannot be overwritten before being used. Corresponding to the nature of our hypothetical architecture in Section 2.3, we list three scheduling principles to satisfy the above rule. For convenience, we only permit a variable loaded from memory to be stored in an input register.

1. For an edge e_{ij} of an MDFG with zero delay, if $P(v_i) = L/C/T$ and $P(v_j) = M/A$, v_j must be executed no later than the next n node (in the same memory bank as v_i) with type $L/C/T$.
2. For an edge e_{ij} of an MDFG with zero delay, if $P(v_i) = M/A$ and $P(v_j) = S/T$, v_j must be executed no

later than the next m ALU instructions (in the same data ALU as v_i).

3. For an edge e_{ij} of an MDFG with zero delay, if v_i and v_j are both ALU instructions and executed in the same data ALU, at most $m-1$ ALU instructions can be executed between v_i and v_j.

We will propose our code generation algorithm RSSA consistent with these principles. Moreover, some operands of ALU instructions may be constants in DSP applications. These constants can be intuitively loaded using immediate load instructions. However, due to parallel move conditions, an immediate load is rarely executed in parallel with other independent move operations. Therefore, we use *load constant* instead of immediate load in our method. We assume constants are stored in all memory banks at specific locations in advance. The load constant instruction will load constants directly from a specific address, which is essentially equivalent to the original load variable instruction. Meanwhile, the load constant can be scheduled for any memory bank to increase performance.

3.2. Detailed algorithms of RSSA

From the related fundamentals and preliminaries above, we introduce our general code generation algorithm RSSA in the following. As mentioned in Section 1, both exploring the instruction-level parallelism and avoiding generating too many spill codes are important issues to design an effective code generation algorithm for non-orthogonal DSP architecture. In our RSSA we will achieve these two goals at the same time. As shown in Fig. 4, we divide the overall algorithm into five main parts: MDFG construction, TDAG construction, instruction scheduling (I), instruction scheduling (II), and initial schedule retiming. Note that RSSA is also an extension of our previous study. Thus, we will inherit some definitions and algorithms from Lee and Chen (2005) and then design new mechanisms to improve it.

3.2.1. MDFG construction

The first part essentially contains two phases, including MDFG construction and memory bank assignment. Both two phases are inherited from Lee and Chen (2005). In addition, during the construction of the MDFG, operands are stored in memory and reloaded into registers only when they are required for use. That is, an ALU instruction in high-level language corresponds to four nodes in the MDFG, and three of these are move operations. This mechanism appears burdensome but is really used in some DSP compilers, because the number of registers is limited in DSP and memory is the only safe repository. We use Fig. 5 to illustrate the relationship between the high-level language and the MDFG in Fig. 1.

For memory bank assignment the three mechanisms proposed in Lee and Chen (2004) and Zhuge et al. (2001) can be chosen. Similarly, if we select the mechanisms proposed in Lee and Chen (2005), the MDFG must be unfolded or tiled according to the number of memory banks.

1. G_c = Construct MDFG;
2. Partition variables to memory banks;
3. Unfold or tile G_c if necessary;
4. G_t = Construct TDAG (G_c);
5. S = Schedule all instructions except memory loads (G_t);
   5.1. G_{op} = Construct DAG G_{op} (G_t);
   5.2. S = Schedule nodes in G_{op} (G_{op});
   5.3. S = Determine and solve accumulator spills (S, G_{op});
6. S = Schedule memory load instructions (S, G_t);
7. S = Retime the initial scheduling result (S, G_t);

Fig. 4. The overall scheduling algorithm.

3.2.2. TDAG construction

According to the algorithm we use to construct the MDFG, clearly register and accumulator spills will not occur if all operations in the MDFG are scheduled. However, this MDFG is too *complete* to degrade the computational performance, because ALU results can be temporarily stored in accumulators or registers. In Lee and Chen (2005) we define a *translated data acyclic graph* (*TDAG*) constructed from the original MDFG, which is used to remove possible unnecessary memory accesses. In this method, we retain the definition and slightly modify the constructing algorithm of a TDAG. Fig. 6 shows the modified TDAG construction algorithm, and Fig. 7(a) is the TDAG corresponding to the MDFG in Fig. 1(b).

Definition 3.1. A *translated data acyclic graph* (*TDAG*) $G=(V,E,X,P)$ is a node-weighted and edge-weighted direct graph, where V is the set of computation nodes; $E \subseteq V \times V$ is the edge set that defines the precedence relations over the nodes in V; $X(e)$ represents the variable accessed by an edge e; $P(v)$ represents the type of node v (see Fig. 1(c)).

Note that in this construction algorithm, if an operand is shared by an addition and a multiplication, a register transfer instruction is inserted before addition as shown

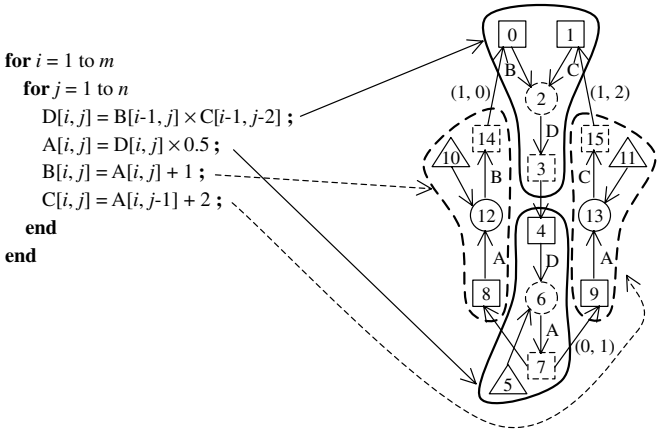


Fig. 5. The relationship between high-level language and the MDFG.

1. Input: $G_c = (V_c, E_c, X_c, d, P_c)$;
2. Output: $G_t = (V_t, E_t, X_t, P_t)$;
3. $V_t = V_c$; $E_t = \{e \mid e \in E_c, d(e) = (0,\ldots, 0)\}$;
4. Assume that v_i, v_j, v_k, v_l, v_m, $v_n \in V_c$, and their types are M, A, S, L, M, and A respectively;
   - 4.1. If ($\exists$a path $v_i \rightarrow v_k \rightarrow v_l \rightarrow v_m \in G_t$) // $M \rightarrow M$
     Insert node v_x into V_t (set $P_t(v_x) = T$); Insert edge e_{ix} into E_t;
     $\forall e_{lm} \in E_t$ delete edges e_{lm} from E_t, insert edges e_{xm} into E_t;
     Delete node v_l from V_t; Delete edge e_{kl} from E_t;
     If ($\exists e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain v_k, e_{ik}
     Else delete node v_k from V_t, delete edge e_{ik} from E_t;
   - 4.2. If ($\exists$a path $v_j \rightarrow v_k \rightarrow v_l \rightarrow v_m \in G_t$) // $A \rightarrow M$
     Insert node v_x into V_t (set $P_t(v_x) = T$); Insert edge e_{jx} into E_t;
     $\forall e_{lm} \in E_t$ delete edges e_{lm} from E_t, insert edges e_{xm} into E_t;
     Delete node v_l from V_t; Delete edge e_{kl} from E_t;
     If ($\exists e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain v_k, e_{jk}
     Else delete node v_k from V_t, delete edge e_{jk} from E_t;
   - 4.3. If ($\exists$a path $v_i \rightarrow v_k \rightarrow v_l \rightarrow v_n \in G_t$) // $M \rightarrow A$
     $\forall e_{ln} \in E_t$ delete edges e_{ln} from E_t, insert edges e_{in} into E_t;
     Delete node v_l from V_t; Delete edge e_{kl} from E_t;
     If ($\exists e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain v_k, e_{ik}
     Else delete node v_k from V_t, delete edge e_{ik} from E_t;
   - 4.4. If ($\exists$a path $v_j \rightarrow v_k \rightarrow v_l \rightarrow v_n \in G_t$) // $A \rightarrow A$
     $\forall e_{ln} \in E_t$ delete edges e_{ln} from E_t, insert edges e_{jn} into E_t;
     Delete node v_l from V_t; Delete edge e_{kl} from E_t;
     If ($\exists e_{kl} \in E_c$ such that $d(e_{kl}) \neq (0,\ldots, 0)$) ; // retain v_k, e_{jk}
     Else delete node v_k from V_t, delete edge e_{jk} from E_t;
5. $X_t(e) = X_c(e)$, if e is remained in E_t;
6. $P_t(v) = P_c(v)$, if v is remained in V_t;
7. Return G_t;

Fig. 6. The TDAG constructing algorithm.

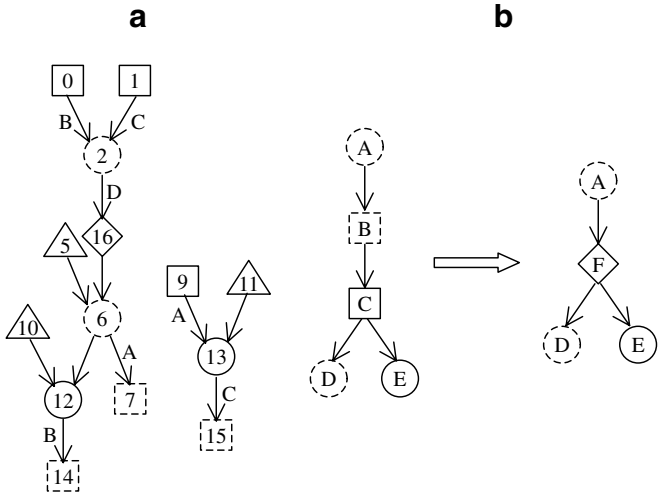


Fig. 7. The TDAG example. (a) Corresponding TDAG of the MDFG in Fig. 1(b), (b) the TDAG constructing example with shared operand.

in Fig. 7(b). That is, although node **E** actually can be scheduled consecutively with node **A**, it will be delayed at least one time step to wait the execution of node **F**.

3.2.3. Instruction scheduling (I)

As described in Section 2.5, we wish to schedule ALU instructions before memory accesses to reduce spill costs incurred by accumulator spills. This mechanism was applied in our previous study (Lee and Chen, 2005) and in RSSA we modify it further to become more efficient. In the third part of RSSA, we schedule all instructions except those with type L/C. This part contains three phases: constructing graph G_{op}, scheduling nodes in G_{op}, and resolving accumulator/register spills. In the following, we introduce each of them in detail.

Definition 3.2. A DAG $G_{op} = (V, E, X, P)$ is a direct graph, where V is the node set representing ALU instructions, register transfers, and store variables; $E \subseteq V \times V$ is the edge set that defines the precedence relations over nodes in V; $X(e)$ represents the variable accessed by an edge e; $P(v)$ represents the type of node v.

The graph G_{op} defined above is constructed from the TDAG that contains nodes of types M, A, T, and S. Fig. 8(a) is another example of TDAG G_t, and its corresponding G_{op} is shown in Fig. 8(b). Next, we simply schedule the nodes in G_{op} using *list scheduling*, assuming the number of accumulators/registers are unlimited. Because

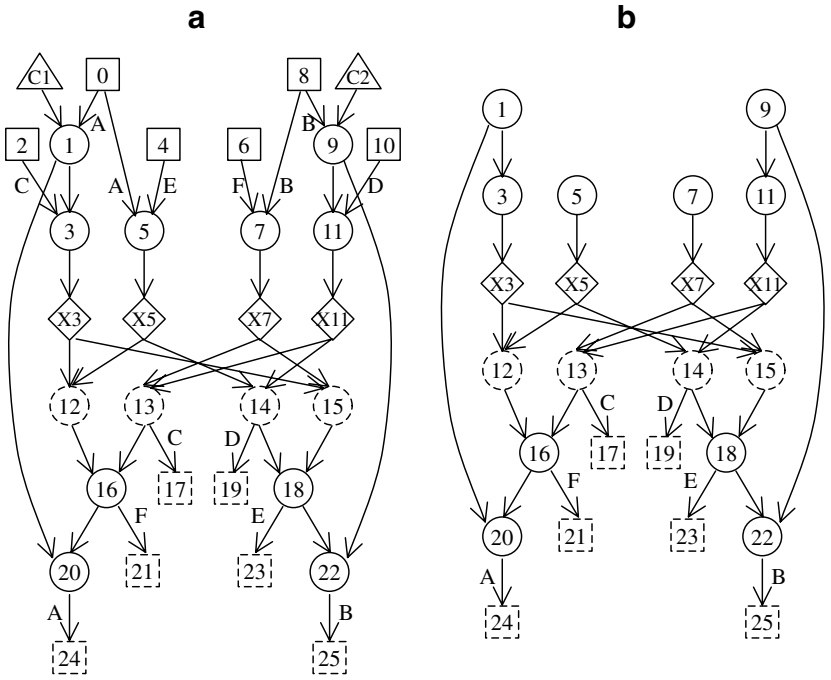


Fig. 8. The G_{op} example. (a) TDAG and (b) corresponding G_{op}.

the limited quantities of accumulators/registers are not yet considered, the current G_{op} scheduling result has minimum length *len*. Fig. 9(a) shows the scheduling result of G_{op} in Fig. 8(b) with only one data ALU.

Then, we describe the new mechanism to resolve accumulator spills. Before doing this the time steps at which accumulator spills occur must be determined. Our idea is to calculate the number of accumulators and input registers used at every time step, and determine accumulator/register spills from these results. Six variables defined for this mechanism are listed in Table 1.

We describe the time interval for which a value must reside in an accumulator or input register for a correct schedule. If an ALU result is defined at time step p and used at time step q, it will occupy the accumulator from time steps p to $q-1$. Similarly, if a variable (or constant) is loaded from memory at time step p and used at time step q, it will occupy the register from time steps p to $q-1$. Fig. 9(a) also shows variables defined in Table 1 for that G_{op} scheduling result. In this example, assume the target architecture contains two accumulators acc_{11} and acc_{12}, we find that accumulator spills occur at time steps 6–9 and 12.

As described in our design motivation, overwritten ALU results are preferably stored in input registers when accumulator spills occur. Therefore, in this mechanism we first transfer all overwritten ALU results to input registers, and temporarily store them to memory only when the number of registers is insufficient. We use the example in Fig. 9(a) to illustrate this. At time step 6, in order to release an accumulator for node 11, we must select a node v from *acclist*_1(6) and transfer the ALU result generated by v to a register. In this situation, we want to transfer a value that will release an accumulator with the longest time free of use. After checking the content of *uselist*(u) for all nodes u in *acclist*_1(6), node 9 is selected to transfer its value from accumulator to input register. Next, an additional register transfer, **X9**, is scheduled at memory bank M_2 at time step 6, because *reg*_1(6) is smaller than *reg*_2(6). All variables defined in Table 1 are changed accordingly. Then, we select other nodes to transfer if necessary until all accumulator spills are resolved. In this example, **X1** is scheduled to M_1 at time step 8. Fig. 9(b) shows the modified G_{op} scheduling result without accumulator spills.

From Fig. 9(b), assume there are four input registers, reg_{11}, reg_{12}, reg_{21}, and reg_{22}, in the target architecture, we find that register spills occur at time steps 8–9 at M_1 and time steps 7–10 at M_2. Therefore, some values need to be stored temporarily in memory and reloaded before use. Note that registers reg_{ij} are dedicated for use for referencing data from M_i in our hypothetical architecture, therefore register spills occurring at each memory bank have to be resolved separately. At time step 8, we must select a node v from *reg*_1(8) to store temporarily. In this situation, we also want to store a value that will be used at latest, in order to release an input register with the longest time interval. After checking the content of *uselist*(u) for all nodes u in *reglist*_1(8), node **X3** is selected. Then, operations **S3** and **L3** are scheduled at time steps 7 and 11 respectively, because this value is used by nodes 12 and 15. Similarly, we change variables defined in Table 1 accordingly, and repeat this step until all register spills are resolved. The G_{op} scheduling result without any accumulator/register spill is shown in Fig. 9(c). Moreover, if a value selected to temporarily store is not yet used, the added store operation can directly replace the corresponding register transfer. The operation **S9** in Fig. 9(c) is such an example.

a

t	ALU_1	M_1	M_2	acc_1	reg_1	reg_2	acclist_1	reglist_1	reglist_2
1	1			1	0	0			
2	3			2	0	0	1		
3	5	X3		2	1	0	1	X3	
4	7		X5	2	1	1	1	X3	X5
5	9	X7		2	2	1	1	X3, X7	X5
6	11			3	2	1	1, 9	X3, X7	X5
7	12		X11	3	2	2	1, 9	X3, X7	X5, X11
8	13			4	2	2	1, 9, 12	X3, X7	X5, X11
9	16	17		3	2	2	1, 9	X3, X7	X5, X11
10	20		21	2	2	2	9	X3, X7	X5, X11
11	14	24		2	2	0	9	X3, X7	
12	15		19	3	0	0	9, 14		
13	18			2	0	0	9		
14	22	23		1	0	0			
15			25	0	0	0			

b

t	ALU_1	M_1	M_2	acc_1	reg_1	reg_2	acclist_1	reglist_1	reglist_2
1	1			1	0	0			
2	3			2	0	0	1		
3	5	X3		2	1	0	1	X3	
4	7		X5	2	1	1	1	X3	X5
5	9	X7		2	2	1	1	X3, X7	X5
6	11		X9	2	2	2	1	X3, X7	X5, X9
7	12		X11	2	2	3	1	X3, X7	X5, X9, X11
8	13	X1		2	3	3	12	X1, X3, X7	X5, X9, X11
9	16	17		1	3	3		X1, X3, X7	X5, X9, X11
10	20		21	1	2	3		X3, X7	X5, X9, X11
11	14	24		1	2	1		X3, X7	X9
12	15		19	2	0	1	14		X9
13	18			1	0	1			X9
14	22	23		1	0	0			
15			25	0	0	0			

Fig. 9. Scheduling result. (a) G_{op} nodes only without resource constraints, (b) G_{op} nodes only with unlimited number of input registers, (c) G_{op} nodes only without accumulator spills, (d) the initial scheduling result of G_t and (e) the retimed scheduling result of G_t.

Consider the variability of the schedule length. After resolving accumulator spills using only additional register transfers, the schedule will not be lengthened. That is, the length of obtained G_{op} scheduling result is also *len* if register spills never occur. However, while resolving register spills, an additional memory access may not be successfully scheduled due to insufficient time steps. In this situation an extra time step is inserted to schedule this instruction individually as late as possible. Because we only lengthen the schedule when required, the obtained G_{op} scheduling result still has minimum length. Until this phase we obtain a schedule with length *len*' that contains all nodes in G_t except those with types L/C. In the next part, the remaining memory load instructions will be inserted to complete the initial schedule of G_t.

3.2.4. Instruction scheduling (II)

Before scheduling memory load instructions, we present the theoretical minimum schedule length of G_t that can be achieved. Strictly speaking, the minimum schedule length of G_t equals to *len*' + 1, where the additional time step is used to load source operands for the first ALU operation. Therefore, the goal of this part is to keep the schedule length no longer than *len*' + 1 if possible.

We inherit the scheduling mechanism for memory load instructions from (Lee and Chen, 2005). Its main feature is to consider the limited number of registers during scheduling, therefore no extra action is required to check and deal with register spills. In the following we list our scheduling rules for memory accesses.

1. According to the execution sequence of ALU instructions, schedule their predecessors as soon as possible.
2. Scheduling principle 1 listed in Section 3.1 must be satisfied, and *reg_i*(t) cannot exceed the number of registers at any time step.
3. If a memory load instruction cannot be scheduled successfully due to insufficient registers, a variable currently residing in a register is selected for storing and reloading using the mechanism described in Section 3.2.3.
4. For previous rule, if the selected variable is not transferred from an accumulator, the additional store variable instruction is unnecessary.

c

t	ALU_1	M_1	M_2	acc_1	reg_1	reg_2	acclist_1	reglist_1	reglist_2
1	1			1	0	0			
2	3			2	0	0	1		
3	5	X3		2	1	0	1	X3	
4	7		X5	2	1	1	1	X3	X5
5	9	X7		2	2	1	1	X3, X7	X5
6	11		S9	2	2	1	1	X3, X7	X5
7	12	S3	X11	2	1	2	1	X7	X5, X11
8	13	X1		2	2	2	12	X1, X7	X5, X11
9	16	17		1	2	2		X1, X7	X5, X11
10	20	L3	21	1	2	2		L3, X7	X5, X11
11	14	24		1	2	0		L3, X7	
12	15		19	2	0	0	14		
13	18		L9	1	0	1			L9
14	22	23		1	0	0			
15			25	0	0	0			

d

t	ALU_1	M_1	M_2	acc_1	reg_1	reg_2	acclist_1	reglist_1	reglist_2
1		0	C1	0	1	1		0	C1
2	1	2	6	1	2	1		0, 2	6
3	3	4	8	2	2	2	1	0, 4	6, 8
4	5	X3		2	1	2	1	X3	6, 8
5	7	C2	X5	2	2	2	1	C2, X3	8, X5
6	9	X7	10	2	2	2	1	X3, X7	10, X5
7	11		S9	2	2	1	1	X3, X7	X5
8	12	S3	X11	2	1	2	1	X7	X5, X11
9	13	X1		2	2	2	12	X1, X7	X5, X11
10	16	17		1	2	2		X1, X7	X5, X11
11	20	L3	21	1	2	2		L3, X7	X5, X11
12	14	24		1	2	0		L3, X7	
13	15		19	2	0	0	14		
14	18		L9	1	0	1			L9
15	22	23		1	0	0			
16			25	0	0	0			

e

t	ALU_1	M_1	M_2	acc_1	reg_1	reg_2	acclist_1	reglist_1	reglist_2
1	3	4	8	2	2	2	1	0, 4	6, 8
2	5	X3		2	1	2	1	X3	6, 8
3	7	C2	X5	2	2	2	1	C2, X3	8, X5
4	9	X7	10	2	2	2	1	X3, X7	10, X5
5	11		S9	2	2	1	1	X3, X7	X5
6	12	S3	X11	2	1	2	1	X7	X5, X11
7	13	X1		2	2	2	12	X1, X7	X5, X11
8	16	17		1	2	2		X1, X7	X5, X11
9	20	L3	21	1	2	2		L3, X7	X5, X11
10	14	24	C1	1	2	1		L3, X7	C1
11	15	0	19	2	1	1	14	0	C1
12	18	2	L9	1	2	2		0, 2	C1, L9
13	22	23	6	1	2	2		0, 2	6, C1
14	1		25	1	2	1	1	0, 2	6

Fig. 9 (*continued*)

5. If a memory load instruction cannot be scheduled successfully due to insufficient time steps, an additional time step is inserted to schedule this instruction individually as late as possible as in Section 3.2.3.

Based on the above rules, all memory load instructions can be successfully scheduled. Fig. 9(d) shows the scheduling result of G_t in Fig. 8(a). The schedule is still lengthened only when required; consequently, the obtained G_t scheduling result is also with minimum length. Finally, because we

Table 1
Variables defined for solving accumulator/register spills

Variable	Type	Definition
$sch(v)$	integer	the time step that G_{op} node v is executed
$uselist(v)$	integer list	time steps that G_{op} node v is used
$acc\_i(t)$	integer	the number of accumulators acc_{ij}, for $j = 1 \ldots m$, been used at time step t
$acclist\_i(t)$	node list	G_{op} nodes with types M or A whose generated ALU results reside in accumulators acc_{ij}, for $j = 1 \ldots m$, at time step t, except the one that are executed in ALU_i at time step t
$reg\_i(t)$	integer	the number of registers reg_{ij}, for $j = 1 \ldots n$, been used at time step t
$reglist\_i(t)$	node list	G_{op} nodes with type T whose transferred ALU results reside in registers reg_{ij}, for $j = 1 \ldots n$, at time step t

have already considered accumulator/register spills, an appropriate assignment of physical accumulators/registers certainly exists.

3.2.5. Initial schedule retiming

From the previous four parts an initial schedule of G_t without any accumulator/register spills is obtained. In the last part of RSSA, we use the retiming technique to explore potential parallelism among iterations. After applying the retiming technique to redistribute nodes in consecutive iterations, the length of the final scheduling result may be less than the critical path of the original G_t. If we assume the target architecture contains unlimited resources, instructions can be rescheduled as soon as possible without violating precedence relations. However, in the real design case, the use of accumulators/registers has to be considered. Therefore, the requirement to guarantee an appropriate assignment of physical accumulators/registers still exists for the retimed scheduling result, and the use of resources cannot exceed their limited number at any time step. Assume that the G_t scheduling result obtained above has length *len_f*. Variables defined in Table 1 are also used to determine at which time step a retimed instruction can be scheduled.

We first describe after rescheduling an instruction at time step t, the time interval that its corresponding value must reside in an accumulator or register. If the retimed instruction is of type $L/C/T$ it must occupy a register from time step t to *len_f* because this value will be used for the later iteration. Similarly, the retimed ALU instruction must occupy an accumulator from time step t to *len_f*. For a retimed instruction, we reschedule it at the earliest time step that satisfies precedence relations and will not cause any accumulator/register spill. Meanwhile, variables defined in Table 1 are also updated after rescheduling an ALU instruction, because some resources will be released by its predecessors. In addition, a load constant can be rescheduled for any memory bank to achieve higher performance, because we store constants in all memory banks in advance. The final retimed G_t scheduling result of Fig. 9(d) is shown in Fig. 9(e).

4. Preliminary performance evaluation

In Section 4.1, we evaluate the proposed RSSA and compare it with previous work using several selected MDFGs. Then, RSSA is experimented with various hypothetical architectures in Section 4.2 to study the influence of different number of resources. After presenting evaluation results, some brief summaries of the effectiveness and efficiency of our method are presented in Section 4.3.

4.1. Comparison with previous work

At first we set our machine model equivalent to the Motorola DSP56000 to compare RSSA with related work. Several one-dimensional or two-dimensional MDFGs are selected from previous studies, which represent nested loops with depth one or two, used in DSP applications. Nevertheless, our RSSA can be easily extended to cover nested loops with depths greater than two. Basically, based on three variable partitioning mechanisms proposed in algorithms RSVR (Zhuge et al., 2001), RSF (Lee and Chen, 2004), and RST (Lee and Chen, 2004), using our method can obtain three scheduling results. However, for one-dimensional MDFG only the first two mechanisms are applied since algorithm RST is designed specifically for nested loops with depths greater than two. As mentioned in Section 2.4, many previous studies only consider specific code generation phases of non-orthogonal DSP architecture due to their independent character. In this paper we compare our method with three methods presented in Cho et al. (2002), Shiue (2001), and Sudarsanam and Malik (2000), because these methods are all design for Motorola DSP56000 and contain code generation phases similar to ours. In addition, methods presented in Lee and Chen (2004) are also compared and evaluated, after inserting necessary spill codes.

In this paper, we use two metrics including schedule length and instruction count to evaluate the performance at the same time. Shorter schedule length obviously indicates shorter execution time for the given nested loop. On the other hand, less instructions been executed indicates not only less power consumption, but also less memory space required to store (smaller code size). Table 2 lists schedule lengths of one iteration for selected MDFGs. Note that some values in this table are fractional. This is because the MDFG is unfolded or tiled using two variable partitioning mechanisms proposed in Lee and Chen (2004), and we show the average schedule length in an original iteration. In Table 2 we can see that our RSSA usually gets the shortest schedule lengths compared to other methods. The main reason is the usage of retiming technique, which reassigns instructions in consecutive iterations to explore potential parallelism among iterations. As more compact codes are generated, system resources are fully utilized and schedule lengths are shortened.

Table 3 displays the instruction count in each iteration. From this table we can find that both RSSA and Cho et al.

Table 2
Schedule lengths obtained by different code generation algorithms

Benchmarks	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Wave digital filter	7	9	9	8	6	8.5	6	5	5.5
Filter	8	13	13	9	11.5	9	6	5.5	5
IIR filter 2D	20	29	33	25	27.5	28	16	16	16
Forward-substitution	7	12	12	9	10	11.5	5	5.5	5
THCS	6	8	8	6	6.5	5.5	4	4	4
DFT	16	21	21	18	21	18.5	13	12.5	13
Floyd-Steinberg	20	36	37	29	32.5	32	18	17.5	17
Transmission line	15	20	21	19	18	18	12	12	12
IIR filter 1D	11	15	15	11	14	–	8	8	–
Differential equation solver	16	20	21	18	21.5	–	13	11.5	–
All-pole lattice filter	21	37	37	35	28	–	17	16	–
Elliptic filter	42	62	66	56	69	–	36	34	–

[1] Cho et al. (2002).
[2] Sudarsanam and Malik (2000).
[3] Shiue (2001).
[4] Lee and Chen, 2004 (with RSVR mechanism).
[5] Lee and Chen, 2004 (with RSF mechanism).
[6] Lee and Chen, 2004 (with RST mechanism).
[7] Proposed (with RSVR mechanism.
[8] Proposed (with RSF mechanism).
[9] Proposed (with RST mechanism).

Table 3
Number of operations really executed in an iteration obtained by different code generation algorithms

Benchmarks	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Wave digital filter	14	16	16	16	15.5	16	14	13	13
Filter	11	16	16	16	16	16	11	10.5	10.5
IIR filter 2D	37	64	68	64	64	64	37	37	37
Forward-substitution	11	20	20	18	17.5	18	11	10.5	10.5
THCS	10	16	16	14	14.5	14	10	9.5	10
DFT	33	48	49	44	44	44	32	30	31.5
Floyd-Steinberg	39	68	70	59	59	59.5	39	39.5	39.5
Transmission line	28	48	48	42	42	42	29	29	29
IIR filter 1D	20	32	32	30	29.5	–	18	18	–
Differential equation solver	25	44	44	37	37	–	26	25.5	–
All-pole lattice filter	37	60	60	51	51.5	–	35	34.5	–
Elliptic filter	77	136	136	125	116.5	–	75	72	–

(2002) generate much less instruction counts than other methods. The reason caused this result is whether methods use accumulator/register to store temporary variables. Except our proposed method and Cho et al. (2002), all other algorithms schedule instructions directly based on MDFGs. That is, operands are stored in memory and reloaded into registers only when they are required for use. As we have introduced in Section 3.2.2, although spill codes are rarely inserted during scheduling in this case, many memory accesses are really unnecessary. RSSA contains steps to remove such memory accesses and Cho et al. (2002) also has similar mechanism. Therefore, instruction counts generated by these two algorithms can be kept relatively low. In Section 4.3 we will further describe the effectiveness of our proposed algorithm on both two metrics.

4.2. The influence of resources

To harvest the benefits provided by the non-orthogonal DSP architecture, an effective code generation algorithm which can fully utilize system resources is obviously essential. However, in order to explore the instruction-level parallelism and reduce occurrences of accumulator/register spills, increasing the number of resources is a more direct way. Thus, in the following, we set our parameterized machine model to simulate architectures with different number of resources, and evaluate MDFGs using the general method RSSA. Scheduling results affected by different kinds of resources will be studied on both evaluation metrics.

We first list some preliminaries. All instructions are still assumed that can be completed in one time step. After transferring selected MDFGs to TDAGs using RSSA, Table 4 lists the number of ALU instructions, the critical path length, and the number of nodes in every TDAG. These data can be used as lower bounds of scheduling results. If the schedule length is equal to or less than the critical path of the corresponding TDAG, it indicates that the shortest schedule can be obtained. The reason to achieve a schedule with length shorter than critical path

Table 4
Characteristics of selected TDAGs

Benchmarks	Number of ALU nodes in G_t	Critical path of G_t	Number of nodes in each TDAG				
			original	unfold 2	tiled 2×1	unfold 3	tiled 3×1
Wave digital filter	4	6	14	13	13.5	12.6	13.3
Filter	4	7	11	10.5	10.5	10.3	10.3
IIR filter 2D	16	7	34	34	34	34	34
Forward-substitution	5	6	11	10.5	10.5	10.3	10.3
THCS	4	4	10	9.5	10	9.3	10
DFT	12	7	32	28	30	26.6	29.3
Floyd–Steinberg	17	12	38	37.5	37.5	37.3	37.3
Transmission line	12	10	26	26	26	26	26
IIR filter 1D	8	6	17	17	–	16.6	–
Differential equation solver	11	11	25	22.5	–	21.6	–
All-pole lattice filter	15	18	33	29	–	27.6	–
Elliptic filter	34	19	65	58	–	55.6	–

is the usage of retiming technique. On the other hand, if an iteration consists of exactly the same number of nodes as the corresponding TDAG, no spill codes are inserted. In Tables 5–8 shown below, we use shaded values to represent a schedule with shortest length or without any spill code. Moreover, when either the schedule length or the number of executed operations is improved by additional resources, the improved result is shown as a bold value.

Table 5 shows results of different number of accumulators. In this table we find that the instruction count decreases obviously when the target architecture contains more accumulators. This result indicates that accumulator spills occur very often. If more ALU results can reside in additional accumulators, spill codes will be reduced due to less occurrences of accumulator spill. Furthermore, since fewer overwritten ALU results are temporarily stored in input registers, occurrences of register spill also can be reduced. As for the schedule length, it is only slightly shortened. This result presents that to increase the number of accumulators cannot explore the instruction-level parallelism.

Then, Table 6 shows results of different number of input registers. These results indicate that neither the schedule length nor the instruction count can be obviously improved. Similarly, the instruction-level parallelism cannot be explored by using more input registers. We also find that the accumulator spills will not occur less, because the number of accumulators has not been increased in this table. After transferring overwritten ALU results to input registers, although more of them can be temporarily stored

Table 5
Experimental results, with target architectures contains different number of accumulators

	1 ALU, 2 acc, 4 reg, 2 mem						1 ALU, 3 acc, 4 reg, 2 mem					
	RSVR		RSF		RST		RSVR		RSF		RST	
	len	#	*len*	#	*len*	#	*len*	#	*len*	#	*len*	#
Wave Digital Filter	6	14	5	13	5.5	13	6	14	5	13	5.5	13
Filter	6	11	5.5	10.5	5	10.5	6	11	5.5	10.5	**4.5**	10.5
IIR 2D	16	37	16	37	16	37	16	**34**	16	**34**	16	**34**
Forward-substitution	5	11	5.5	10.5	5	10.5	5	11	5.5	10.5	5	10.5
THCS	4	10	4	9.5	4	10	4	10	4	9.5	4	10
DFT	13	32	12.5	30	13	31.5	**12**	32	**12**	**28.5**	**12**	**31**
Floyd-Steinberg	18	39	17.5	39.5	17	39.5	18	**38**	17.5	**38.5**	17	**38.5**
Transmission Line	12	29	12	29	12	29	12	**27**	12	**27**	12	**27**
IIR 1D	8	18	8	18	--	--	8	**17**	8	**17**	--	--
Equation Solver	13	26	11.5	25.5	--	--	13	26	11.5	**25**	--	--
All-pole Lattice Filter	17	35	16	34.5	--	--	17	**33**	16	**31.5**	--	--
Elliptic Filter	36	75	34	72	--	--	**35**	**70**	**30.5**	**66.5**	--	--

Table 6
Experimental results, with target architectures contains different number of input registers

	1 ALU, 2 acc, 4 reg, 2 mem						1 ALU, 2 acc, 6 reg, 2 mem					
	RSVR		RSF		RST		RSVR		RSF		RST	
	len	#	*len*	#	*len*	#	*len*	#	*len*	#	*len*	#
Wave Digital Filter	6	14	5	13	5.5	13	6	14	5	13	**5**	13
Filter	6	11	5.5	10.5	5	10.5	**5**	11	**5**	10.5	**4**	10.5
IIR 2D	16	37	16	37	16	37	16	37	16	37	16	37
Forward-substitution	5	11	5.5	10.5	5	10.5	5	11	**5**	10.5	5	10.5
THCS	4	10	4	9.5	4	10	4	10	4	9.5	4	10
DFT	13	32	12.5	30	13	31.5	13	32	12.5	**28.5**	**12**	**30.5**
Floyd-Steinberg	18	39	17.5	39.5	17	39.5	18	39	17.5	39.5	17	39.5
Transmission Line	12	29	12	29	12	29	12	29	12	29	12	29
IIR 1D	8	18	8	18	--	--	8	18	8	18	--	--
Equation Solver	13	26	11.5	25.5	--	--	13	**25**	12	**23**	--	--
All-pole Lattice Filter	17	35	16	34.5	--	--	17	35	16	**33**	--	--
Elliptic Filter	36	75	34	72	--	--	**35**	**73**	**30.5**	**69.5**	--	--

Table 7
Experimental results, with target architectures contains different number of data ALUs

	1 ALU, 2 acc, 4 reg, 2 mem						2 ALU, 2 acc, 4 reg, 2 mem					
	RSVR		RSF		RST		RSVR		RSF		RST	
	len	#	*len*	#	*len*	#	*len*	#	*len*	#	*len*	#
Wave Digital Filter	6	14	5	13	5.5	13	6	14	5	13	5.5	13
Filter	6	11	5.5	10.5	5	10.5	6	11	5.5	10.5	**4.5**	10.5
IIR 2D	16	37	16	37	16	37	**12**	39*	**13**	40*	**12.5**	40*
Forward-substitution	5	11	5.5	10.5	5	10.5	**4**	11	**4**	10.5	**4.5**	10.5
THCS	4	10	4	9.5	4	10	**3**	10	**3**	9.5	**3**	10
DFT	13	32	12.5	30	13	31.5	**12**	34*	**11.5**	32*	13	35*
Floyd-Steinberg	18	39	17.5	39.5	17	39.5	**15**	41*	**16.5**	45.5*	17.5	45.5*
Transmission Line	12	29	12	29	12	29	**8**	**28**	**11**	29.5*	**11**	29.5*
IIR 1D	8	18	8	18	--	--	**6**	18	**7**	18.5*	--	--
Equation Solver	13	26	11.5	25.5	--	--	**10**	**25**	11.5	26.5*	--	--
All-pole Lattice Filter	17	35	16	34.5	--	--	**16**	**33**	**14**	35.5*	--	--
Elliptic Filter	36	75	34	72	--	--	**27**	80*	**29**	80*	--	--

in input registers instead of memory, additional register transfer instructions are still inserted. Therefore, if we only increase the number of input registers, spill codes cannot be efficiently reduced.

In Table 7 we show result of different number of data ALUs. In this case we do not add more accumulators with the number of data ALU increasing, which means both architectures contain only two accumulators. That is, when the target architecture has two data ALUs, an accumulator is specifically allocated to a data ALU to store all destination operands calculated from it. From Table 7, schedule lengths are obviously shortened because the second data ALU is beneficial to explore instruction-level parallelism. However, instruction counts increase in some MDFGs as

Table 8
Experimental results, with target architectures contains different number of data ALUs

	1 ALU, 4 acc, 4 reg, 2 mem						2 ALU, 4 acc, 4 reg, 2 mem					
	RSVR		RSF		RST		RSVR		RSF		RST	
	len	#	*len*	#	*len*	#	*len*	#	*len*	#	*len*	#
Wave Digital Filter	6	14	5	13	5.5	13	6	14	**4.5**	13	5.5	13
Filter	6	11	5.5	10.5	4.5	10.5	6	11	5.5	10.5	4.5	10.5
IIR 2D	16	34	16	34	16	34	**10**	34	**10**	34	**9**	34
Forward-substitution	5	11	5.5	10.5	5	10.5	**4**	11	**4**	10.5	**4**	10.5
THCS	4	10	4	9.5	4	10	**3**	10	**3**	9.5	**3**	10
DFT	12	32	12	28.5	12	30	**10**	32	**9**	**28**	**9.5**	30
Floyd-Steinberg	18	38	17	37.5	17	37.5	**13**	38	**14**	39.5	**13.5**	39.5
Transmission Line	12	26	12	26	12	26	**8**	26	**8.5**	26	**8.5**	26
IIR 1D	8	17	8	17	--	--	**6**	17	**6**	17	--	--
Equation Solver	13	26	11.5	25	--	--	**10**	**25**	**10**	25.5	--	--
All-pole Lattice Filter	17	33	16	30.5	--	--	**16**	33	**13**	30.5	--	--
Elliptic Filter	35	67	30.5	63.5	--	--	**23**	70	**24**	68	--	--

asterisked in the table, which represents more spill codes are inserted. Apparently these additional spill codes are mainly incurred from frequently occurred accumulator spills, since only one dedicated accumulator is capable to store destination operands calculated from a data ALU. If an ALU result will be used later than next ALU instruction been executed, it must be temporarily stored into input registers or memory to avoid being overwritten. Therefore, from these results we conclude that increasing the number of data ALUs only is not an appropriate way to explore instruction-level parallelism.

Similarly, Table 8 still shows results of different number of data ALUs. This time we increase the number of accumulators to four, and evenly allocated them when the architecture contains two data ALUs. Compared to Table 7, clearly that not only schedule lengths are further shortened, but also spill codes are inserted infrequently since many values in Table 8 are shaded. These results are essentially the combination of results shown in Tables 5 and 7. Using more data ALUs is beneficial to shorten schedule lengths, and adding additional accumulators can reduce occurrences of spill codes efficiently. Therefore, if we want to explore instruction-level parallelism, both numbers of data ALUs and accumulators must be increased.

Finally, in Table 9 we show results of increasing the number of memory banks. Both architectures consist of six input registers, and they will be evenly allocated to each memory bank. From these results, we find that schedule lengths are hardly improved without additional data ALUs. As for the instruction count, using more memory banks seems helpful due to many bold and shaded values in this table. The reason is that when the target architecture contains N memory banks, up to N move operations can be executed simultaneously. With the number of memory banks increasing, more independent memory accesses, as well as register transfers inserted to resolve accumulator spills, can be executed in one cycle. In this situation, instruction-level parallelism among move operations will be explored, which is beneficial to reduce occurrences of register spills. However, implementing additional memory banks, associated with dedicated data buses, definitely requires heavy hardware costs. Besides, recall that before using variable partitioning mechanisms proposed in RSF or RST, the TDAG is unfolded or tiled with factor equal to the number of memory banks. A larger TDAG also costs longer time doing code generation. Therefore, we do not recommend using more memory banks to reduce the instruction count, because the cost-performance is not worth.

4.3. Brief summaries

After showing our RSSA is quite effective compared to previous work under the Motorola DSP56000 architecture, we present its effectiveness on both two evaluation metrics in some detail as follows. As shown in Tables 7 and 8, using RSSA can get schedule lengths no longer than their critical paths in most selected MDFGs when the target architecture consists of two data ALUs. These results represent lower bounds of schedule lengths are usually achieved. Besides, if there is only one data ALU, it still can almost obtain schedule lengths equal to the number of ALU instructions in the corresponding TDAGs. This situation

Table 9
Experimental results, with target architectures contains different number of memory banks

	1 ALU, 3 acc, 6 reg, 2 mem						1 ALU, 3 acc, 6 reg, 3 mem					
	RSVR		RSF		RST		RSVR		RSF		RST	
	len	#	*len*	#	*len*	#	*len*	#	*len*	#	*len*	#
Wave Digital Filter	6	14	5	13	5	13	**4**	14	**4**	**12.6**	5	13.3
Filter	6	11	5.5	10.5	4.5	10.5	6	11	**5.3**	**10.3**	**4**	**10.3**
IIR 2D	16	34	16	34	16	34	16	34	16	34	16	34
Forward-substitution	5	11	5	10.5	5	10.5	5	11	5	**10.3**	5	**10.3**
THCS	4	10	4	9.5	4	10	4	10	4	**9.3**	4	10
DFT	12	32	12	28	12	30	12	32	12	**26.6**	12	30
Floyd-Steinberg	18	38	17	38.5	17	38.5	18	38	17	**38.3**	17	**38.3**
Transmission Line	12	27	12	27	12	27	12	27	12	27	12	27
IIR 1D	8	17	8	17	--	--	8	17	8	**16.6**	--	--
Equation Solver	13	25	11.5	22.5	--	--	**12**	25	**11.3**	**21.6**	--	--
All-pole Lattice Filter	17	33	16	31.5	--	--	17	33	**15.6**	**30**	--	--
Elliptic Filter	35	68	30.5	65	--	--	35	68	35.3	**64**	--	--

indicates that these scheduling results cannot be shortened further, because all ALU instructions must be executed on the single data ALU. On the other hand, according to Tables 4–9, the proposed method really can generate quite few spill codes especially when the architecture has more than four accumulators. This is because we give an overwritten ALU result higher priority to be temporarily stored in a register, and insert spill codes only when required. In addition, we compact spill codes with other regular codes as far as possible during the scheduling process to prevent lengthening the final schedule length. Whereas on both evaluation metrics the proposed RSSA achieves usually optimal results, we conclude that it is quite effective.

Then, we summarize the influence of differing number of resources on the scheduling result. According to descriptions in Section 4.2, we find that accumulator is the most critical resource in non-orthogonal DSP architecture. Adding more accumulators can keep more ALU results for further using, which is the most efficient way to reduce spill codes. From our evaluation results, almost all spill codes can be eliminated if the target architecture contains more than four accumulators. Other than accumulator, increasing the number of input registers or memory banks is also useful for reducing the instruction count, but its improvement is not as obvious as using more accumulators. Besides, implementing additional memory banks and associated data buses needs heavy hardware costs. Thus, due to their unworthy cost-performance, we think a target architecture that consists of two memory banks and four input registers, two for each memory bank, is appropriate. As for exploring instruction-level parallelism, adding additional data ALUs is certainly necessary. Based on evaluation results shown in Table 8, we find that two data ALUs are actually sufficient, because RSSA generates the shortest schedules in most MDFGs. Using more than two data ALUs no doubt can further shorten obtained scheduling results, but the improvement will be clearly slight. Meanwhile, in additional to data ALU, the number of accumulators has to be increased concurrently for instruction-level parallelism exploration. Many spill codes may be incurred if adding more data ALUs only, where the reason we have already listed in previous subsection. Furthermore, with the generality of our hypothetical machine model and proposed method, they can be used to determine how many resources are required for an application to achieve its optimal schedule. From the detailed scheduling results, we also find that the variable partitioning mechanism proposed in RSF is unsuitable for one-dimensional MDFGs. This is because loop-carried data dependences in one-dimensional MDFGs are usually with distance one, and most memory accesses will reference variables from the same memory bank after applying loop unfolding. Thus, a memory access may easily fail to be scheduled successfully in time, which will lengthen the schedule.

In the following, we evaluate the efficiency of RSSA and compare to our previous study (Lee and Chen, 2005). Recall that both methods mainly contain following phases: constructing the MDFG, partitioning variables, constructing the TDAG, constructing G_{op}, scheduling instructions (I), resolving accumulator spills, scheduling instructions (II), and retiming the initial schedule. Among these phases, resolving accumulator spills is the most time-consuming one. In Lee and Chen (2005), we design a relatively complicated mechanism to predict accumulator spills by analyzing the topology of given TDAG. In RSSA, this phase is completed based on variables listed

in Table 1, which are constantly maintained during entire code generation process. Apparently, mechanism used in RSSA is more efficient. Moreover, it is also more general, accurate, and will not generate possibly unnecessary spill codes. Except how to resolve accumulator spills, other phases of these two methods are very similar in essence. Therefore, we conclude that RSSA is efficient than Lee and Chen (2005).

Compared with two related studies (Cho et al., 2002; Sudarsanam and Malik, 2000), our RSSA still has advantages. Because Cho et al. (2002) and Sudarsanam and Malik (2000) do not contain procedures to generate uncompacted codes, we also omit complexities of constructing the MDFG and the TDAG. In addition, all three methods use *list scheduling* to schedule instructions, so we focus on discussing their complexities in partitioning variables, allocating accumulators/registers, and resolving accumulator/register spills. In Sudarsanam and Malik (2000), it uses *graph labeling* to assign variables and accumulators/registers simultaneously. After that, it requires a mechanism to determine and resolve accumulator/register spills, although this mechanism is not presented in detail. However, despite which mechanism is used to insert spill codes, this method suggests applying *simulated annealing* to solve the *graph labeling* problem. Because *simulated annealing* is a well-known time-consuming algorithm, it makes the entire method much more complicated. Next, in Cho et al. (2002), it uses *graph coloring* to partition variables and allocate accumulators/registers separately. Similarly, it does not present the mechanism to determine and resolve accumulator/register spills, but it is definitely required. This method itself also contains a heuristic to solve the *graph coloring* problem. Therefore, no matter how it inserts spill codes, the method proposed in Cho et al. (2002) is clearly efficient than that in Sudarsanam and Malik (2000). Finally, we consider the complexity of our RSSA. Three variable partitioning mechanisms we applied are all very simple, especially the two proposed in Lee and Chen (2004) that assign variables directly based on array indices. Moreover, we insert spill codes according to variables defined in Table 1, which is clearly quite efficient. After resolving accumulator/register spills, the physical assignment of accumulators/registers becomes trivial. Therefore, the proposed RSSA is not only effective but also efficient.

5. Conclusions and future work

In this paper, we propose a code generation algorithm *Rotation Scheduling with Spill Codes Avoiding* (*RSSA*) to schedule uniform loops on non-orthogonal DSP architecture. It is extended from our previous study, and its scheduling goal is to achieve shorter schedule length and avoid generating spill codes as far as possible. RSSA mainly contains following features: generating uncompacted codes directly from a high-level language, performing variable partition before code compaction, separately scheduling ALU and memory load instructions, and applying the retiming technique to explore potential parallelism between iterations. According to evaluation results, RSSA actually achieves its scheduling goal under the Motorola DSP56000 architecture. In addition, we also define a hypothetical machine model to represent a scalable non-orthogonal DSP architecture. After evaluating RSSA on our hypothetical machine models, we can study the influence of different number of resources on the scheduling result. Evaluation results show that using more accumulators can efficiently reduce spill codes. Increasing the number of input registers or memory banks has similar improvement, but is not as obvious as adding more accumulators. As for exploring instruction-level parallelism, both numbers of data ALUs and accumulators must be concurrently increased. Adding more data ALUs only will incur many unwished spill codes, which is not an appropriate way to shorten the schedule length. Furthermore, compared to related work, our RSSA is not only quite effective on both evaluation metrics, but also an efficient algorithm.

Apart from the features described above there remain several promising issues for future research. In Lee and Chen (2005) we list two future research issues including designing a general algorithm and considering memory offset assignment. The former is completed in this paper and we will continue studying the later. After including two phases of memory offset assignment and address register allocation, our code generation algorithm will become more complete. In addition, in addition to high data throughput, low power consumption is also a significant factor in DSP architecture. In today's systems, software constitutes a major component and its role is projected to grow even further. Therefore, there is a clear need for considering the power consumption from the point of view of software. Software affects the system power consumption at various levels of the design. Among these, power analysis at the instruction level is the most interesting, because it is sufficiently accurate and will not cost too much time to analyze. Therefore, in the near future, we will study instruction level power models and try to design new code generation algorithms that optimize both schedule length and power consumption as well.

References

Cho, J., Paek, Y., Whalley, D., 2002. Efficient register and memory assignment for non-orthogonal architectures via graph coloring and MST algorithms. In: Proceedings of ACM Joint conference LCTES-SCOPES, June, pp. 130–138.

Daveau, J.M., Thery, T., Lepley, T., Santana, M., 2004. A retargetable register allocation framework for embedded processors. In: Proceedings of ACM SIGPLAN/SIGBED, June, pp. 202–210.

Eyre, J., Bier, J., 2000. The evolution of DSP processors. IEEE Signal Processing Magazine 17 (2), 43–51.

Hsu, Y.C., Jeang, Y.L., 1993. Pipeline Scheduling Techniques in High-Level Synthesis. In: Proceedings of 6th Annual IEEE International ASIC Conference and Exhibition, Rochester, pp. 396–403.

Kessler, C., Bednarski, A., 2002. Optimal integrated code generation for clustered VLIW architectures. In: Proceedings of ACM Joint Conference LCTES-SCOPES, June, pp. 102–111.

Kung, S.Y., 1988. VLSI Array Processors. Prentice-Hall, Englewood, NJ.
Lamport, L., 1974. The parallel execution of DO loops. Communications of the ACM SIGPLAN 17 (2), 82–93.
Lapsley, P., Bier, J., Shoham, A., Lee, E.A., 1996. DSP Processor Fundamentals: Architectures and Features. Berkeley Design Technology, Inc.
Lee, Y.H., Chen, C., 2004. Efficient variable partitioning and scheduling methods of multiple memory modules for DSP. In: Proceedings of 10th Workshop on Compiler Techniques for High-Performance Computing, March, pp. 80–89.
Lee, Y.H., Chen, C., 2005. An efficient code generation algorithm for non-orthogonal DSP architecture. In: Proceedings of 11th Workshop on Compiler Techniques for High-Performance Computing, March, pp. 49–58.
Leiserson, C.E., Saxe, J.B., 1991. Retiming synchronous circuitry. Algorithmica 6 (1), 5–35.
Leupers, R., Kotte, D., 2001. Variable partitioning for dual memory bank DSPs. In: Proceedings of International Conference on Acoustics, Speech, and Signal Processing, vol. 2, pp. 1121–1124.
Madisetti, V.K., 1995. VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis. Butterworth-Heinemann, Boston.
MESCAL, Availabel from: <http://www.princeton.edu/~mescal/>, <http://embedded.eecs.berkeley.edu/mescal/>.
Motorola, DSP56000/DSP56001 Digital Signal Processor User's Manual, Motorola Inc., Phoenix, AZ.
OptimoDE, Available from: <http://www.arm.com/products/CPUs/families/OptimoDE.html>.
ORC, Available from: <http://ipf-orc.sourceforge.net/>.
Saghir, M.A.R., Chow, P., Lee, C.G., 1994. Towards better DSP architectures and compilers. In: Proceedings of International Conference on Signal Processing Applications and Technology, October, pp. 658–664.
Saghir, M.A.R., Chow, P., Lee, C.G., 1996. Exploiting dual-memory banks in digital signal processors. In: Proceedings of 7th International Conference on Architecture Support for Programming Language and Operating Systems, pp. 234–243.
Scholz, B., Eckstein, E., 2002. Register allocation for irregular architectures. In: Proceedings of ACM Joint Conference LCTES-SCOPES, June, pp. 139–148.
Shiue, W.T., 2001. Energy-efficient backend compiler design for embedded systems. In: Proceedings of 10th International Conference on Electrical and Electronic Technology, vol. 1, August, pp. 103–109.
Sudarsanam, A., Malik, A., 2000. Simultaneous reference allocation in code generation for dual data memory bank ASIPS. ACM Transactions on Design Automation of Electronic Systems 5 (2), 242–264.
Tensilica, Available from: <http://www.tensilica.com/>.
Trimaran, Available from: <http://www.trimaran.org/>.
Wang, Z., Hu, X.S., 2004. Power aware variable partitioning and instruction scheduling for multiple memory banks. In: Proceedings of Design, Automation and Test in Europe Conference and Exhibition, vol. 1, February, pp. 312–317.
Zhuang, X., Zhang, T., Pande, S., 2004. Hardware-managed register allocation for embedded processors. In: Proceedings of ACM SIGPLAN/SIGBED, June, pp. 192–201.
Zhuge, Q., Xiao, B., Sha, E.H.M., 2001. Exploring variable partitioning for dual data-memory bank processors. In: Proceedings of 34th International Symposium on Microarchitecture, December, pp. 45–52.

Cheng Chen is a professor in the Department of Computer Science and Information Engineering at National Chiao Tung University, Taiwan, ROC. He received his B.S. degree from the Tatung Institute of Technology, Taiwan, ROC in 1969 and M.S. degree from the National Chiao Tung University, Taiwan, ROC in 1971, both in electrical engineering. Since 1972, he has been on the faculty of National Chiao Tung University, Taiwan, ROC. From 1980 to 1987, he was a visiting scholar at the University of Illinois at Urbana Champaign. During 1987 and 1988, he served as the chairman of the Department of Computer Science and Information Engineering at the National Chiao Tung University. From 1988 to 1989, he was a visiting scholar of the Carnegie Mellon University (CMU). Between 1990 and 1994, he served as the deputy director of the Microelectronics and Information Systems Research Chnter (MISC) in National Chiao Tung University. His current research interests include computer architecture, parallel processing system design, parallelizing compiler techniques, and high performance video server design.

Yi-Hsuan Lee is a Ph.D. candidate in Computer Science and Information Engineering at National Chiao Tung University, Taiwan, ROC. She received her B.S. degree in Computer Science and Information Engineering at National Chiao Tung University, Taiwan, ROC in 1999. Her current research interests include computer architecture, parallelizing compiler techniques, task scheduling for heterogeneous systems, scheduling problem in DSP architecture, and low-power scheduling techniques in embedded system.