Journal of VLSI Signal Processing 47, 281–296, 2007

DOI: 10.1007/s11265-007-0053-x

An Efficient Code Generation Algorithm for Non-orthogonal DSP Architecture

YI-HSUAN LEE AND CHENG CHEN
Department of Computer Science and Information Engineering, National Chiao-Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 300, People's Republic of China

Received: 19 September 2006; Accepted: 29 January 2007

Abstract. To meet strict speed and power requirements for embedded applications, many high-end digital Signal Processors (DSPs) commonly employ *non-orthogonal* architectures that are typically characterized by irregular data paths, heterogeneous registers, and multiple memory banks. Obviously to harvest the benefits provided by this non-orthogonal architecture sufficient compiler support is necessary and important. However, the complexity of such architectures presents a great challenge to compiler design and the usual compilation techniques for general-purpose CPUs do not adapt well to the irregularity of DSP. The entire code generation process must include the following phases: intermediate representation, code compaction, instruction scheduling, memory bank assignment (or variable partition), and register/accumulator assignment. Much related research only considers some phases, which is inadequate. In this paper, we present an effective code generation algorithm named *Rotation Scheduling with Spill Codes Predicting* (*RSSP*) to maximally exploit the benefits of non-orthogonal architectures. It contains six parts that cover almost the entire phases of the code generation process. As well as introducing the detailed principles and algorithms of the proposed RSSP, we use an analytic model to evaluate its preliminary performance. Evaluation results clearly demonstrate the effectiveness of the proposed method. Furthermore, we also present some preliminary ideas to generalize RSSP, which can make it more practicable and suit various DSPs with similar architectural features.

Keywords: DSP, non-orthogonal architecture, code generation

1. Introduction

To meet ever-increasing demands for higher performance a *Digital Signal Processors* (*DSPs*) with sophisticated architecture are being designed and produced to better match the target applications [1–5]. Such architecture typically has a *non-orthogonal* architecture, which can be characterized by irregular data paths containing heterogeneous register sets and multiple memory banks [6]. In the data path, this architecture lacks a large number of centralized general-purpose homogeneous registers. Instead, it has multiple small register sets where different sets are dedicated to different functional units. In addition, this architecture employs multiple memory banks connected through independent data buses. Therefore, variables in programs can be partitioned into separate banks and accessed simultaneously. A number of embedded DSPs, such as Analog Device ADSP2100, Motorola DSP56000, and NEC uPD77016, are based on this architecture.

Although parallel access, which is enabled by multi-bank memory, is useful to explore the potential of higher memory bandwidth, it gives rise to the problem of how to partition the variables into the multiple memory banks [1, 5–16]. Similarly, using

heterogeneous register sets can decrease the architectural complexity but increases the difficulty of deciding which register set to use for a certain instruction [6, 7]. Obviously, to harvest the benefits provided by this non-orthogonal architecture adequate compiler support is necessary and important [3, 4]. However, it is well known that compilation techniques for general-purpose CPU do not adapt well to the irregularities of DSP. Therefore, many researches seek to design code generation methods for specific DSP architectures to fully use their features.

The complete code generation process must include many phases, such as intermediate representation, code compaction, instruction scheduling, memory bank assignment (or variable partition), and register/accumulator assignment [7]. We previously proposed two scheduling methods for multi-bank memory architecture that cover all phases except register/accumulator assignment [11]. Therefore, in this paper, we present a new method named *Rotation Scheduling with Spill Codes Predicting* (*RSSP*) to consider this phase and further improve overall execution performance. Unlike previous similar work, we predict the occurrence of register and accumulator spills and generate corresponding spill codes before code compaction. This enables these spill codes to be scheduled in parallel with the other operations, which can prevent extension of the schedule length. As well as introducing detailed algorithms for RSSP, we use an analytic model and some DSP applications to evaluate its preliminary computational performance. According to these evaluation results RSSP clearly outperforms other related methods, including our previous studies. Furthermore, because RSSP is designed only for specific target architecture, its practicability is really limited. Hence, we also present some preliminary ideas to generalize RSSP, which suits for various DSPs with similar architectural features.

The remainder of this paper is organized as follows. Section 2 surveys the fundamental background and related work. An overview of the Motorola DSP56000 architecture and our motivations are also presented in this section. Design principles, detailed algorithms, and preliminary generations of proposed RSSP are introduced in Section 3. Section 4 contains evaluation results from experiments with an analytic model. Finally, our conclusions and plans for future work are presented in Section 5.

2. Fundamental Background

In this section, we first model the given program and briefly survey some fundamentals. Then, an overview of the Motorola DSP56000 architecture and related work are presented. Finally, we briefly summarize previous studies and describe the motivation for designing the new method.

2.1. *Modeling the Given Program*

Because multimedia and DSP applications usually contain repetitive groups of operations, they can be easily represented by uniform nested loops. A *Multi-dimensional Data Flow Graph* (*MDFG*) is commonly used to model uniform nested loops. We define a MDFG is slightly differently from previous studies [5, 11] as follows.

Definition 2.1 A Multi-dimensional Data Flow Graph (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 over nodes in V; $X(e)$ 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; $P(v)$ represents the type of node v (see Fig. 1c).

Nodes in the data flow graph include both arithmetic logic unit (ALU) operations (multiplications and additions) and other operations (memory accesses and register transfers). An edge label $X(e)$ indicates a variable that is accessed by a memory access through edge e. Note that edges that do not involve a memory access do not have a label. Figure 1 shows an example of nested loop and its corresponding MDFG.

A MDFG is *realizable* if there exists a *schedule vector* s such that $s \cdot d \geq 0$, where d are its loop-carried dependencies [17]. An *iteration* is equivalent to the execution of each node in V exactly once. The period during which all nodes in an iteration are executed, according to data dependencies and without resource constraints, is called a *cycle period*. Clearly, the cycle period will dominate the execution time of a nested loop. Note that many MDFGs can represent a single

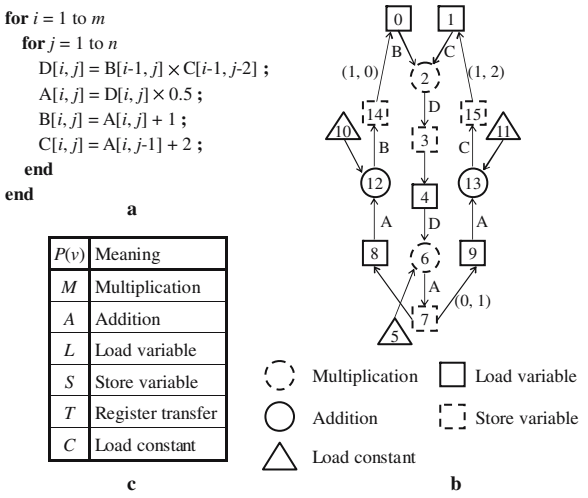


Figure 1. **a** Nested loop, **b** corresponding MDFG, **c** node types.

DSP application, depending on its representation by nested loops.

2.2. *Retiming*

Retiming [18] is a popular technique used in loop scheduling that redistributes nodes in consecutive iterations to enhance the execution performance. A multi-dimensional retiming vector r is a function from V to Z^n that redistributes nodes in consecutive iterations, where n is the number of dimensions. The retiming vector $r(v)$ represents the offset between the original iteration and that after retiming. A new MDFG $G_r = (V, E, X, d_r, P)$ is created after applying r such that each iteration still has one execution of each node. The difference between G and G_r is only the delay vectors, which change to preserve the original dependencies.

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

2.3. *Motorola DSP56000 Architecture*

Our target architecture consists of multiple memory banks and a heterogeneous register set. Associated with each memory bank is an independent set of address bus, data bus, and address generation unit (AGU). The Motorola DSP56000 [19] is an example of such an architecture with two memory banks, therefore used it in our method design and experiments.

As shown in Fig. 2, the DSP56000 architectural units of interest are the Data ALU, Address General Unit (AGU),and X/Y memory banks. The Data ALU consists of four input registers called *X0*, *X1*, *Y0*, and *Y1*, and two accumulators, *A* and *B*. 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. Two buses *XDB* and *YDB* permit two input registers or accumulators to be read or written in conjunction during execution of an ALU operation. Therefore, up to two *move* operations (including memory access, register transfer, and immediate load) and one Data ALU operation may be executed simultaneously in one cycle.

Two independent move operations executed in the same cycle are called *parallel moves*. However, due to the nature of the DSP56000 architecture, not all pairs of move operations can be performed in parallel. Detailed *parallel move conditions* can be found in [6, 7, 19]. In this paper we especially consider the following conditions: (1) the two move operations reference data in different memory banks; (2) the two destination registers are different; (3) the X memory access loads into restricted locations X0, X1, A, or B; and (4) the Y memory access loads into restricted locations Y0, Y1, A, or B.

2.4. *Related Work*

In the architecture with multiple memory banks and a heterogeneous register set, a complete code generation algorithm must include five phases: intermediate representation, code compaction, instruction scheduling, memory bank assignment (or variable partition), and register/accumulator assignment [7]. These five phases can be performed in various sequences. Moreover, because these phases are extremely dependent, some code generation algorithms are designed by simultaneously performing more than one phase. However, these algorithms with tightly coupled phases are very time consuming,

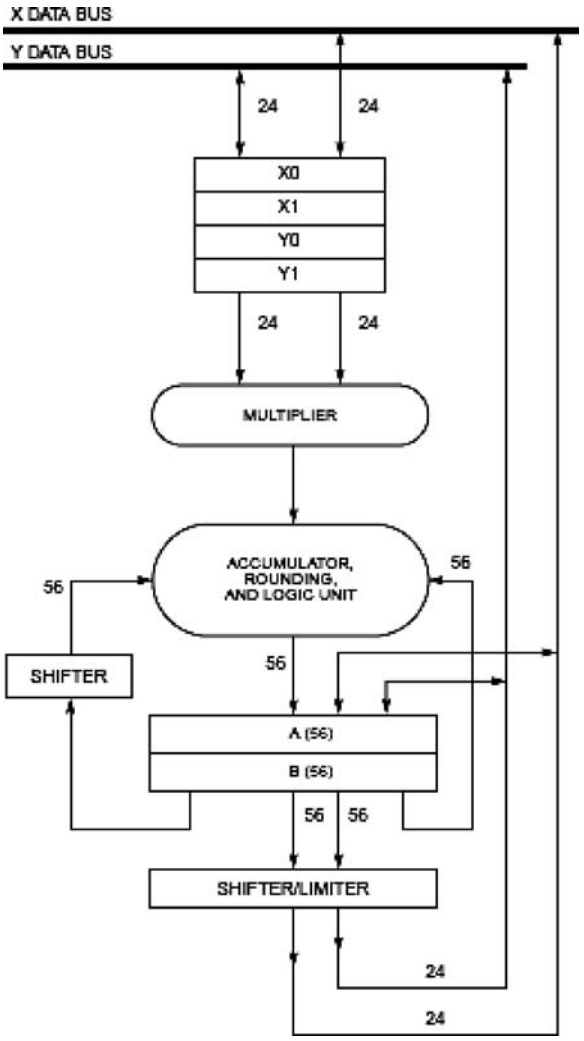


Figure 2. The datapath of Motorola DSP56000 architecture.

in this paper we do not design our method using this mechanism.

A number of papers have investigated the use of multi-bank memory to achieve maximum instruction level parallelism [1, 5–16]. Among these previous studies, only two methods in [6–9] contain all five phases. Methods in [1, 5, 10, 11] contain all phases except for register/accumulator assignment, and others in [12, 13] are simply variable partitioning mechanisms. For heterogeneous register sets, [14–16] present specific register allocation algorithms to fit their irregularity. In addition, because nested loops are the time-critical sections in DSP applications, their execution time will dominate the entire computational performance. However, in most previous works code is generated per basic block, which cannot explore the embedded parallelism of between different iterations [1, 6–10, 12, 13].

2.5. *Motivations*

After surveying previous methods, we briefly summarize them and introduce our motivation for designing the new method. First, memory bank assignment can be performed before or after code compaction. In the architecture with multiple memory banks, memory accesses involved in a parallel move must reference variables in different banks [6, 7, 19]. If variables are partitioned after code compaction, such as the algorithms proposed in [6, 7], memory accesses are scheduled without information of memory bank assignment. Therefore, two memory accesses may be assumed to be executed in parallel but in fact, they cannot be. In this situation, an extra cycle (*spill code*) would be required to access them. If spill codes occur frequently, clearly the computational performance is degraded. On the other hand, if variables are partitioned before code compaction, spill codes will not occur. In our method, we use the later method to avoid the occurrence of spill codes.

Apart from location conflict for parallel moves, spill codes can possibly also occur in the register/accumulator assignment phase. All previous code generation algorithms that consider register/accumulator assignment perform this phase last [6, 7]. That is, during the code compaction phase, variables are stored in unlimited *symbolic* registers/accumulators. However, the capacities of the registers and accumulators are strictly limited in DSP. Therefore, spill codes are required when register and accumulator spills occur, and their *spill costs* may be more than one extra cycle. In our method, we propose mechanisms to predict the occurrence of register and accumulator spills and generate corresponding spill codes before code compaction. Then, in the code compaction phase, these spill codes can be scheduled in parallel with other operations, this can decrease the spill costs.

3. Rotation Scheduling with Spill Codes Predicting (RSSP)

In this section, we introduce our proposed method named *Rotation Scheduling with Spill Codes Predicting* (*RSSP*). Section 3.1 contains some preliminaries concerning our assumptions and scheduling

principles. Detailed scheduling steps of RSSP are introduced in Section 3.2 and its subsections. Note that although our RSSP is designed and experimentally tested on the Motorola DSP56000 architecture it can be extended to an architecture with a homogeneous register set or more memory banks. In Section 3.3 we list preliminary ideas to generalize RSSP, in order to deal with various DSPs with similar architectural features.

3.1. Preliminaries

To correctly execute a MDFG, its data dependencies cannot be violated. Under the restricted constraint of a heterogeneous register set, registers and accumulators should be used very carefully to preserve data dependencies. Corresponding to the nature of DSP56000 architecture, we list conditions for a correct schedule as follows:

1. For an operand op_i loaded from memory to reg_i, reg_i cannot be loaded with another operand op_j before op_i is read.
2. For an operand op_i transferred from accumulator acc_i to reg_i, reg_i cannot be loaded with another operand op_j before op_i is read.
3. For an ALU result rt_i stored in accumulator acc_i, acc_i cannot store another ALU result rt_j before rt_i is stored back in memory.
4. For an ALU result rt_i stored in accumulator acc_i, acc_i cannot be stored with another ALU result rt_j before rt_i being transferred to register reg_i.
5. For an ALU result rt_i stored in accumulator acc_i, acc_i cannot store another ALU result rt_j before rt_i is read as an operand.

From the above conditions and the limited quantity of registers/accumulators, we conclude five scheduling principles that a correct schedule must satisfy. Assume that the DSP architecture consists of $2m$ input registers and n accumulators. Furthermore, for convenience, we only permit a variable loaded from memory to be stored in an input register.

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

These five scheduling principles and the above five conditions are one-to-one. That is, the scheduling principles are designed to satisfy the conditions and generate a correct schedule. In the next section, we propose a new scheduling method based on these principles.

In DSP applications, operands of ALU operations may be constants. Intuitively these constants can be loaded using immediate load operations but, because of the parallel move conditions, the immediate load is rarely executed in parallel with other independent move operations. Therefore, in our method we use *load constant* instead of immediate load. Assuming the constants are stored in memory at specific locations in advance, the load constant is essentially equivalent to the original load variable operation, which will load constants directly from specific address. We also assume that all constants are stored in both memory banks. Therefore, the load constant operations can be scheduled for any memory bank to increase performance.

3.2. Detailed Algorithms of RSSP

From the above related fundamentals and preliminaries, we introduce our proposed method as shown in Fig. 3. We divide the overall algorithm into six main parts: MDFG construction, TDAG construction, TDAG modification, ALU operation scheduling, other operation scheduling, and initial schedule retiming. Detailed description of each part is in the following subsections.

3.2.1. MDFG Construction. In previous research [1, 6–8], the intermediate representation phase is not included. That is, these methods need another tool to generate uncompacted symbolic intermediate code

1. G_c = Construct MDFG;
2. Partition variables to X and Y memory banks;
3. Unfold or tile G_c if necessary;
4. G_t = Construct TDAG;
5. Modify TDAG G_t;
   5.1. G_t = Insert register transfer nodes (G_t);
   5.2. (G_{op}, G_{pr}) = Construct DAG G_{op} and G_{pr} (G_t);
   5.3. G_{op} = Mark_edge (G_{op}, E_{op});
   5.4. G_{op} = Mark_edge (G_{op}, E_{pr});
   5.5. G_{op} = Check_cycle (G_{op}, G_t);
   5.6. G_t = Insert memory access nodes (G_{op}, G_t);
6. S = Schedule ALU operations (G_{op});
7. S = Schedule other operations (S, G_t);
8. S = Retime the initial scheduling result (S);

Figure 3. The overall scheduling algorithm of RSSP.

from the high-level language. On the other hand, the methods in [5, 11] use the MDFG directly generated from high-level language. In this paper, we use the same mechanism to construct the MDFG as in our previous study [11]. This mechanism directly uses memory to store temporarily variables. 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. For example, Fig. 1b shows the MDFG of the nested loop in Fig. 1a.

Except for constructing the MDFG, in the first part we also assign variables to the X and Y memory banks. Three variable partitioning mechanisms, proposed in [5, 11], can be chosen. Note that if we choose the mechanisms proposed in [11], the MDFG must be unfolded or tiled according to the number of memory banks.

3.2.2. TDAG Construction. Many operations with type L and S are contained in the MDFG, because it stores the ALU result to memory and reloads it into the register only when required for use. If we schedule operations according to this MDFG, obviously register and accumulator spills will not occur. However, this MDFG seems too *complete* to degrade the computational performance, because ALU results can be temporarily stored in registers or accumulators instead of directly written back to memory. Therefore, in the second part of proposed method, we construct a *Translated Data Acyclic Graph* (*TDAG*) defined as follows from the original MDFG; this is proposed to remove possible unnecessary memory accesses.

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 nodes in V; $X(e)$ represents the variable accessed by an edge e; $P(v)$ represents the type of node v (see Fig. 1c).

The algorithm for constructing the TDAG is shown in Fig. 4. For a given MDFG, the first step of the TDAG construction is removing edges with non-zero delays. Then, for an ALU result written back and reloaded in the same iteration, it can be temporarily stored in an accumulator to remove the corresponding operations of type L and S. However, if an ALU results will be used in latter iteration, its corresponding store variable operation must be retained. In addition, because both source operands of a multiplication must always be registers, a register transfer is added between two ALU operations. Figure 5a shows two cases of removing memory accesses, and Fig. 5b is the corresponding TDAG of the MDFG in Fig. 1b. Note that in this TDAG construction algorithm we simply assume that numbers of registers and accumulators are unlimited.

3.2.3. TDAG Modification. As described above, we wish to avoid register and accumulator spills by predicting their occurrence in advance. In the third part of RSSP description, we analyze and modify the TDAG to predict accumulator spill, and register spill will be dealt with in the fifth part.

Three main steps are required for this TDAG modification: insertion of register transfers, analysis of TDAG, and insertion of memory accesses. Because we do not consider the limited number of accumulators when constructing the TDAG an ALU operation with type M or A may have many immediate successors with type A in the TDAG. As shown in Fig. 6a, the ALU result rt_i of v_i stored in accumulator acc_i is an operand of all additions v_{j1} to v_{jm}. However, if the architecture only consists of one ALU and n accumulators, rt_i may be recovered

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;

Figure 4. The TDAG constructing algorithm.

before finishing execution of all additions v_{j1} to v_{jm} if $m > n$. To resolve this situation, rt_i must be transferred to an input register and reside in it until v_{j1} to v_{jn} have finished execution. Figure 6b contains the TDAG after inserting v_k with type T, and the algorithm to insert register transfer operations is listed in Fig. 7.

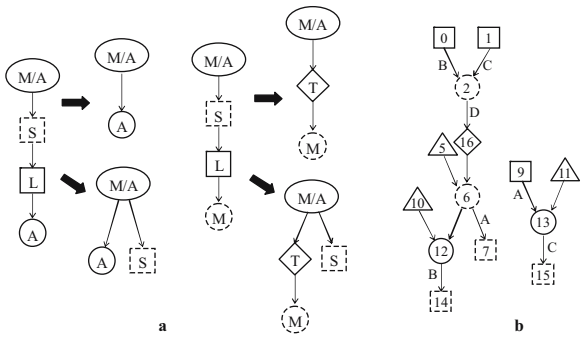


Figure 5. **a** Two cases of removing memory accesses, **b** TDAG of Fig. 1b.

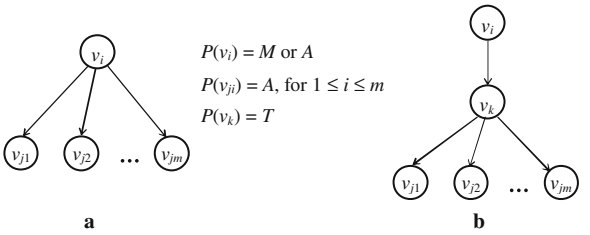


Figure 6. **a** A TDAG fragment, **b** after inserting the register transfer v_k.

Then, in the analyzing TDAG step, we analyze TDAG topologies to predict the occurrence of accumulator spill. Two intermediate DAGs G_{op} and G_{pr} defined as follows are constructed using algorithm listed in Fig. 8. Initially we set $S(e) = F$ for all edges in G_{op} and G_{pr}, which means no accumulator spill will occur. After applying algorithms listed in Figs. 9 and 10, some edges in G_{op} will be set $S(e) = T$ to indicate the occurrence of accumulator spill. *Mark_Edge* and *Check_Cycle* algorithms are proposed based on our analyses of TDAG topologies. That is, they are only designed for the architecture consisting of one ALU and two accumulators, such as the DSP56000. We leave for future research the design of general algorithms for different numbers of ALUs and accumulators. Figure 11 shows two G_{op} fragments with accumulator spill that will be checked by algorithms *Mark_Edge* and *Check_Cycle*, respectively.

Definition 3.2 A DAG $G_{op} = (V, E, S)$ is a direct graph, where V is the node set representing ALU operations; $E \subseteq V \times V$ is the edge set that defines the precedence relations over the nodes in V; $S(e)$ is an edge mark that represents two nodes that must be scheduled at separate control steps or not.

1. Input: $G = (V, E, X, P)$, n;
2. Output: $G_t = (V_t, E_t, X_t, P_t)$;
3. $G_t = G$;
4. Suppose that $v_i \in V_t$ and $P_t(v_i) = M$ or A;
5. If (v_i has more than n immediate successors $v_1,\ldots, v_n$ with type A)
   Delete edges $e_{i1},\ldots, e_{in}$ from E_t;
   Insert nodes v_x into V_t (set $P_t(v_x) = T$);
   Insert edges $e_{x1},\ldots, e_{xn}$ into E_t;
6. Return G_t;

Figure 7. The register transfer inserting algorithm.

1. Input: $G = (V, E, X, P)$;
2. Output: $G_{op} = (V_{op}, E_{op}, S)$, $G_{pr} = (V_{op}, E_{pr}, S)$;
3. $V_{op} = \{v \mid v \in V, P(v) = M \text{ or } A\}$;
4. $E_{op} = \{e_{ij} \mid e_{ij} \in E, v_i, v_j \in V_{op}\}$;
5. $E_{pr} = \{e_{ij} \mid e_{ji} \in E_{op}\}$;
6. $S(e) = \{F \mid e \in E_{op} \text{ and } E_{pr}\}$;
7. Return (G_{op}, G_{pr});

Figure 8. The G_{op} and G_{pr} constructing algorithm.

Definition 3.3 A DAG G_{op}, corresponds to an undirected DAG $G_{pr} = (V, E, S)$ with the same topology and characteristics.

Finally, the third step of the TDAG modification part is to insert memory accesses. For an edge in G_{op} with $S(e) = T$, two operations of type S and L are inserted into the TDAG at the corresponding locations. The algorithm for inserting memory accesses is listed in Fig. 12.

After applying all of proposed algorithms for TDAG modification we have introduced in this subsection in the sequence shown in Fig. 3, the modified TDAG can be scheduled without any accumulator spill.

3.2.4. ALU Operation Scheduling. In this and next parts of our proposed RSSP, all operations in the modified TDAG are scheduled considering the nature of DSP56000 architecture. In this part, we first describe scheduling rules for ALU operation scheduling.

List Scheduling is simply used to schedule ALU operations based on G_{op} (V, E, S). For an edge $e_{ij} \in E$, its edge mark $S(e_{ij})$ may be F, T, or X, which indicates different rules for scheduling v_i and v_j. Assume that $v_i \in V$ is scheduled at control step i, and the ALU result rt_i of v_i is stored in accumulator acc_i. If $S(e_{ij}) = F$ and X, v_j must be scheduled at control step $i+1$ or $i+2$ to prevent rt_i being recovered before being used. Conversely, if $S(e_{ij}) = T$, v_j can be scheduled at control step later than $i+2$, because rt_i will be transferred to register reg_i. In addition, if $S(e_{ij}) = X$ and v_j is scheduled at control step $i+1$, an idle control step is inserted between v_i and v_j for scheduling register transfer operation further. Because we have already considered the occurrence of accumulator spill, all ALU operations can be scheduled exactly according to the above three rules. Essentially, these rules are equivalent to the fifth scheduling principle listed in Section 3.1. Figure 13b shows the scheduling result

1. Input: $G = (V, E, S)$, E_t;
2. Output: $G_r = (V_r, E_r, S_r)$;
3. $G_r = G$;
4. $label(v) = N$, $\forall v \in V$;
5. $label(v) = S$, $\forall v$ doesn't have any immediate predecessor;
6. While ($\exists$ $label(v) == N$)
   6.1. $\exists e_{ij} \in E_t$, such that v_i is the only immediate predecessor of v_j

$$label(v_j) = \begin{cases} V & \text{if } label(v_i) == S \\ label(v_i) & \text{otherwise} \end{cases}$$

   6.2. $\exists e_{ik}, e_{jk} \in E_t$
   If ($label(v_i) == N$ or $label(v_j) == N$) $label(v_k) = N$;
   else if ($label(v_i) == S$ or $label(v_j) == S$) $label(v_k) = H$;
   else if ($label(v_i) == V$ and $label(v_j) == V$) $label(v_k) = H$;
   else $label(v_k) = G$; $S_r(e_{jk}) = T$;
7. Return G_r;

Figure 9. The *Mark_Edge* algorithm.

1. Input: $G = (V, E, S)$, $G_t = (V_t, E_t, X_t, P_t)$;
2. Output: $G_r = (V_r, E_r, S_r)$;
3. $G_r = G$;
4. Delete edge e from E, such that $S(e) = T$;
5. $\forall e_{ij} \in E_t$ such that $P_t(v_j) = T$
   $\forall e_{jk} \in E_t$, insert edge e_{ik} into E (set $S(e_{ik}) = X$);
6. Remove edge direction in G;
7. Level each node $v \in V$ ($level(v)$ indicates the longest path length from v to any root node; $level(v) = 1$ if v is a root node)
8. If ($\exists$ a cycle $v_i \rightarrow v_{i+1} \rightarrow \ldots \rightarrow v_k \rightarrow v_{k+1} \rightarrow \ldots \rightarrow v_j \rightarrow v_i$ in G)
   8.1. Suppose v_j has the smallest $level(v)$ value in this path;
   8.2. If (($level(v_j) < level(v_{i+1})$ in path $v_i \rightarrow \ldots \rightarrow v_k$) and ($level(v_k) < level(v_{k+1})$ in path $v_k \rightarrow \ldots \rightarrow v_i$)) $S_r(e_{ij}) = T$;
   else $S_r(e_{ij}) = T$, $\forall$ $level(v) = level(v_i)$ in this path;
9. Return G_r;

Figure 10. The *Check_Cycle* algorithm.

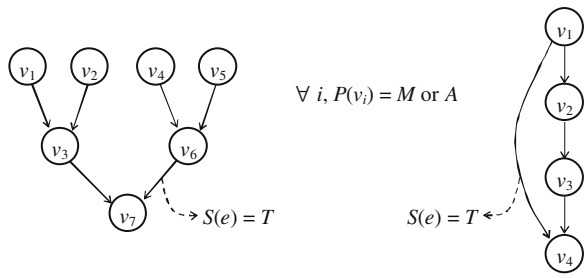


Figure 11. Two G_{op} fragments with accumulator spill.

1. Input: $G = (V, E, S)$, $G_1 = (V_1, E_1, X_1, P_1)$;
2. Output: $G_t = (V_t, E_t, X_t, P_t)$;
3. $G_t = G_1$;
4. $\forall e_{ij} \in E$ such that $S(e_{ij}) = = T$
   Delete edge e_{ij} from E_t;
   Insert nodes v_s, v_l into V_t (set $P_t(v_s) = S$, $P_t(v_l) = L$);
   Insert edges e_{is}, e_{sl}, e_{lj} into E_t (set $X_t(e_{is}) = t$, $X_t(e_{lj}) = t$, where t is a temporary variable);
5. Return G_t;

Figure 12. The memory access inserting algorithm.

of the ALU operations only for the TDAG shown in Fig. 13a.

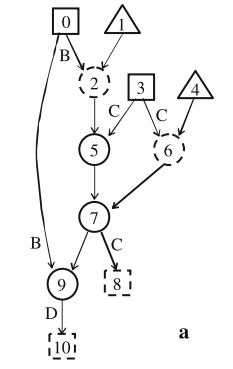
3.2.5. Other Operation Scheduling. After scheduling ALU operations, other operations, including memory accesses and register transfers, are scheduled based on the modified TDAG G_t. Meanwhile, we consider the limited number of registers during scheduling, therefore no extra action is required to check and deal with the occurrences of register spill. Two variables, *reg_x(t)* and *reg_y(t),* are used to record the number of registers used at control step t for X and Y memory banks, respectively. These two variables are dynamically maintained during this scheduling. Obviously, if we can generate a schedule where *reg_x(t)* and *reg_y(t)* do not exceed the quantity of registers for all control steps register spill will not occur. Therefore, in the fifth part of the proposed method we design appropriate rules for other operations scheduling to satisfy the conditions above.

To determine the scheduling rules we analyze the time interval that an operand must reside in a register for a correct schedule. If a variable (or constant) is loaded from memory at control step i and used at control step j, it will occupy the register from control step i to $j - 1$. Similarly, an ALU result must occupy a register from control step i to $j - 1$ if it is transferred from an accumulator at control step i and used at control step j. For scheduling rules, we conclude as follows:

1. According to the execution sequence of ALU operations, schedule their predecessors as soon as possible.
2. Scheduling principles 1∼4 listed in Section 3.1 must be satisfied, and *reg_x(t)* and *reg_y(t)* cannot exceed the number of registers for any control step.
3. If a variable is stored and loaded at consecutive control steps, the two memory accesses can be replaced by a single register transfer operation.
4. If a memory access or register transfer operation cannot be scheduled successfully, due to insufficient registers, a variable currently residing in a register should be overwritten and reloaded again when required.
5. If an overwritten variable is not used after transferring from the accumulator, the corresponding register transfer is replaced by a store variable operation.

From the scheduling rules above, all other operations can be successfully scheduled. Figure 13c shows the scheduling result of the TDAG shown in Fig. 13a. Finally, because we have alredy considered accumulator and register spills, an appropriate assignment of physical accumulators and registers exists for scheduling.

3.2.6. Initial Schedule Retiming. After generating the initial scheduling result, we use the retiming technique to explore potential parallelism among the



	ALU	X	Y	*reg_x*	*reg_y*
1		0	1	1	1
2	2	4	3	2	1
3	5			2	1
4	6			1	0
5	7			1	0
6	9		8	0	0
7		10		0	0

c

	ALU	X	Y
1	2		
2	5		
3	6		
4	7		
5	9		

b

	ALU	X	Y	*reg_x*	*reg_y*
3	5		1	2	2
4	6	0	3	2	2
5	7			2	2
6	9	4	8	2	2
7	2	10		2	1

d

Figure 13. Scheduling results, **a** TDAG, **b** ALU operations only, **c** initial schedule, **d** after retiming.

iterations. Note that the retiming technique will redistribute nodes in consecutive iterations. Hence, in loop scheduling it can be used to explore the instruction-level parallelism between different iterations, which is beneficial to fully utilize system resources and shorten the scheduling results. In our previous studies, retimed operations are rescheduled as soon as possible to reduce the schedule length. However, this time we also consider the usage of the accumulators and registers, the retimed scheduling result cannot exceed their limited numbers. Assume that the length of the initial schedule is l. In the following we present conditions so that a retimed operation can be rescheduled at control step i.

1. A retimed operation with type L or C must occupy a register from control step i to l, because this value of constant will be used for a later iteration.
2. A retimed operation with type T must occupy a register from control step i to l, because this ALU result will be used in a later iteration. In addition, the fourth scheduling principle listed in Section 3.1 has to be satisfied.
3. Rescheduling a retimed operation with type S must satisfy the third scheduling principle listed in Section 3.1.
4. Rescheduling a retimed operation with type M or A must satisfy scheduling principles 1, 2 or 5 listed in Section 3.1. In addition, *reg_x(t)* and *reg_y(t)* are updated after rescheduling this ALU operation.

In this scheduling part, we reschedule a retimed operation at the earliest control step that satisfies above conditions. Because we store constants in both memory banks in advance, a load constant operation can be rescheduled for any memory bank to achieve higher performance. Besides, in order to guarantee the assignment of physical accumulators and registers still exists for the retimed scheduling result, the limited number of registers cannot be exceeded during rescheduling. Figure 13d shows the retimed scheduling result of Fig. 13c.

3.3. The Preliminary Generalization of RSSP

In above subsection we have introduced proposed RSSP in some detail. We expect it is an effective method, and also efficient enough. However, because algorithms *Mark_Edge* and *Check_Cycles* used to predict accumulator spills only designed for target architecture with one Data ALU and two accumulators, the practicability of RSSP becomes much limited. Therefore, in this subsection, we discuss the preliminary generalization of RSSP.

About the general version of RSSP, we hope it can handle target architectures with various numbers of Data ALUs, accumulators, and registers. The proposed mechanism for avoiding register spill in RSSP can be readily extended to cover architectures with various numbers of registers. Hence, we only have to design new algorithm to predict accumulator spills. Figure 14 lists the pseudo code used for architecture with one Data ALU and m accumulators. For every node v of the input G_{op}, we use a variable $count(v)$ to record the required number of accumulators when v is executed. If there exists any $count(v)$ greater than m, corresponding spill codes must be inserted. Detailed steps for inserting spill

1. Input: $G_t = (V_t, E_t, X, P)$, $G_{op} = (V_{op}, E_{op}, S)$, m;
2. Output: $G_t' = (V_t', E_t', X', P')$, $G_{op}' = (V_{op}', E_{op}', S')$;
3. $G_t' = G_t$; $G_{op}' = G_{op}$;
4. $\forall v \in V_t'$, $count(v) = -1$; // Initialize
5. $count(v) = 1$, $\forall v \in V_t'$ and v doesn't have any predecessor;
6. **While** ($\exists\, count(v) = = -1$)
   - 6.1. $\exists\, e_{ij} \in E_t'$, such that v_i is the only predecessor of v_j
     - 6.1.1. **If** (v_i has successors other than v_j) $count(v_j) = count(v_j)+1$;
       Else $count(v_j) = count(v_i)$;
     - 6.1.2. **If** ($count(v_j) > m$) Insert corresponding spill codes;
   - 6.2. $\exists\, e_{ik}, e_{jk} \in E_t'$, such that v_k has two predecessors of v_i and v_j
     - 6.2.1. **If** (both v_i and v_j have successors other than v_k)
       If ($count(v_i) = = count(v_j)$) $count(v_k) = count(v_i)+2$;
       Else $count(v_k) = \mathbf{Max}(count(v_i), count(v_j))+1$;
       Else if (only v_i has successors other than v_k)
       $count(v_k) = \mathbf{Max}(count(v_i)+1, count(v_j))$;
       Else if (only v_j has successors other than v_k)
       $count(v_k) = \mathbf{Max}(count(v_i), count(v_j)+1)$;
       Else if ($count(v_i) = = count(v_j)$) $count(v_k) = count(v_i)+1$;
       Else $count(v_k) = \mathbf{Max}(count(v_i), count(v_j))$;
     - 6.2.2. **If** ($count(v_k) > m$)
       If (both v_i and v_j have successors other than v_k)
       Insert corresponding spill codes;
       Else if (both v_i and v_j have only one successor v_k)
       Insert corresponding spill codes;
       Else Insert corresponding spill codes;
7. **Return** (G_t', G_{op}');

Figure 14. Pseudo code to predict accumulator spills for architecture with one Data ALU and m accumulators.

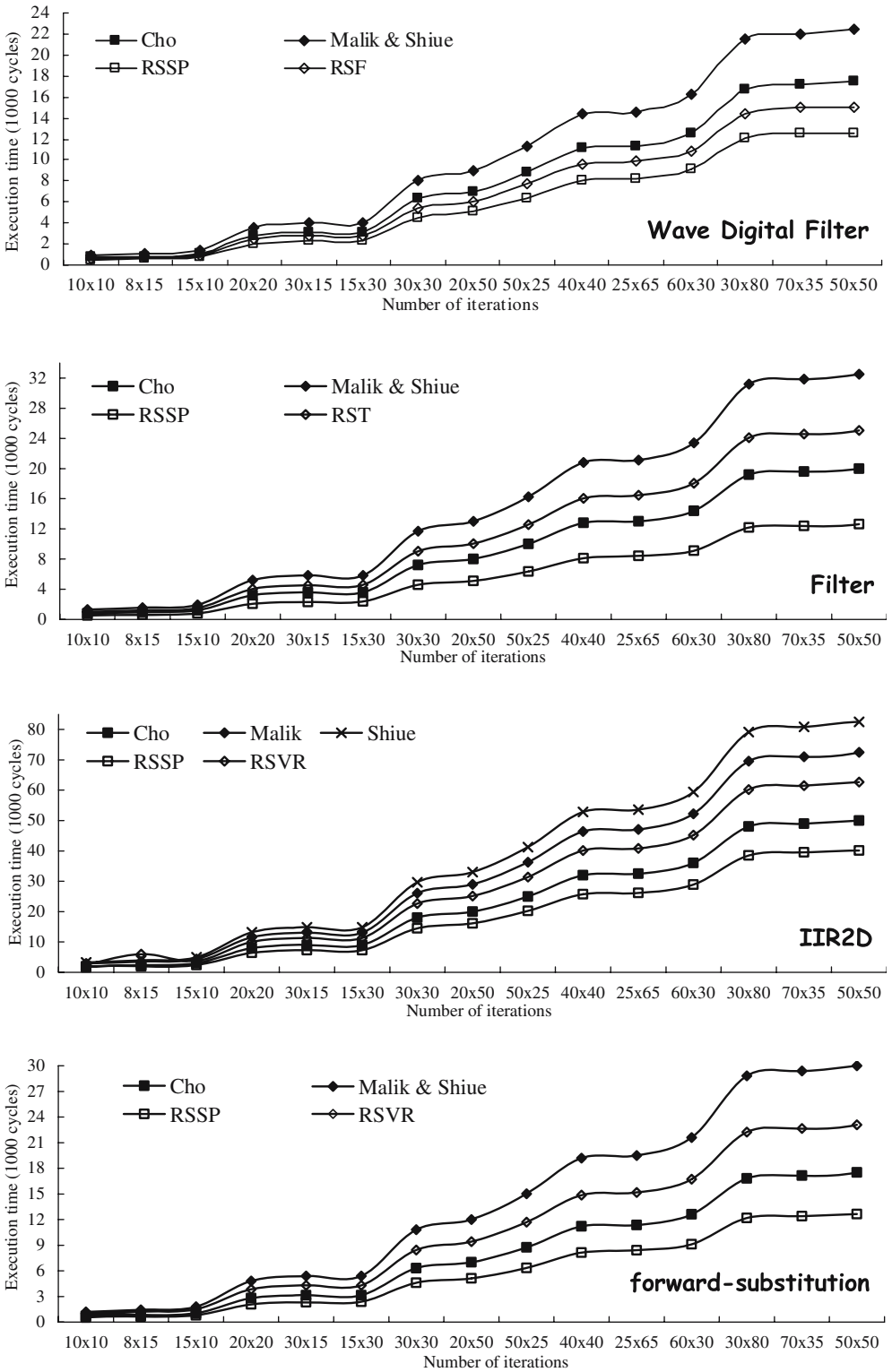


Figure 15. Experimental results.

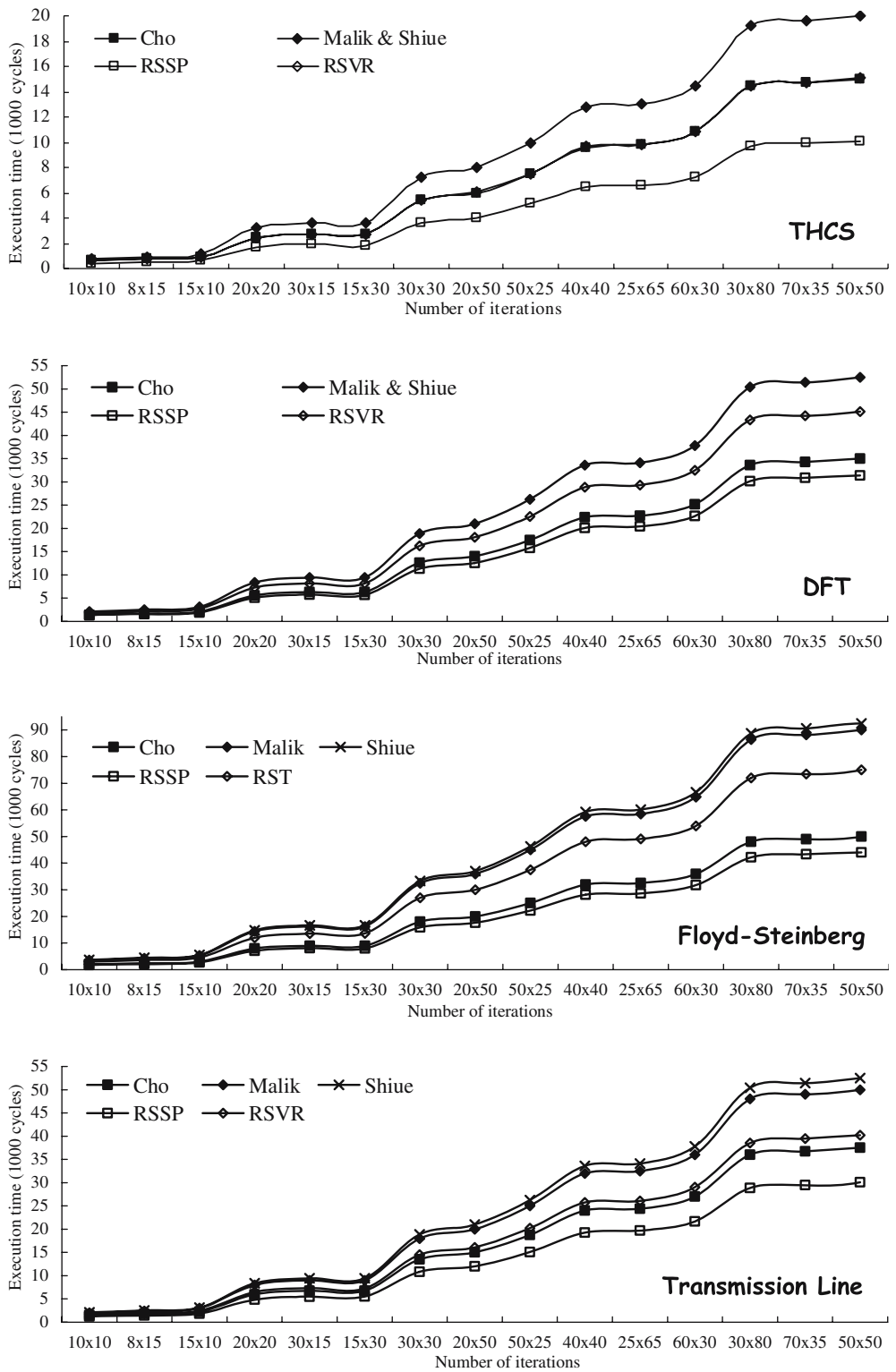


Figure 16. Experimental results.

codes we have not completed yet. From this pseudo code, we can find it is much more complicated than those algorithms introduced in Subsection 3.2.3. Besides, predicting results obtained by this pseudo code are not accurate enough, because *count*(v) is usually greater than the actually required number of accumulators when v is executed. In the future we will continue to complete and improve this pseudo code.

As for the target architecture contains k Data ALU, we plan to partition the input G_{op} into k subgraphs. Then, each subgraph is allocated to one Data ALU, and pseudo code listed in Fig. 14 can be used to schedule nodes of it. Detailed graph partitioning mechanism is still under our design. In addition, we also want to propose a single algorithm to predict accumulator spills for architecture with k Data ALU and m accumulators. After replacing the new predicting algorithm into RSSP, we will have a general code generation method which suits for various DSPs with similar architectural features. In the future we will continue to work on related research.

4. Performance Studies

Because nested loops used in DSP applications usually have a depth of two, in this paper we only experimented with our method on two-dimensional MDFGs. Nevertheless, RSSP can be easily extended to cover nested loops with depths greater than two. The following methods are all evaluated: Cho [6], Sudarsanam and Malik [7], Shiue [8], RSVR [5], RSF [11], RST [11], RSSP (with RSVR mechanism), RSSP (with RSF mechanism), and RSSP (RST mechanism). Last three are derived from the proposed RSSP, which apply different variable partitioning mechanisms. The target architecture is the same as Motorola DSP56000, which consists of one ALU, two memory banks, two accumulators, and four registers (two for each memory bank). All operations can be completed in one control step.

For a retimed (nested) loop, its execution process contains three parts prologue, repetitive pattern, and epilogue. Prologue and epilogue are instruction sets that must be executed before and after repetitive pattern, respectively. The repetitive pattern will be iterated many times, which will dominate the entire computation performance of the given loop. Therefore, we first focus on a single iteration in the repetitive pattern to compare above methods. Table 1 lists scheduling results of all applications. From these results it is clear that methods scheduled from TDAG, including Cho [6] and RSSP, outperform methods scheduled from MDFG. The intuitive

Table 1. Experimental results (for a single iteration in the repetitive pattern).

	[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	11	11	10	6	5.5	5
IIR2D	20	29	33	25	28.5	28	16	16	16
Forward-substitution	7	12	12	9	9.5	10.5	5	5.5	5
THCS	6	8	8	6	7	6	4	4	4
Discrete Fourier transform	14	21	21	18	20.5	18	13	12.5	13
Floyd–Steinberg	20	36	37	30	32	30	18	17.5	17
Transmission line	15	20	21	16	20	20	12	12	12

[1] Cho et al. [6]
[2] Sudarsanam and Malik [7]
[3] Shiue [8]
[4] RSVR [5]
[5] RSF [10]
[6] RST [10]
[7] RSSP (with RSVR mechanism)
[8] RSSP (with RSF mechanism)
[9] RSSP (with RST mechanism)

reason is that we remove additional memory accesses during construction of the TDAG in advance of processing. Furthermore, schedule lengths obtained by RSSP with various variable partitioning mechanisms are all shorter than those reported by Cho [6], Sudarsanam and Malik [7], and Shiue [8]. This is because the retiming technique is applied to compact the initial schedule, which can explore the potential instruction-level parallelism between successive iterations. The effectiveness among three cases of RSSP for most applications is very similar. This indicates that RSSP is sufficiently flexible and can obtain reasonable execution results using various mechanisms for memory bank assignment.

In the following, we evaluate above methods in view of the entire nested loop. Many previous studies introduce the time required to run prologue and epilogue are negligible while the nested loop contains sufficient iterations. However, if an unsuitable schedule vector is used, prologue and epilogue may still occupy considerable part of the overall execution time. In [11] we have already designed an analytic model to calculate the overall execution time of a two-dimensional MDFG scheduled using methods RSVR, RSF, and RST. Their proposed formulas are listed as follows, which can be directly used in experiments on RSSP.

Assumption:

1. A nested loop with depth two, and its loop bounds of outer and inner loops are m and n
2. The nested loop can be tiled directly
3. The target architecture contains N memory banks
4. Schedule vector (s_1, s_2)

Variables:

1. *length*, *prologue*, and *epilogue* to represent corresponding schedule lengths
2. *list* is the schedule length of a single repetitive iteration produced by the *List Scheduling*
3. d is the number of iterations must be moved into the prologue and epilogue
4. *half* (k, N) is the schedule length of k original iterations under N memory banks

Formula 1: (used for RSSP with RSVR mechanism)

$$\begin{aligned}&\text{overall execution time}\\&= length \times (m - s_2 d)(n - s_1 d)\\&\quad + (prologue + epilogue)\\&\quad \times (s_1 m + s_2 n - s_1 s_2 - 2d s_1 s_2)\\&\quad + list \times s_1 s_2 d(d+1)\end{aligned}$$

Formula 2: (used for RSSP with RSF mechanism)

$$\begin{aligned}&\text{overall execution time}\\&= length \times (m - s_2 d)(\lfloor n/N \rfloor - s_1 d)\\&\quad + (prologue + epilogue)\\&\quad \times (s_1 m + s_2 \lfloor n/N \rfloor - s_1 s_2 - 2d s_1 s_2)\\&\quad + list \times s_1 s_2 d(d+1)\\&\quad + half(n \bmod N, N) \times m\end{aligned}$$

Formula 3: (used for RSSP with RST mechanism)

$$\begin{aligned}&\text{overall execution time}\\&= length \times (\lfloor m/N \rfloor - s_2 d)(n - s_1 d)\\&\quad + (prologue + epilogue)\\&\quad \times (s_1 \lfloor m/N \rfloor + s_2 n - s_1 s_2 - 2d s_1 s_2)\\&\quad + list \times s_1 s_2 d(d+1)\\&\quad + half(m \bmod N, N) \times n\end{aligned}$$

Figures 15 and 16 show the overall execution time of every application calculated by above formulas. For methods proposed in [11] and RSSP, we only sketch the best results among using three variable partitioning mechanisms. In this figure, it is clear that

as the size of nested loop increases, the difference in execution times between all methods increases. That is, the methods proposed in this paper can save more execution time in larger problem sizes.

5. Conclusions and Future Work

In this paper, we propose a method named RSSP to schedule nested loops in a Digital Signal Processor with multiple memory banks and a heterogeneous register set. It contains six parts to schedule all operations while considering the limited resources and applies a retiming technique to explore the potential parallelism between iterations. With various variable partitioning mechanisms, three scheduling results are derived from RSSP. An analytic model and DSP applications were used to evaluate its computational performance. Evaluation results shows that the proposed RSSP is very effective compared with other published research. Furthermore, in order to make RSSP suit various DSPs with similar architectural features, we also present some preliminary ideas for designing its general version.

Apart from the features described above there remain several promising issues for future research. At first we will continue to design the general version of RSSP as presented in Section 3.3. Then, based on the parallel move conditions listed in [19], a special addressing mode also must be satisfied when simultaneously executing multiple memory accesses. Moreover, each memory access may be performed only if an address register is available that points to the correct memory location. However, in our proposed method, we have not considered the memory offset assignment of each variable and the address register allocation. Therefore, to make our proposed method more complete, we will extend our method to include this. Finally, in addition to high data throughput, low power consumption is another significant factor in DSP architecture. Some instruction level power models and related scheduling methods are proposed, in order to reduce the power consumption from the point of view of software. In the near future, we will study instruction level power models and try to design energy-efficient code generation algorithms, which can optimize both schedule length and power consumption as well.

References

1. Z. Wang and X. S. Hu, "Power Aware Variable Partitioning and Instruction Scheduling for Multiple Memory Banks," *Proc. of Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, 2004, pp. 312–317.
2. J. Eyre and J. Bier, "The Evolution of DSP Processors," *IEEE Signal Process. Mag.*, vol. 17, no. 2, 2000, pp. 43–51.
3. P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology, Inc., 1996.
4. V. K. Madisetti, *VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis*, Butterworth-Heinemann, 1995.
5. Q. Zhuge, B. Xiao, and E. H.-M. Sha, "Exploring Variable Partitioning for Dual Data-memory Bank Processors," *Proc. of 34th International Symposium on Microarchitecture*, 2001, pp. 45–52.
6. J. Cho, Y. Paek, and D. Whalley, "Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms," *Proc. of ACM Joint conference LCTES-SCOPES*, 2002, pp. 130–138.
7. A. Sudarsanam and S. Malik, "Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs," *ACM Transact. Des. Automat. Electron. Syst.*, vol. 5, no. 2, 2000, pp. 242–264.
8. W.-T. Shiue, "Energy-efficient Backend Compiler Design for Embedded Systems," *Proc. of 10th International Conference on Electrical and Electronic Technology*, vol. 1, 2001, pp. 103–109.
9. C. Kessler and A. Bednarski, "Optimal Integrated Code Generation for Clustered VLIW Architectures," *Proc. of ACM Joint conference LCTES-SCOPES*, 2002, pp. 102–111.
10. M. A. R. Saghir, P. Chow, and C. G. Lee, "Exploiting Dual-memory Banks in Digital Signal Processors," *Proc. of 7th International Conference on Architecture Support for Programming Language and Operating Systems*, 1996, pp. 234–243.
11. Y.-H. Lee and C. Chen, "Efficient Variable Partitioning and Scheduling Methods of Multiple Memory Modules for DSP," *Proc. of 10th Workshop on Compiler Techniques for High-Performance Computing*, 2004, pp. 80–89.
12. M. A. R. Saghir, P. Chow, and C. G. Lee, "Towards Better DSP Architectures and Compilers," *Proc. of International Conference on Signal Processing Applications and Technology*, 1994, pp. 658–664.
13. R. Leupers and D. Kotte, "Variable Partitioning for Dual Memory Bank DSPs," *Proc. of International Conference on Acoustics, Speech, and Signal Processing*, 2001, vol. 2, pp. 1121–1124.
14. J. M. Daveau, T. Thery, T. Lepley, and M. Santana, "A Retargetable Register Allocation Framework for Embedded Processors," *Proc. of ACM SIGPLAN/SIGBED*, 2004, pp. 202–210.
15. B. Scholz and E. Eckstein, "Register Allocation for Irregular Architectures," *Proc. of ACM Joint conference LCTES-SCOPES*, 2002, pp. 139–148.

16. X. Zhuang, T. Zhang, and S. Pande, "Hardware-managed Register Allocation for Embedded Processors," *Proc. of ACM SIGPLAN/SIGBED*, 2004, pp. 192–201.
17. L. Lamport, "The Parallel Execution of DO Loops," *Comm. ACM* (SIGPLAN), vol. 17, no. 2, 1974, pp. 82–93, Feb.
18. C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, no. 1, 1991, pp. 5–35.
19. *DSP56000/DSP56001 Digital Signal Processor User's Manual*, Motorola Inc., Phoenix, AZ.

Yi-Hsuan Lee is a Ph.D. candidate in Computer Science and Information Engineering at National Chiao Tung University, Taiwan, R.O.C. She received her B.S. degree in Computer Science and Information Engineering at National Chiao Tung University, Taiwan, R.O.C. in 1999. Her current research interests include computer architecture, parallelizing compiler techniques, multiprocessor scheduling problem, and scheduling problem in DSP architecture.

Cheng Chen is a professor in the Department of Computer Science and Information Engineering at National Chiao Tung University, Taiwan, R.O.C. He received his B.S. degree from the Tatung Institute of Technology, Taiwan, R.O.C. in 1969 and M.S. degree from the National Chiao Tung University, Taiwan, R.O.C. in 1971, both in Electrical Engineering. Since 1972, he has been on the faculty of National Chiao Tung University, Taiwan, R.O.C. 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 Chaio 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 Center (MISC) in National Chiao Tung University. His current research interests include computer architecture, parallel processing system design, and parallelizing compiler techniques.