Code Generation of Nested Loops for DSP Processors with Heterogeneous Registers and Structural Pipelining

Wei-Kai Cheng
and
Youn-Long Lin
Department of Computer Science
Tsing Hua University
Hsin-Chu, Taiwan 30043, R.O.C.

We propose a microcode optimizing method targeting a programmable DSP processor. Efficient generation of microcodes is essential to better utilize the computation power of a DSP processor. Since most state-of-the-art DSP processors feature some sort of irregular architectures, and most DSP applications have nested loop constructs, their code generation is a non-trivial task. In this paper, we consider two features frequently found in contemporary DSP processors – structural pipelining and heterogeneous registers. We propose a code generator that performs instruction scheduling and register allocation simultaneously. The proposed approach has been implemented and evaluated using a set of benchmark core algorithms. Simulation of the generated codes targeted towards the TI TMS320C40 DSP processor shows that our system is indeed more effective compared with a commercial optimizing DSP compiler.

Categories and Subject Descriptors: A.4 [**System Design**]: Embedded System

General Terms: Compiler

Additional Key Words and Phrases: Code Generation, DSP

---

1. INTRODUCTION

Many applications require performing various computationally intensive digital signal processing (DSP) algorithms. When a pure software implementation (on a general-purpose processor) cannot deliver adequate performance, users have to seek

---

Address: Department of Computer Science, Tsing Hua University, Hsin-Chu, Taiwan 30043, R.O.C.

solutions in either ASIC or programmable DSP processors. While the ASIC approach can achieve the highest level of performance, the programmable DSP processor approach can lower the design cost and shorten the design time. In both cases, the microcode generation problem is very important as it significantly affects the overall performance.

In Paulin's survey [Paulin et al. 1995] on future CAD requirements by a group of designers, code generation and microcode synthesis are among the most urgent needs. Paulin's survey reports that at least two thirds of the DSP applications are implemented in either programmable DSP processors or ASIPs (Application Specific Instruction Set Processors). This further stresses the importance of the code generation problem. Ptolemy [Buck et al. 1991; Pino et al. 1995], Flexware [Paulin et al. 1995], and Cathedral 2ND [Goossens et al. 1995] are all representative examples of software synthesis and code generation systems targeting towards DSP applications.

Two of the most important tasks to code generation and optimization are instruction scheduling and register allocation. In this paper, we assume that an instruction has been selected for each operation, addressing mode determined for each memory access, and address assignment finished before instruction scheduling and register allocation.

We take into account two architectural features: structural pipelining and heterogeneous registers. Structural pipelining is one of the most popular means to speed up the computation at minimal hardware overhead. However, there may be conflicts over register or memory access among instructions at different pipeline stages, resulting in postponing the execution of subsequent instructions till no more conflict exists.

In a DSP processor, due to area or speed consideration, not all registers are accessible by an instruction. In addition, when multiple instructions are issued in parallel, register usage may subject to further restriction. This heterogeneous register architecture makes register allocation and its interaction with instruction scheduling more complicated.

Facing these two architectural features, we perform instruction scheduling and register allocation simultaneously to minimize the number of pipeline conflicts and code spilling[2], and try to parallelize as many instructions as possible.

The rest of this paper is organized as follows. Previous work is surveyed in section 2. In section 3, we illustrate with examples the essential considerations and the modeling of the two architectural features. A simultaneous instruction scheduling and register allocation algorithm is proposed in section 4. In section 5, some experimental results are presented. Finally, in section 6 we draw conclusions and point to some directions for possible future research.

2. RELATED WORK

There have been a large amount of work on the instruction scheduling and register allocation problems in the compiler domain. List scheduling [Davidson et al. 1981] is a representative method for the scheduling problem while register allocation is

[2]code spilling is performed by temporarily moving a variable from the register to the memory and bringing it back later. It is used when we run out of register allocation.

usually modeled as an interference graph coloring problem [Briggs et al. 1989; Chow and Hennessy 1984; L. J. Hendren 1992]. Because of the loop execution and the heterogeneous register architecture, the register allocation problem becomes more complicated.

For loop execution, special attentions must be paid to variables with lifetimes across loop boundaries during register allocation. Callahan and Koblenz [Callahan and Koblenz 1991] group code blocks into a tree reflecting the program's hierarchical control structure, then allocate registers for each block using a standard graph coloring technique. Local allocation and conflict information is passed around the tree and spill code is placed in some less frequently executed portions.

To take into account the heterogeneous register architecture, attention has also been paid to select register for allocation in addition to determine which set of variables are assigned to the same register. Araujo et al. [Araujo and Malik 1995; Araujo et al. 1996] model the register architecture by a structural representation called the register transfer graph (RTG), and label some edges in the data flow graph (DFG) for code spilling based on some rules developed from the RTG model. But, this modeling can only be used in a very special class of register architecture. To consider different register architecture, a better way is to classify registers based on the architecture and the instruction functionality. By this method, CodeSyn [Liem et al. 1994] performs allocation using a modified left-edge algorithm [Kurdahi and Parker 1987] to minimize the amount of code spilling.

However, neither of the work described above consider both the loop construct and the heterogeneous register architecture simultaneously. Kolson et al. [Kolson et al. 1995] combine tree construction and loop unrolling to solve this problem. A tree is constructed and add to it register classes during global search for the optimal solution, and loop body is unrolled to minimize the amount of code spilling for variables across loop boundary. Unfortunately, this method has the disadvantage of increasing the code size due to loop unrolling. Moreover, it does not take the effect of instruction scheduling into consideration together with register allocation.

In traditional compiler design, these two tasks are performed separately. Instruction scheduling is performed before register allocation in the school of *prepass scheduling* [Bradlee et al. 1991; Goodman and Hsu 1988; Hwu and Chang 1988]. While in the school of *postpass scheduling* [Gibbons and Muchnick 1986; Hennessy and Gross 1983], register allocation is done first. Performing instruction scheduling earlier can maximize the parallelism. However, it may overuse the registers, resulting in code spilling. On the other hand, performing register allocation earlier can reduce the amount of code spilling. However, the available parallelism may become limited. Because of the heterogeneous register architecture, the interaction between them become much more complicated and make it necessary to optimize these two tasks simultaneously.

Shlomit [Pinter 1993] proposed an graph model called parallel interference graph to solve these two problems simultaneously. This model provides an optimal register allocation and preserves the property that no false dependences are introduced, thus all the options for parallelism are kept for scheduling. However, neither the loop construct nor the heterogeneous register architecture is considered during register allocation. Liao et al. [Liao et al. 1995] propose a branch-and-bound methodology together with a heuristic and a hashing technique for further pruning the search space of these two problems simultaneously. But their method can only be applied for a special register architecture. Some research optimized these two tasks by modifying the list scheduling algorithm [Davidson et al. 1981]. Kim and Barnwell [Kim and Barnwell 1990] extend the algorithm with partial backtracking along which registers are allocated. Rimey and Hilfinger [Rimey and Hilfinger 1988] use it to schedule function operations. They insert data transfer operations into existing control steps, and assign variables to registers dynamically by employing a look-ahead technique based on the network flow algorithm. But neither of them consider the loop construct. Wilson et al. [Wilson et al. 1994] propose an ILP formulation for all these problems. The formulation map combinations of generic DFG operations onto more inclusive special machine instructions, and allocate registers under loop construct and heterogeneous register architecture. However, the ILP formulation has difficulty in extending to the nested loop construct.

Our algorithm also perform instruction scheduling and register allocation simultaneously. Like Callahan and Koblenz [Callahan and Koblenz 1991], we allocate registers hierarchically for nested loop constructs. During both register allocation and instruction scheduling, we take into account the heterogeneous register architecture by means of register classification. In addition, we take into account the effect of structural pipelining during instruction scheduling which is not covered in any of the previous work described. As soon as an instruction is scheduled, we allocate registers for it to guide the subsequent scheduling and allocation.

3. ESSENTIAL CONSIDERATION AND MODELING

In this section, we will give a model for each of the pipeline structure and the instruction set architecture. The pipeline structure model defines the conditions that cause pipeline stall. The instruction set architecture model defines the accessible registers for instructions, and describes how registers are classified into register sets. We illustrate with examples the impact of these two architectural features on instruction scheduling and register allocation.

3.1 Structural Pipelining

We consider two kinds of pipeline execution conflicts. The first kind is due to the use of address register in indirect addressing. Many DSP processors have registers that are used to store addresses. Before the correct address value is loaded into the address register, the address for memory access cannot be generated during the instruction decode stage, causing pipeline stall. The other is due to memory bandwidth constraint. If the total number of memory accesses from all stages in a cycle exceed the constraint, the pipeline will also be stalled.

To model the pipeline execution conflicts, we specify pipeline execution model and memory access architecture. A pipeline execution model is defined by a record $P :< F, D, R, W, L >$ to specify the number of stages for instruction fetch (F), instruction decode and address generation (D), memory data read (R), memory data write (W), and address loading (L). The value $L - D + 1$ is the minimum number of cycles between the scheduling of an instruction using the address register for indirect addressing and the instruction that loads the address value. The other three terms of record P are used together with the memory access architecture to model the memory access constraints.

Memory access includes instruction fetch and data access, which in turn includes both data read and data write. We assume the existence of both on-chip (internal) and off-chip (external) memory. The on-chip memory is divided into several blocks. The memory access architecture is defined by four records, $M_t :< T, I, D_t, D_r, D_w >$, $M_i :< T, I, D_t, D_r, D_w, B >$, $M_e :< T, I, D_t, D_r, D_w >$, and $B_n :< T, I, D_t, D_r, D_w >$. Record M_t defines constraint on the total number of memory accesses per instruction cycle. Records M_i and M_e define the constraint on the number of simultaneous memory accesses to the internal and the external memory, respectively. Record B_n specifies the number of simultaneous memory accesses to the nth internal memory block. In each record, T denotes the constraint on the total number of memory access including instruction fetch and data access. I denotes the constraint on the number of instruction fetches. D_t, D_r, and D_w denote the constraints on the number of data access, data read, and data write, respectively. B denotes the number of internal memory blocks. To minimize the number of pipeline conflicts due to violation against the memory access constraint, we have to take into account all instructions at various stages of execution during a cycle.

We illustrate the pipeline conflicts in a DSP processor–TI TMS320C40 [TI. 1991a]. Its instruction pipeline consists of four stages: *instruction fetch (F), instruction decode (D), operand fetch (R),* and *execute (E).* Operands are fetched during the *operand fetch* stage if they are stored in the memory, or during the *execute* stage if they are stored in the registers. The internal memory has one ROM block (B_1) and two RAM blocks (B_2, and B_3). The records are specified as $P :< 1, 2, 3, 4, 4 >$, $M_t :< 3, 1, 2, 2, 2 >$, $M_i :< 3, 1, 2, 2, 2, 3 >$, $M_e :< 2, 1, 1, 1, 1 >$, $B_1 :< 2, 1, 2, 2, 0 >$, and $B_2 = B_3 :< 2, 1, 2, 2, 2 >$.

Figure 1 depicts a pipeline conflict example due to usage of address register. Rx (x = 1, 2, ...) denotes a data register, while ARx (x = 1, 2, ...) an address register. The second instruction (MPYF) must fetch from the memory an operand whose address is held in AR2. During the fourth cycle, the second instruction cannot read its operand from the memory because the address has not yet been loaded into AR2. Therefore, it must wait for two cycles. To avoid the conflict, the MPYF instruction must be scheduled at least 3 $(4 - 2 + 1)$ cycles after the LDI instruction.

Figure 2(a) shows a memory access conflict example. Suppose that instructions are stored in the internal ROM block, and data are stored in the internal RAM block B_2. The simultaneous execution of two STF (store) instructions results in two data transfers from the register to the memory during the fourth cycle. Hence, the third instruction (ADDF) cannot fetch its operand from the memory during the fourth cycle, resulting in a one-cycle delay. As shown in Figure 2(b), if we pack into one microcode word the instruction ADDF together with any one of the two STF instructions (schedules #1 and #2), or schedule the instruction ADDF before the pair of STF instructions (schedule #3), there will be no conflict.

3.2 Instruction Set Architecture and Register Classification

We divide the instructions into two subsets: *Simple_Inst* for instructions that perform only one operation, and *Complex_Inst* for those perform multiple operations. Let $inst\_s_i$ denotes the ith instruction of *Simple_Inst*, $inst\_c_i$ the ith instruction of *Complex_Inst*, and $oper\_c_i^j$ the jth operation of $inst\_c_i$. For each $inst\_s_i$ and

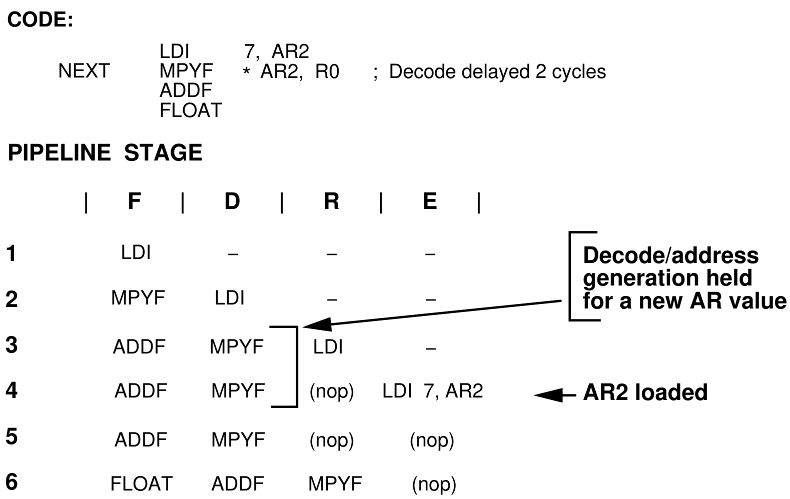


Fig. 1. An example of pipeline conflict due to usage of address register.

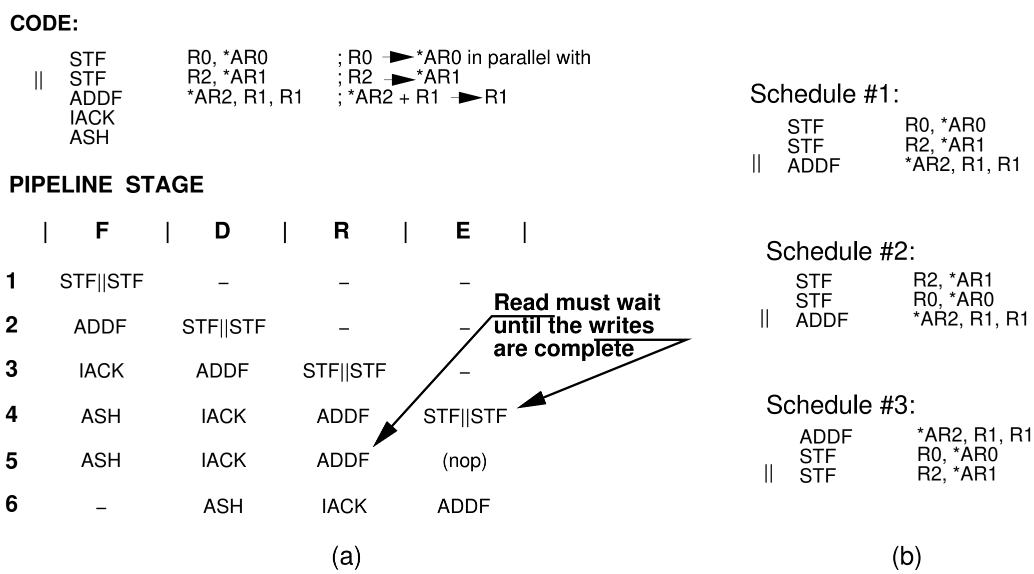


Fig. 2. (a) An example of memory access conflict; (b) Scheduling to eliminate memory access conflict.

$oper\_c_i^j$, we specify the applicable addressing modes and the accessible registers for each addressing mode for all operands.

The applicable addressing modes include: register addressing, direct addressing, indirect addressing, immediate addressing, and program counter (PC) relative addressing. Except the immediate addressing mode, all others have to access registers. For each of the three operands of a simple instruction $inst\_s_i$ and a parallel operation $oper\_c_i^j$, we specify registers to be used as data register in register addressing, data page register in direct addressing, address register and index register in indirect addressing, and program counter in PC relative addressing.

As not all registers are accessible by all instructions, and the set of registers accessible by an instruction may depends on whether or not the instruction is executed in parallel with others, we classify registers into register sets as a basis in the design of cost functions for instruction scheduling and register allocation. The classification is done according to the union of register access range of instructions.

Figure 3 shows the register classification of the TMS320C40 processor. Figure 3(a) shows the simplified datapath. The datapath contains one multiplier and one ALU, two auxiliary register arithmetic units (ARAUs) capable of generating two addresses in a single cycle, and three register files. The extended precision register file contains 12 registers for storing and supporting 32-bit integer and 40-bit floating point numbers. The auxiliary register file contains 8 registers for generating 32-bit addresses or used as 32-bit general purpose registers. The remaining 12 registers are used for special functions such as indexing, data page register, or program counter, ...etc. They can also be used as 32-bit general purpose registers. The function of the registers are listed in Figure 3(b).

Figure 3(c) shows the register classification for different addressing modes. RS_s^{pc}, RS_s^{in}, RS_s^{ptr}, RS_s^{pge}, and RS_s^{d} denote the set of accessible registers by instructions in $Simple\_Inst$ that are used as program counter, index register, address pointer, data page register, and data register, respectively. For those instructions in $Complex\_Inst$, they are denoted as RS_c^{pc}, RS_c^{in}, RS_c^{ptr}, RS_c^{pge}, and RS_c^{d}.

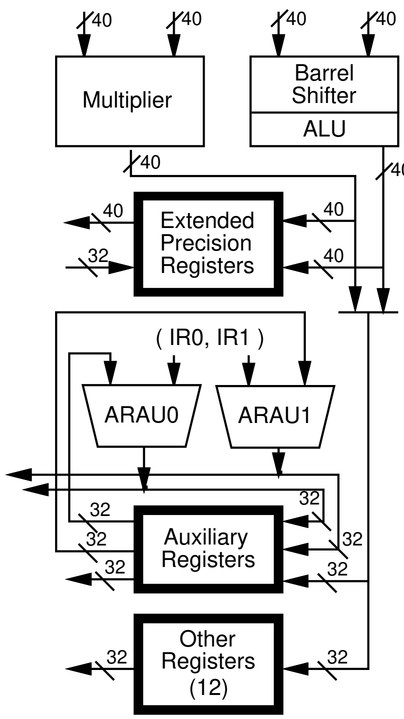
By the union of all the classifications in Figure 3(c), Figure 3(d) shows that there are 17 register sets after the final register classification. We can see that the register classification is not disjoint. Some registers are in multiple register sets, and some register sets are subset of other register sets. As its accessible registers may change when an instruction is executed in parallel with others, the interaction between instruction scheduling and register allocation become more complicated.

4. CODE GENERATION

In this section, we will first describe the code generation order for the basic blocks in a nested loop structure. Then, we propose an algorithm for instruction scheduling and register allocation of a basic block. Finally, we will define cost functions for selecting instructions for scheduling and registers for allocation in the algorithm.

4.1 Code Generation Order

We generate codes for basic blocks one at a time starting with the inner most block. Because of this ordering, variables that are accessed in both the inner and outer blocks must have been assigned to registers before code generation of the outer block. Therefore, before code generation for each basic block, we classify its



(a)

CPU Registers :

R0~R11 :	Extended Precision Registers
AR0~AR7 :	Auxiliary Registers
IR0~IR1 :	Index Registers
DP :	Data Page Register
BK :	Block Size Register
SP :	System Stack Pointer
ST :	Status Register
DIE :	DMA Interrupt Enable Register
IIE :	Internal Interrupt Enable Register
RS :	Block Repeat Start Register
RE :	Block Repeat End Register
RC :	Repeat Count Register
PC :	Program Counter

(b)

Register Classification for Different Addressing Modes :

$RS_s^{pc} = \{\{PC\}\}$ $\quad RS_c^{pc} = \phi$

$RS_s^{in} = \{\{IR0, IR1\}\}$ $\quad RS_c^{in} = \{\{IR0, IR1\}\}$

$RS_s^{ptr} = \{\{AR0,.., AR7\}\}$ $\quad RS_c^{ptr} = \{\{AR0,.., AR7\}\}$

$RS_s^{pge} = \{\{DP\}\}$ $\quad RS_c^{pge} = \phi$

$RS_s^{d} = \{\{All\ Reg\}, \{R0,.., R11\}, \{AR0,.., AR7\}, \{BK\}, \{SP\}, \{ST\}, \{DIE\}, \{IIE\}, \{RS\}, \{RE\}, \{RC\}\}$

$RS_c^{d} = \{\{R0,.., R7\}, \{R0, R1\}, \{R2, R3\}\}$

(c)

Final Register Classification

{{All Reg} , {R0~R11} , {R0~R7} , {R0~R1} , {R2~R3} , {AR0~AR7} ,
{IR0~IR1} , {DP} , {BK} , {SP} , {ST} , {DIE} , {IIE} , {RS} , {RE} , {RC} , {PC}}

(d)

Fig. 3. Register classification of TMS320C40: (a) The simplified datapath; (b) The CPU registers; (c) Register classification for different addressing modes; (d) The final register classification.

variables into five types as following:

(1) *internal*: variables that are produced and then consumed within a basic block during the same iteration.

(2) *input_assigned*: variables that are produced outside the basic block or during some previous iterations, and have been assigned to registers.

(3) *input_unassigned*: variables that are produced outside the basic block or during some previous iterations, but have not yet been assigned to registers.

(4) *output_assigned*: variables that are produced within the basic block, still alive at the end of an iteration, and have been assigned to registers.

(5) *output_unassigned*: variables that are produced within the basic block, but have not yet been assigned to registers.

Figure 4(a) shows a nested loop program. Figure 4(b) shows the type of each variable. None of the variables in the inner block have yet been assigned to registers before code generation. Therefore, all variables that are written are of type *output_unassigned* while all variables that are read are of type *internal* or *input_unassigned*. Note that variable v_3 is of type *input_unassigned* in the 9th line because it is produced during the previous iteration while in the 11th line it is of type *internal* because it is produced by an instruction in the same iteration. For variables in the outer block, all except v_5 and v_7 have been assigned to registers in the inner block. Therefore, all variables that are written except v_7 are of type *output_assigned* while all variables that are read except v_5 and v_7 are of type *input_assigned*.

```
1   for ( loop 1 )
2    {
3      v7 = v1 + v5
4      v8 = v2 + v7
5      v6 = v3 + v4
6      for ( loop 2 )
7       {
8         v1 = v2 + v8
9         v4 = v3 + v6
10        v3 = v1 + v4
11        v8 = v1 + v3
12      }
13   }
```

(a)

```
1   for ( loop 1 )
2    {
3     o_u= i_a+ i_u
4     o_a= i_a+  i
5     o_a= i_a+ i_a
6     for ( loop 2 )
7      {
8        o_u= i_u+ i_u
9        o_u= i_u+ i_u
10       o_u=  i  +  i
11       o_u=  i  +  i
12     }
13   }
```

Notations:
i : internal
i_a : input_assigned
i_u : input_unassigned
o_a : output_assigned
o_u : output_unassigned

(b)

Fig. 4. Variable type example: (a) A nested loop program; (b) Variable type of each variable in (a).

4.2 Instruction Scheduling and Register Allocation

We propose a transformation–based heuristic algorithm for simultaneous instruction scheduling and register allocation. We adopt some techniques from high level synthesis [Huang et al. 1990; Lee et al. 1992]. In the first phase, instructions are scheduled as soon as possible. Only the data dependency constraint is taken into account while all other types of violations are ignored. What obtained after this phase is a maximally parallel schedule. During the second phase, we resolve all violations by a rescheduling procedure. In the mean time, register allocation (and re-allocation) is performed using a bipartite weighted matching algorithm [Papadimitriou and Steiglitz 1982]. We design cost functions for both instruction rescheduling and register allocation such that there exist as many register sets having empty registers for later use as possible. Therefore, we can reduce the amount of code spilling while making available as many instructions as possible for parallel execution. In addition, we also design cost function for minimizing the number of delays caused by pipeline conflicts during rescheduling.

Figure 5 illustrates the rescheduling algorithm. Figure 5 (a) depicts the ASAP scheduling of a data flow graph. Violations are resolved one step at a time. Suppose that there is no violation in the first two control steps, and variables v_1, v_2, v_3, v_5, v_6, v_9, and v_{10} have been successfully assigned to registers. The third control step is checked now to see whether there is any violation. Firstly, we inspect whether the instructions in this step satisfy all but the constraint on register allocation for parallel execution. If there exists no violation, we inspect whether the registers allocated to the assigned input variables (i.e., v_3, v_5, and v_6) satisfy the architectural constraint, and allocates a register for the unassigned variable (v_4) if possible. Then, we inspect whether the register allocated to the assigned output variable (v_{10}) satisfies the architectural constraint, and allocate registers for the unassigned output variables (v_7 and v_8). As an instruction may be given fewer accessible registers when it is executed in parallel with others, violations are resolved by rescheduling some instructions to the next step if there exists any violation in the above three inspections. We reschedule one instruction at a time. The selection of instruction for rescheduling is based on a priority function to be defined in the next subsection. Figure 5 (b) shows the result when instruction 3 is selected and rescheduled.

We use a bipartite matching procedure in the violation check procedure described above to allocate registers for the unassigned input variables and output variables, separately. By means of bipartite matching, we can allocate registers for multiple variables at a time in a global way. Figure 5 (c) illustrates how the bipartite matching works on the unassigned output variables in the third control step of Figure 5 (a). In the graph, there are two partite of nodes, one representing variables, and the other registers. In this case, we are to assign two variables (v_7 and v_8) to five registers (r_1, r_2,...,r_5) in the optimal way. Each edge weight denotes the gain of assigning the corresponding variable to the register. The calculation of the weight will be described in section 4.4. On the right hand side of Figure 5 (c) we show that the maximal weight matching is to assign v_7 to r_2 and v_8 to r_5 with a total weight of 27.

Figure 6 gives a pseudo code description of the algorithm. Functions *inst_check*(), *input_reg_check*(), and *output_reg_check*() inspect the violations described above, respectively. If there exists any violation and more than one instruction have been scheduled into this control step, one instruction is selected and rescheduled as illustrated above. For violations associated with an instruction that has been scheduled into a step alone, they must be due to the lack of empty registers, and must be

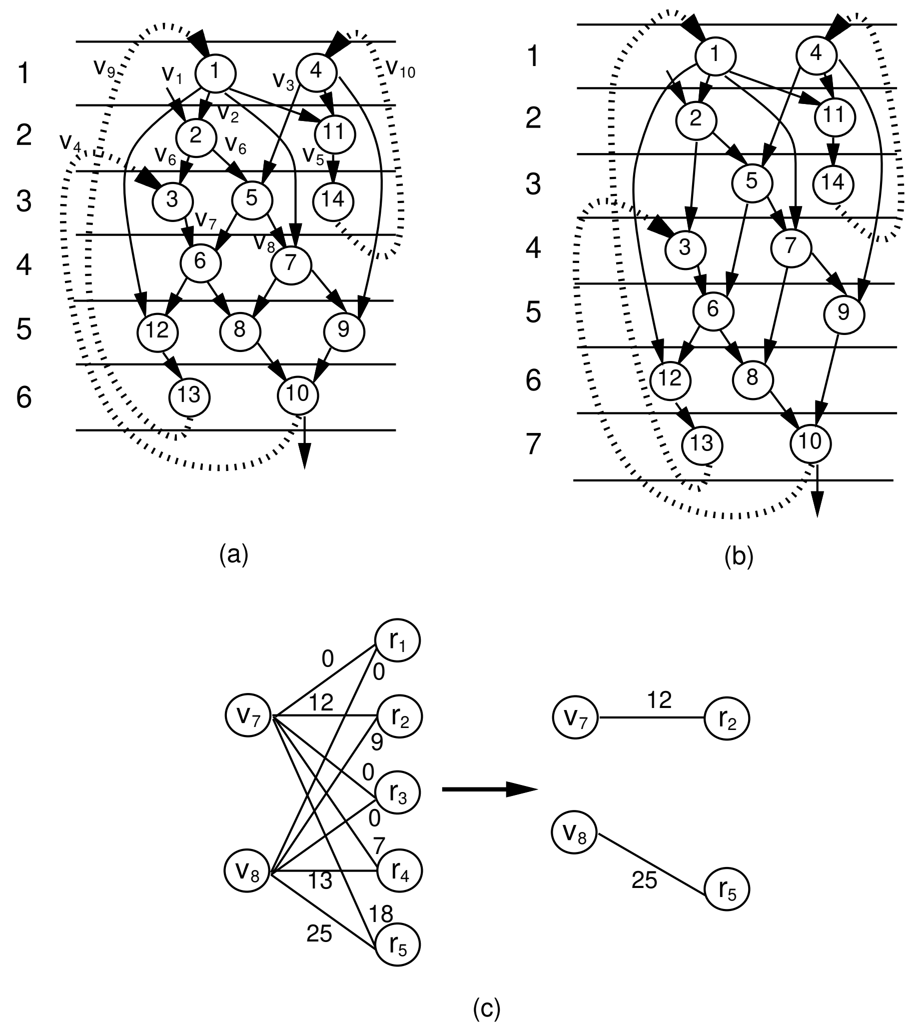


Fig. 5. An illustrative example of the rescheduling algorithm: (a) An ASAP scheduling; (b) Rescheduling of instruction 3; (c) Assignment of v_7 and v_8 by the bipartite weighted matching method.

resolved by code spilling. The algorithm proceeds to the next control step after having resolved all violations, and terminates itself when no more step violates any constraint.

```
Algorithm Code_Generation ()
    /** Phase 1 **/
    ASAP_Sch();
    /** Phase 2 **/
    for Step_index = 1 to TotalStep do
        violation_flag = true;
        while violation_flag is true do /** exist violation **/
            VI_index = set of input variables in Step_index;
            VO_index = set of output variables in Step_index;
            inst_flag = inst_check(); /** begin inspections **/
            if inst_flag is false then
                violation_flag = true;
            else
                input_flag = input_reg_check(VI_index);
                if input_flag is false then
                    violation_flag = true;
                else
                    output_flag = output_reg_check(VO_index);
                    if output_flag is false then
                        violation_flag = true;
                    else
                        violation_flag = false;
                    endif
                endif
            endif /** end inspections **/
            if violation_flag is true then
                if TotalOp_Step_index > 1 then /** #inst > 1 **/
                    Op = the instruction with the highest priority for rescheduling;
                    reschedule(Op);
                else /** #inst = 1 **/
                    code_spilling();
                endif
            endif
        endwhile
    endfor
end Code_Generation.
```

Fig. 6. The code generation algorithm.

Figure 7 shows the pseudo code description of the function *input_reg_check*(). Variables of type *input_assigned* are checked to see whether the allocated registers satisfy the architectural constraint. As variables of type *internal* were assigned to registers in the previous control steps of the current code generation block, we re-allocate registers for those variables that violate the architectural constraint (*reallocation*()) so as to increase the possibility of resolving the violation without rescheduling instructions. Then these variables are checked again to actually determine whether there exists any register allocation violation. Variables of type *input_unassigned* are firstly assigned to registers and then checked for any violation. For each variable var_i in the bipartite graph, the register reg_i allocated to it may not satisfy all the architectural constraints of instructions accessing var_i in other control steps or other basic blocks. If there exists any conflict, additional register move instructions are inserted to resolve the conflicts.

```
Function input_reg_check (VI_index)
    /* VI_index = V_input_assigned ∪ V_internal ∪ V_input_unassigned */
    V_reallocation = ∅;
    for each variable var_i ∈ V_input_assigned do
        range_flag = input_range_check(var_i);
        if range_flag is false then /** violate the constraint **/
            return (false);
        endif
    endfor
    for each variable var_i ∈ V_internal do
        range_flag = input_range_check(var_i);
        if range_flag is false then
            V_reallocation = V_reallocation ∪ var_i;
        endif
    endfor
    reallocation(V_reallocation);
    for each variable var_i ∈ V_reallocation do /** inspect reassigned variables **/
        range_flag = input_range_check(var_i);
        if range_flag is false then
            return (false);
        endif
    endfor
    matching(V_input_unassigned);
    for each variable var_i ∈ V_input_unassigned do
        range_flag = input_range_check(var_i);
        if range_flag is false then
            return (false);
        endif
    endfor
    for each variable var_i ∈ V_input_unassigned do
        reg_i = the register that allocated to var_i;
        conflict_flag = reg_conflict_check(reg_i,var_i);
        if conflict_flag is true then /** violate the constraint of some instructions **/
            insert_reg_move(reg_i,var_i); /** insert register move instructions **/
        endif
    endfor
    return (true);
end input_reg_check.
```

Fig. 7. The input register allocation and checking algorithm.

Figure 8 gives a pseudo code description of the function *reallocation*(). First, it finds the set of steps in which the variables to be reassigned are produced. Then for each step, registers are reallocated for all output variables of type *output_unassigned*. For example, if v_6 in the third control step of Figure 5 (a) violates the architectural constraint, then for all variables of type *output_unassigned* (v_5 and v_6) in the second control step where v_6 is produced, we re-allocate registers to them using the bipartite matching procedure.

```
Function reallocation (V_reallocation)
    Step_Realloc = the set of steps having at least one output variable
                 var_o ∈ V_reallocation;
    for each step Step_i ∈ Step_Realloc starting with the earliest step do
        Var_Out_i = the set of output variables in step Step_i and the
                 type is output_unassigned;
        matching(Var_Out_i);
        for each variable var_j ∈ Var_Out_i do
            reg_j = the register that allocated to var_j;
            conflict_flag = reg_conflict_check(reg_j,var_j);
            if conflict_flag is true then
                 insert_reg_move(reg_j,var_j);
            endif
        endfor
    endfor
end reallocation.
```

Fig. 8. The reallocation algorithm.

Figure 9 illustrates the cases of inserting register move instructions. Suppose basic block 2 is scheduled before basic blocks 1 and 3. Node V denotes an instruction in the current scheduling step that accesses var_i. Empty nodes denote instructions that have no assignment conflict associated with var_i. Gray nodes denote instructions that have assignment conflict associated with var_i. For the instruction, denoted as S, that produces var_i, a register move instruction, denoted as I, is inserted and a new variable var_i' is generated. Figure 9 (a) depicts the case when S exhibits no assignment conflict. The DFG is modified such that all nodes having assignment conflict now take var_i' produced by node I as input, and the remaining nodes still take var_i as input. Figure 9 (b) depicts the case when S exhibits assignment conflict. The DFG is modified such that var_i is now viewed as one produced by the inserted node I and used by all nodes that have no assignment conflict, and var_i' is produced by node S and used by all nodes having assignment conflict.

4.3 Cost Function for Instruction Rescheduling

The cost $Cost_i$ of rescheduling instruction $inst_i$ is defined as

$$Cost_i = \alpha_1 \times Resch1_i + \alpha_2 \times Resch2_i + \alpha_3 \times Resch3_i - \alpha_4 \times Resch4_i, \quad (1)$$

where $\alpha_1 \ldots \alpha_4$ are empirical parameters. The larger the value of $Cost_i$ is, the less likely that $inst_i$ will be selected for rescheduling. The four terms of the cost function are described as following:

(1) Control Step Length ($Resch1_i$):
If it will increase the total number of control steps, an instruction is not a good candidate for rescheduling.

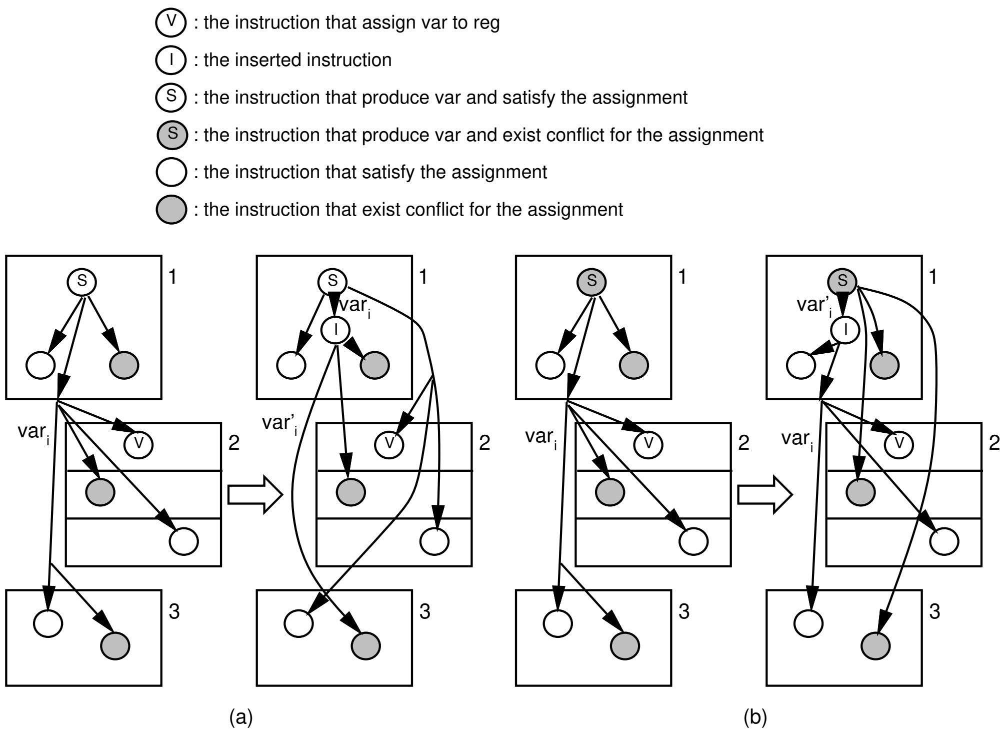


Fig. 9. An example of inserting register move instructions: (a) The case that the instruction produce var_i satisfy the assignment; (b) The case that the instruction produce var_i exist conflict for the assignment.

(2) Register Utilization ($Resch2_i$):
If its execution ends many variables' lifetime, an instruction is not preferred for rescheduling because it can release more registers for later use. In contrast, if its output operand needs to be stored in a register, an instruction is preferred for rescheduling. Because of the heterogeneous register architecture, registers have different weight. If an instruction has released registers and allocated registers of large weight difference, it is not preferred for rescheduling.

(3) Number of Pipeline Delays ($Resch3_i$):
If it will cause more pipeline delays, an instruction is not preferred for rescheduling.

(4) Amount of Parallel Execution ($Resch4_i$):
If it will result in more parallel execution, an instruction is preferred for rescheduling.

We use the scheduling of Figure 5(a) to illustrate the calculation of the cost function for rescheduling. Suppose we want to calculate the cost of rescheduling instruction 3 as shown in Figure 5(b). As the number of control step is increased by 1, the value of $Resch1_3$ is 1. If the rescheduling of $inst_i$ will not increase the number of control step, the value of $Resch1_i$ is 0. Figure 10 illustrates the cost calculation of the other three terms.

Figure 10(a) shows the calculation of $Resch2_3$. Suppose that (1) registers are classified into five sets, (2) reg_2, reg_5, and reg_8 are empty during the third control step, (3) v_6 has been assigned to register reg_1 during the previous control step, and (4) both v_4 and v_7 have some empty registers to assign to. We define the use ratio, $ratio_k$, of a classified register set set_k as the percentage of currently nonempty registers in set_k. The use ratio of register sets in Figure 10(a) are $ratio_1 = 5/8$, $ratio_2 = 3/4$, $ratio_3 = 3/4$, $ratio_4 = 1/2$, and $ratio_5 = 1/2$. The weight of reg_j is the total use ratio of register sets that contain reg_j. As v_6 is assigned to reg_1, the register released gain is $Total\_ratio(reg_1){=}ratio_1{+}ratio_2{+}ratio_5{=}15/8$. While as v_4 has not been assigned to register, its register released gain is one of $Total\_ratio(reg_2)$, $Total\_ratio(reg_5)$, and $Total\_ratio(reg_8)$, where reg_2, reg_5, and reg_8 are the registers to which v_4 can be assigned. As we aim at assigning a variable to a register belong to sets having more empty registers, we take the minimal value of $Total\_ratio(reg_2)$, $Total\_ratio(reg_5)$, and $Total\_ratio(reg_8)$, that is $Total\_ratio(reg_8){=}9/8$. The register allocation cost of the unassigned variable v_7 is estimated similarly and its value is 15/8. The final weight difference is therefore $Resch2_3 = 15/8 + 9/8 - 15/8 = 9/8$.

Figure 10(b) shows the calculation of $Resch3_3$. Suppose that there exist two pipeline conflicts in the scheduling of Figure 5(b). Eliminating the first one requires instruction 6 or 9 being moved down one control step, while eliminating the second one requires instruction 12 being moved down two control steps. We use the product of the number of instructions and the number of steps needed to be moved down as the cost of a pipeline conflict. The cost of this term is the sum of cost of all pipeline conflicts. The value of $Resch3_3$ is therefore $1 \times 1 + 1 \times 2 = 3$.

Figures 10(c) and (d) show the calculation of $Resch4_3$. Figure 10(c) shows the ALAP scheduling for instructions that have been scheduled after control step 2 in Figure 5(b). Among the rescheduling ranges obtained from these two schedulings, suppose there are possibly three parallelized instruction execution as shown in Figure 10(d). The probability that an instruction $inst_m$ be scheduled into a control step among the rescheduling ranges is $\frac{1}{sch^m_{alap} - sch^m_i + 1}$, where sch^m_{alap} and sch^m_i are the control steps into which $inst_m$ is scheduled by the two schedulings, respectively. And the probability of a parallelized instruction execution in a control step is the product of the probability of its constituent instructions being scheduled into this control step. In Figure 10(d), the probabilities of the parallelized instruction executions are all $1 \times 1/2 = 1/2$. The gain of rescheduling is the sum of probability of all possible parallelized instruction execution for all control steps. Therefore, the value of $Resch4_3$ is 3/2.

The notations and the formulations for cost calculation are defined below:

$TotalStep$	the total number of control steps before rescheduling
$TotalStep\_i$	the total number of control steps after rescheduling of $inst_i$

$$Resch1_i = TotalStep\_i - TotalStep$$

$In\_Var^a_i$	set of variables whose lifetimes end after the execution of $inst_i$ and have been assigned to registers
$In\_Var^u_i$	set of variables whose lifetimes end after the execution of $inst_i$ and have not yet been assigned to registers

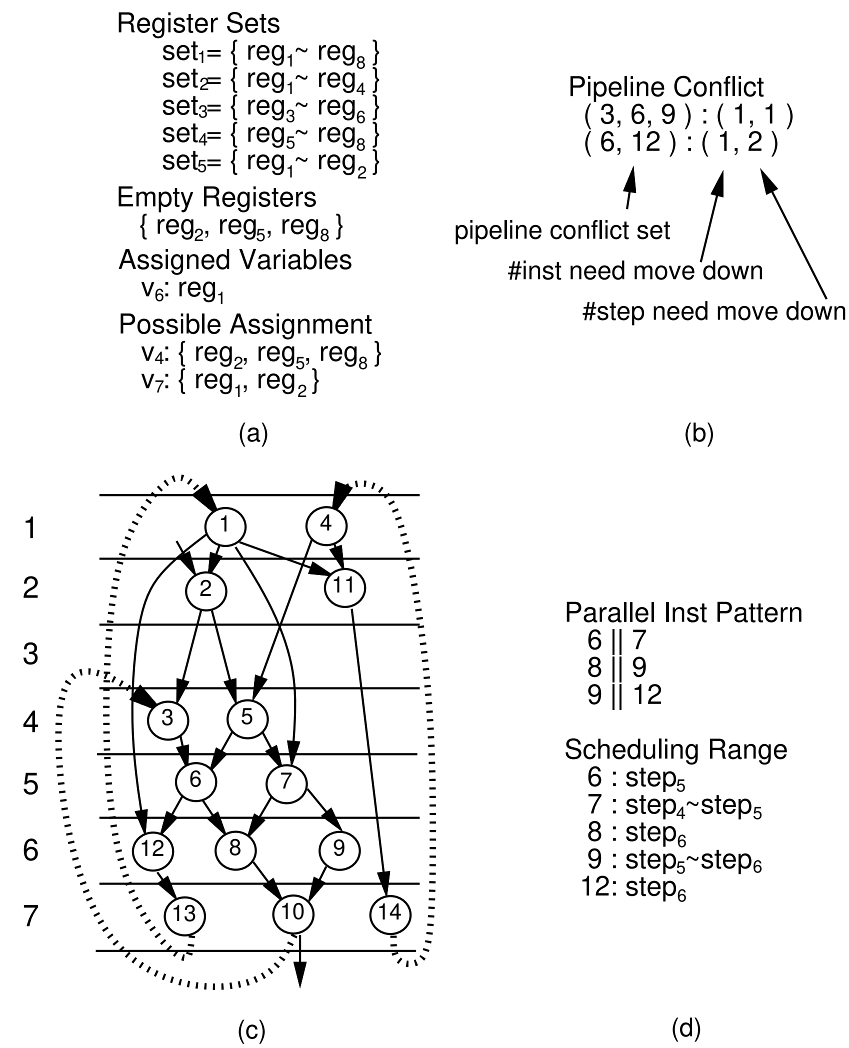


Fig. 10. An example of cost calculation for rescheduling: (a) Register classification of the target architecture, and allocation condition of variables of instruction 3; (b) The pipeline conflicts existed in Figure 5(b); (c) The ALAP scheduling for instructions that scheduled after step 2 in Figure 5(b); (d) Possible parallel execution of instructions among the rescheduling ranges obtained from (c) and Figure 5(b).

$Out\_Var_i^a$	set of variables produced by $inst_i$ and have been assigned to registers
$Out\_Var_i^u$	set of variables produced by $inst_i$ and have not yet been assigned to registers
R_Set_j	set of register sets that contain reg_j
R_In_i	set of registers allocated for variables in $In\_Var_i^a$
R_Out_i	set of registers allocated for variables in $Out\_Var_i^a$
R_Var_l	set of empty registers to which var_l can be assigned

$$Resch2_i = Resch2_i^1 + Resch2_i^2 - Resch2_i^3 - Resch2_i^4,$$

where

$$Resch2_i^1 = \sum_{reg_j \in R\_In_i} Total\_ratio(reg_j)$$

$$Resch2_i^2 = \sum_{var_l \in In\_Var_i^u} (min(Total\_ratio(reg_n)), \quad \forall reg_n \in R\_Var_l)$$

$$Resch2_i^3 = \sum_{reg_s \in R\_Out_i} Total\_ratio(reg_s)$$

$$Resch2_i^4 = \sum_{var_u \in Out\_Var_i^u} (min(Total\_ratio(reg_v)), \quad \forall reg_v \in R\_Var_u)$$

$$Total\_ratio(reg_j) = \sum_{set_k \in R\_Set_j} ratio_k$$

$\#pipe$	the number of pipeline stages
$TotalType$	the total number of pipeline conflict types
$\#delay_{j,k}^n$	the delay due to a $type_n$ constraint for pipeline execution between instructions of $Step_j$ and instructions of $Step_k$
$\#inst_{j,k}^n$	the minimal number of instructions in $Step_k$ that must be postponed in order to eliminate all $type_n$ conflicts between instructions of $Step_j$ and other instructions of $Step_k$

$$Resch3_i = \sum_{j=1}^{TotalStep} \sum_{k=(j+1)}^{(j+\#pipe-1)} \sum_{n=1}^{TotalType} (\#delay_{j,k}^n \times \#inst_{j,k}^n)$$

P_Set_j	set of possible parallelized instruction execution in $Step_j$

$$Resch4_i = \sum_{j=Step_{index}}^{TotalStep} \sum_{In\_P_j^k \in P\_Set_j} \prod_{inst_m \in In\_P_j^k} \frac{1}{sch_{alap}^m - sch_i^m + 1}$$

After the values of all issues for all candidate instructions in $Step_{index}$ are calculated, they are normalized to the range between 0 (-1 for $Resch2_i$) and 1 in order not to change the empirical parameters for each individual benchmark.

4.4 Cost Function of Register Allocation

This subsection describes the weight of assigning a variable var_j to a register reg_k in the bipartite matching graph. The weight $Weight_j^k$ of assigning var_j to reg_k is 0 if reg_k is unavailable during the lifetime of var_j, or that the assignment will violate the architectural constraint of instructions that access var_j in either current or previously scheduled steps. Otherwise, it is defined according to the following equation:

$$Weight_j^k = \beta_1 \times (1 - Alloc1_j^k) + \beta_2 \times Alloc2_j^k + \beta_3 \times (1 - Alloc3_j^k), \qquad (2)$$

where $\beta_1 \ldots \beta_3$ are empirical parameters. The cost function takes the following issues into account:

(1) Number of Register Moves ($Alloc1_j^k$):
A register allocation that inserts fewer register move instructions is preferred.

(2) Amount of Parallel Execution ($Alloc2_j^k$):
A register allocation is preferred if it can satisfy more instructions' register allocation constraint for parallel execution. The term is calculated in a way similar to that of the fourth term for rescheduling.

(3) Register Utilization ($Alloc3_j^k$):
It is preferred to assign a variable to a register such that all the classified register sets containing this register have more empty registers. This can balance the use ratio of register sets and prevent code spilling due to overuse of registers in a register set.

The notations and the formulations for cost calculation are defined below:

$Insert_j^k$ — the set of inserted instructions due to assigning var_j to reg_k

$loop\_count_n$ — the number of times that the inserted instruction $inst_n$ will be executed

$$Alloc1_j^k = \sum_{inst_n \in Insert_j^k} loop\_count_n$$

sch^r — the control step into which $inst_r$ is scheduled in the current scheduling

$P\_Alloc_{j,k}^p$ — set of possible parallelized instruction execution in $Step_p$, where for each $In\_P_p^q \in P\_Alloc_{j,k}^p$, at least one of its constituent instructions accessing var_j and the assignment of var_j to reg_k satisfies the architectural constraint for parallel execution

$$Alloc2_j^k = \sum_{p=Step_{index}}^{TotalStep} \sum_{In\_P_p^q \in P\_Alloc_{j,k}^p} \prod_{inst_r \in In\_P_p^q} \frac{1}{sch_{alap}^r - sch^r + 1}$$

$$Alloc3_j^k = \sum_{set_t \in R\_Set_k} ratio_t$$

After the values for all possible allocations are calculated, they are also normalized to the range between 0 and 1.

5. EXPERIMENTAL RESULTS

We have implemented the proposed simultaneous instruction scheduling and register allocation algorithm in a software system using the C programming language. Our system has been tested with a set of benchmark programs, compared with TI's TMS320C40 C compiler [TI 1991] and hand crafted assembly code in terms of the code efficiency targeted towards the TMS320C40 DSP processor. We use TI's TMS320C40 simulator [TI. 1991b] to simulate the codes and count the number of execution cycles. The simulator can model the pipeline in the architecture accurately.

Seven frequently used DSP algorithms in the C language are used as the benchmarks. The first four algorithms come from Reference [Embree and Kimble 1991]. They are a 35 point highpass FIR filter (fir), a 6th order IIR filter (iir), a 16-point discrete fourier transform (dft) algorithm, and a fast fourier transform (fft) algorithm. The next three algorithms come from Reference [Pitas 1993]. They are an edge detection algorithm for 32-by-32 images (edge), a 16-by-16 discrete consine transform (dct) algorithm, and a 16-by-16 wiener filter frequency response(wiener) algorithm.

As we have normalized the values of each term of the cost functions to the range between 0 and 1, we do not have to change the empirical parameters for each individual benchmark. And as we aim at paralleling as many instructions as possible and minimizing the number of pipeline delays, we define the empirical parameters according to the rule that the terms of considering instruction parallelism (α_4, β_2) and pipeline delays (α_3) have greater weight than the others. Unless a parameter is much larger or smaller than the others, the program is not sensitive to the empirical parameters. For all the benchmarks, the weight of the empirical parameters are $\alpha_1 : \alpha_2 : \alpha_3 : \alpha_4 = 3 : 2 : 4 : 4$, and $\beta_1 : \beta_2 : \beta_3 = 3 : 4 : 2$.

Table 1 shows the total execution cycles of the codes generated by our system, TI's TMS320C40 compiler, and the hand crafted assembly code, respectively. The first row shows the execution cycles of the codes generated by the TMS320C40 compiler with the highest level (Level 2) of optimization. The next thirteen rows show the execution cycles of the codes generated by our system when different terms of the cost functions are activated. The next row (HSA) shows the result of the hand crafted instruction scheduling and register allocation. In the last two rows, we show the results when instruction selection, data ordering in the memory, and address assignment for memory access are optimized by hand (HI), then instruction scheduling and register allocation are performed by our system and optimized by hand, respectively. The HI optimization removes unnecessary load/store instructions, and increases the number of instructions that can be parallelized. In the last column, we list the total execution cycles of all the benchmarks. The CPU time for each benchmark is no more than 1 second on a SUN SPARC-20 workstation.

Table 2 shows the improvement over TI's compiler. We have at least equal or better results than TI's compiler in all benchmarks. Table 3 and Table 4 show the number of execution cycles for parallelized instruction execution and pipeline delays, respectively. Our system generates more parallelized instruction execution and no more pipeline delays than TI's compiler did. We do not generate code spilling or insert additional register move instructions during rescheduling for all

Table 1. Comparison with TI's TMS320C40 compiler and hand crafted assembly code in terms of the number of execution cycles.

		fir	iir	dft	fft	edge	dct	wiener	overall
TI		364	61	12969	1536	25561	9604	7489	57584
$Theda$	$s_1 + A + r$	331	52	12713	1440	25561	8292	7233	55622
	$s_2 + A + r$	331	55	12713	1448	25561	8804	7489	56401
	$s_3 + A + r$	331	55	12473	1397	23821	8548	7249	53874
	$s_4 + A + r$	331	52	12233	1408	24691	8548	7233	54496
	$S + a_1 + r$	331	52	12473	1442	23821	8292	7233	53644
	$S + a_2 + r$	331	52	12473	1382	23821	8276	6993	53328
	$S + a_3 + r$	331	52	12729	1397	23821	8292	7249	53871
	$P + r$	331	52	12473	1423	24691	8564	7233	54767
	$P + s_2 + r$	331	52	12473	1408	24691	8548	7233	54736
	$P + a_3 + r$	331	52	12233	1408	24691	8564	7233	54512
	$P + s_2 + a_3 + r$	331	52	12233	1408	24691	8548	7233	54496
	$S + A$	331	52	12473	1365	23821	8292	6993	53327
	$S + A + r$	331	52	12233	1350	23821	8276	6993	53056
HSA		331	52	12233	1333	23821	8020	6993	52783
$HI + Theda$		298	49	11486	1263	19291	6420	6481	45288
$HI + HSA$		298	46	11486	1231	19291	5908	6481	44741

TI: TI's TMS320C40 compiler with the highest level of optimization.
$Theda$: Our compiler.
s_x: The xth term of the cost function for rescheduling is used.
a_x: The xth term of the cost function for allocation is used.
S: $s_1 + s_2 + s_3 + s_4$.
A: $a_1 + a_2 + a_3$.
P: $s_4 + a_2$.
r: The register reallocation optimization is used.
HI: Hand crafted instruction selection, data ordering, and address assignment optimization.
HSA: Hand crafted instruction scheduling and register allocation.

the benchmarks.

For the wiener benchmark in Table 3 and the fft benchmark in Table 4, the results of the fully optimized hand crafted assembly code ($HI + HSA$) do not get the optimal solution because some parallelized load/store instructions are unnecessary and removed. Still they have fewer execution cycles as shown in Table 1.

Although the last column of Table 2 shows that our system does not have much improvement over that of the TI's compiler in the overall result, however we have got the optimal solution for five benchmarks when all terms of the cost functions are activated in comparison with hand crafted instruction scheduling and register allocation (HSA). The two benchmarks that do not get the optimal solution are all because of more pipeline conflict delays as shown in Table 4.

When instruction selection, data ordering, and address assignment are optimized by hand, our system ($HI + Theda$) gets the optimal solution for four benchmarks in comparison with the fully optimized hand crafted assembly code ($HI + HSA$). Among the three benchmarks that we do not get the optimal solution, one (iir) is due to the less number of instruction parallelism, and the other two are due to both the less number of instruction parallelism and more pipeline conflict delays as shown in Table 3 and Table 4. From these two comparisons ($Theda$ vs. HSA, and $HI + Theda$ vs. $HI + HSA$), we demonstrate that our algorithm has gotten the

Table 2. Ratio of execution cycles of the codes generated by our system and hand crafted assembly code over those generated by TI's TMS320C40 compiler.

		fir	**iir**	**dft**	**fft**	**edge**	**dct**	**wiener**	overall
$\frac{Theda}{TI}$	$s_1 + A + r$	0.909	0.852	0.980	0.938	1.0	0.863	0.966	0.967
	$s_2 + A + r$	0.909	0.902	0.980	0.943	1.0	0.917	1.0	0.979
	$s_3 + A + r$	0.909	0.902	0.962	0.910	0.932	0.890	0.970	0.936
	$s_4 + A + r$	0.909	0.852	0.943	0.917	0.966	0.890	0.966	0.946
	$S + a_1 + r$	0.909	0.852	0.962	0.939	0.932	0.863	0.966	0.932
	$S + a_2 + r$	0.909	0.852	0.962	0.90	0.932	0.862	0.934	0.926
	$S + a_3 + r$	0.909	0.852	0.981	0.910	0.932	0.863	0.970	0.936
	$P + r$	0.909	0.852	0.962	0.926	0.966	0.892	0.966	0.951
	$P + s_2 + r$	0.909	0.852	0.962	0.917	0.966	0.890	0.966	0.951
	$P + a_3 + r$	0.909	0.852	0.943	0.917	0.966	0.890	0.966	0.947
	$P + s_2 + a_3 + r$	0.909	0.852	0.943	0.917	0.966	0.892	0.966	0.946
	$S + A$	0.909	0.852	0.962	0.889	0.932	0.863	0.934	0.926
	$S + A + r$	0.909	0.852	0.943	0.879	0.932	0.862	0.934	0.921
$\frac{HSA}{TI}$		0.909	0.852	0.943	0.868	0.932	0.835	0.934	0.917
$\frac{HI+Theda}{TI}$		0.819	0.803	0.886	0.822	0.755	0.668	0.865	0.786
$\frac{HI+HSA}{TI}$		0.819	0.754	0.886	0.801	0.755	0.615	0.865	0.777

Table 3. Comparison with TI's TMS320C40 compiler and hand crafted assembly code in terms of the number of parallelized instruction execution.

		fir	**iir**	**dft**	**fft**	**edge**	**dct**	**wiener**	overall
TI		0	0	0	17	0	256	256	529
$Theda$	$s_1 + A + r$	0	9	16	81	0	768	512	1386
	$s_2 + A + r$	0	6	16	49	0	768	256	1095
	$s_3 + A + r$	0	6	256	34	0	512	256	1064
	$s_4 + A + r$	0	9	496	81	0	784	512	1882
	$S + a_1 + r$	0	9	256	32	0	768	512	1577
	$S + a_2 + r$	0	9	256	81	0	784	512	1642
	$S + a_3 + r$	0	9	0	66	0	768	256	1099
	$P + r$	0	9	256	66	0	768	512	1611
	$P + s_2 + r$	0	9	256	81	0	784	512	1642
	$P + a_3 + r$	0	9	496	81	0	768	512	1866
	$P + s_2 + a_3 + r$	0	9	496	81	0	784	512	1882
	$S + A$	0	9	256	66	0	768	512	1611
	$S + A + r$	0	9	496	81	0	784	512	1882
HSA		0	9	496	81	0	784	512	1882
$HI + Theda$		33	12	497	96	900	1056	256	2850
$HI + HSA$		33	15	497	111	900	1312	256	3124

Table 4. Comparison with TI's TMS320C40 compiler and hand crafted assembly code in terms of the number of pipeline delays.

		fir	iir	dft	fft	edge	dct	wiener	overall
TI		33	0	720	139	1740	1056	240	3928
$Theda$	$s_1 + A + r$	0	0	480	107	1740	256	240	2823
	$s_2 + A + r$	0	0	480	83	1740	768	240	3311
	$s_3 + A + r$	0	0	480	17	0	256	0	753
	$s_4 + A + r$	0	0	480	75	870	528	240	2193
	$S + a_1 + r$	0	0	480	60	0	256	240	1036
	$S + a_2 + r$	0	0	480	49	0	256	0	785
	$S + a_3 + r$	0	0	480	49	0	256	0	785
	$P + r$	0	0	480	75	870	528	240	2193
	$P + s_2 + r$	0	0	480	75	870	528	240	2193
	$P + a_3 + r$	0	0	480	75	870	528	240	2193
	$P + s_2 + a_3 + r$	0	0	480	75	870	528	240	2193
	$S + A$	0	0	480	17	0	256	0	753
	$S + A + r$	0	0	480	17	0	256	0	753
HSA		0	0	480	0	0	0	0	480
$HI + Theda$		0	0	480	47	0	256	0	783
$HI + HSA$		0	0	480	30	0	0	0	510

optimal solution for most of the benchmarks in terms of the instruction scheduling and register allocation problems.

We also compare the effect of the cost functions and the register reallocation optimization. The first four rows of *Theda* compare the four issues of the cost function for rescheduling. Table 3 and Table 4 show that the issues of considering instruction parallelism (s_4) and pipeline delay (s_3) have the best results in terms of the number of instruction parallelism and pipeline delays for all benchmarks, respectively. Overall, considering instruction parallelism (s_4) has better results in most of the benchmarks as shown in Table 2.

The next three rows compare the effect of the three issues of the cost function for register allocation. The issue of considering instruction parallelism (a_2) also has the best results in terms of the number of instruction parallelism. As the pipeline delays are caused by poor scheduling, the allocation issues do not affect the scheduling directly. They almost have the same results except for the fft and the wiener filter benchmarks. Overall, the issue of considering instruction parallelism (a_2) has the best results in all the benchmarks.

In the above two comparisons, the issue of register utilization has the worst results in both rescheduling (s_2) and allocation (a_3) for most of the benchmarks. This is because that it does not increase the number of instruction parallelism nor reduce the number of pipeline delays directly. In fact, it is better to consider it together with the issues of instruction parallelism ($P : s_4 + a_2$) to avoid the condition of no empty register satisfying the architectural constraint of parallelized instructions. The next four rows of Table 3 show this effect. In addition to instruction parallelism ($P + r$), considering s_2 during rescheduling ($P + s_2 + r$) increases the number of instruction parallelism in two benchmarks (fft, dct). Considering it together in allocation ($P + a_3 + r$) also improve the results in two benchmarks (dft, fft). And considering it together in both rescheduling and allocation ($P + s_2 + a_3 + r$) improves the results in three benchmarks (dft, fft, dct). As there is no side effect of increasing the number of pipeline delays as shown in Table 4, it indeed reduce the total execution cycles.

Finally, the last two rows in *Theda* show the effect of the register reallocation optimization. With reallocation, we generate more parallelized instruction execution than that without reallocation in three benchmarks. When all terms of the cost functions are activated ($S + A + r$), it has the best results in term of the number of instruction parallelism, pipeline delays, and the total execution count for all the benchmarks among the thirteen rows of *Theda*.

6. CONCLUSION AND FUTURE WORK

We have presented an approach to the code generation problem for DSP processors with structural pipelining and heterogeneous registers. In addition to maximizing the parallelism and reducing register moves and code spilling, we emphasize on minimizing pipeline delays caused by simultaneous access to the registers and memories between instructions of different pipeline stages. By integrating instruction scheduling and register allocation (and re-allocation), we are able to deal with the interdependency between these traditionally separated subtasks. We have designed cost functions to reflect all practical considerations. Experimental results over a set of benchmark DSP kernel programs have shown that our proposed software system is able to generate more efficient codes in comparison with a state of the art commercial compiler. Although we only target to one processor in the experiment, our approach can be retargeted to other DSP processors or ASIPs with different register classification and structural pipelining.

In the future, we will automate the procedure of instruction selection, data ordering, and address assignment optimization (HI) to remove unnecessary load/store instructions, and increase the probability of parallelized instruction execution. And under the multiprocessor structure, taking into account the FIFO and circular buffer architecture for interface communication and interconnection network during tasks partitioning and code generation is also another future research.

1991. *TMS320 Floating-Point DSP Optimizing C Compiler*. Texas Instruments.

1991a. *TMS320C4x*. Texas Instruments.

1991b. *TMS320C4x C Source Debugger*. Texas Instruments.

Araujo, G. and Malik, S. 1995. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proc. 8th International Symposium on System Synthesis* (September 1995), pp. 36–41.

Araujo, G., Malik, S., and Lee, M. T.-C. 1996. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proc. 33rd Design Automation Conference* (June 1996), pp. 591–596.

Bradlee, D. G., Eggers, S. J., and Henry, R. R. 1991. Integrating register allocation and instruction scheduling for riscs. In *Proc. SIGPLAN Symposium on Compiler Construction* (June 1991), pp. 122–131.

Briggs, P., Cooper, K. D., Kennedy, K., and Torczon, L. 1989. Coloring heuristics for register allocation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (July 1989), pp. 275–284.

Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. 1991. Ptolemy: A platform for heterogeneous simulation and prototyping. In *Proc. European Simulation Conference* (1991), pp. 155–182.

CALLAHAN, D. AND KOBLENZ, B. 1991. Register allocation via hierarchical graph coloring. In *Register Allocation via Hierarchical Graph Coloring* (July 1991), pp. 192–202.

CHOW, F. AND HENNESSY, J. 1984. Register allocation by priority-based coloring. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1984), pp. 98–108.

DAVIDSON, S., LANDSKOV, D., SHRIVER, B. D., AND MALLETT, P. W. 1981. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers 30*, 7 (July), 460–477.

EMBREE, P. M. AND KIMBLE, B. 1991. *C Language Algorithms for Digital Signal Processing.* Prentice-Hall.

GIBBONS, P. B. AND MUCHNICK, S. S. 1986. Efficient instruction scheduling for a pipelined architecture. In *Proc. SIGPLAN Symposium on Compiler Construction* (June 1986), pp. 11–16.

GOODMAN, J. R. AND HSU, W.-C. 1988. Code scheduling and register allocation in large basic blocks. In *Proc. International Conference on Supercomputing* (July 1988), pp. 442–452.

GOOSSENS, G., LANNEER, D., PAUWELS, M., DEPUYDT, F., SCHOOFS, K., KIFLI, A., CORNERO, M., PETRONI, P., CATTHOOR, F., AND MAN, H. D. 1995. Integration of medium throughput signal processing algorithms on flexible instruction set architectures. *Journal on VLSI Signal Processing, special issue on "Synthesis for DSP" 9*, 1-2 (January), 49–65.

HENNESSY, J. L. AND GROSS, T. 1983. Postpass code optimization of pipeline constrains. *ACM Transactions on Programming Languages and Systems 5*, 3 (July), 422–448.

HUANG, C. Y., CHEN, Y. S., LIN, Y. L., AND HSU, Y. C. 1990. Data path allocation based on bipartite weighted matching. In *Proc. 27th Design Automation Conference* (June 1990), pp. 499–504.

HWU, W.-M. AND CHANG, P. P. 1988. Exploiting parallel microprocessor microarchitectures with a compiler code generator. In *Proc. 15th Annual International Symposium on Computer Architecture* (June 1988), pp. 45–53.

KIM, B. M. AND BARNWELL, T. P. 1990. Resource allocation and code generation for pointer based pipelined dsp multiprocessors. In *Proc. 1990 International Symposium on Circuits and Systems* (May 1990), pp. 2685–2688.

KOLSON, D. J., NICOLAU, A., DUTT, N., AND KENNEDY, K. 1995. Optimal register assignment to loops for embedded code generation. In *Proc. 8th International Symposium on System Synthesis* (September 1995), pp. 42–47.

KURDAHI, F. J. AND PARKER, A. C. 1987. Real: A program for register allocation. In *Proc. 24th Design Automation Conference* (June 1987), pp. 210–215.

L. J. HENDREN, E. A. 1992. *Register Allocation using Cyclic Interval Graphs: A New Approach to an Old Problem.*

LEE, T. F., WU, C. H., GAJSKI, D. D., AND LIN, Y. L. 1992. An effective methodology for functional pipelining. In *Proc. International Conference on Computer Aided Design* (November 1992), pp. 230–233.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1995. Code optimization technique for embedded dsp microprocessors. In *Proc. 32nd Design Automation Conference* (June 1995), pp. 599–604.

LIEM, C., MAY, T., AND PAULIN, P. 1994. Register assignment through resource classification for asip microcode generation. In *Proc. International Conference on Computer Aided Design* (November 1994), pp. 397–402.

PAPADIMITRIOU, C. H. AND STEIGLITZ, K. 1982. *Combinatorial Optimization.* Prentice-Hall.

PAULIN, P., LIEM, C., MAY, T., AND SUTARWALA, S. 1995. Dsp design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal on VLSI Signal Processing, special issue on "Synthesis for DSP" 9*, 1-2 (January), 23–47.

PINO, J., HA, S., LEE, E. A., AND BUCK, J. T. 1995. Software synthesis for dsp using ptolemy. *Journal on VLSI Signal Processing, special issue on "Synthesis for DSP" 9*, 1-2 (January), 7–21.

PINTER, S. 1993. Register allocation with instruction scheduling: a new approach. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (July 1993), pp. 248–257.

PITAS, I. 1993. *Digital Image Processing Algorithms.* Prentice-Hall.

RIMEY, K. AND HILFINGER, P. 1988. A compiler for application-specific signal processors. In *VLSI Signal Processing III* (1988), pp. 341–351.

WILSON, T., GREWAL, G., HALLEY, B., AND BANERJI, D. 1994. An integrated approach to retargetable code generation. In *Proc. 7th International Symposium on High Level Synthesis* (May 1994), pp. 70–75.