Spring 1999

A systematic integration of register allocation and instruction scheduling

Yukong Zhang
Louisiana Tech University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A SYSTEMATIC INTEGRATION OF REGISTER ALLOCATION AND INSTRUCTION SCHEDULING

Yukong Zhang, M.S.

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

May 1999

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

May 17, 1999
Date

We hereby recommend that the dissertation prepared under our supervision by Yukong Zhang

entitled A Systematic Integration of Register Allocation and Instruction Scheduling

be accepted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Lee Hyuk Lee
Supervisor of Thesis Research

Richard J. Greechie
Head of Department

Applied Computational Analysis and Modeling (ACAM)
Department

Recommendation concurred in:

Advisory Committee

Approved:

Director of Graduate Studies

Approved:

Director of Graduate School

Dean of the College

GS Form 13
2/97

ABSTRACT

In order to achieve high performance, processor architecture has become more and more complicated. As a result, compiler-time optimizations have become more and more important for the effective use of a complex processor. One of the promising compiler-time optimizations is the integration of register allocation and instruction scheduling based on register-reuse chains. In the previous approach, however, the generation of register-reuse chains was not completely systematic and consequently created many unnecessary dependencies that restrict instruction scheduling.

This research proposes a new register allocation technique based on a systematic generation of register-reuse chains. The first phase of the proposed technique is to generate register-reuse chains that are optimal in the sense that no additional dependencies are created. Thus, register allocation can be done without restricting instruction scheduling. For the case when the optimal register-reuse chains require more than available registers, the second phase reduces the number of required registers by merging the register-reuse chains. A heuristic is developed for the second phase in order to reduce the additional dependencies created by merging chains. The first step of the second phase is to derive a conflict graph in which each node corresponds to a register-reuse chain, while an edge represents where the corresponding two chains cannot be merged. Applying a graph-coloring algorithm to the conflict graph, the number of chains can be effectively reduced. The final step of the second phase is to run the 0-1 knapsack

algorithm to make the number of chains exactly the same as the number of available registers. The proposed register allocation is implemented in LCC (Local C Compiler). An instruction scheduler is also implemented in LCC and then integrated with the proposed register allocator. Evaluation results show that the proposed algorithm and heuristic effectively reduce the number of necessary registers.

TABLE OF CONTENTS

LIST OF FIGURES

Figures Page

ACKNOWLEDGEMENTS

At this moment, the author wishes to express his gratitude to all the people who contributed to the completion of this dissertation.

First of all, I would like to express my deepest appreciation and thanks to Dr. Hyuk Jae Lee, my research advisor, for his numerous hours helping me to condense my thoughts and for his invaluable guidance to carry me through difficult times throughout my dissertation research at Louisiana Tech University.

I would like to thank Dr. Richard Greechie, Dr. Barry Kurtz, Dr. Weizhong Dai, and Dr. Raja Nassar for taking time to read my dissertation and providing me with very helpful suggestions.

I also owe a special thank to Dr. Greechie for his countless assistance throughout my study in ACAM program at Louisiana Tech University.

I wish to thank Ms. Frances Welch at College of Engineering and Science for her kindness and assistance whenever needed during my graduate study at Louisiana Tech University.

Many thanks go to my fellow students, Sanqiang Li, Xiaorong Ma, and Danny Parker for their helpful suggestions to this research and friendship.

Finally, special thanks go to my wife Xiaochun and my daughter Geran whose understanding, support, and love encourage me to complete this dissertation.

CHAPTER ONE

INTRODUCTION

1.1 Statement of Problem

Embedded systems are application-specific systems that are designed with microprocessors. These systems are employed for applications other than general-purpose computing. Examples of these systems include cellular phones, automobile engine-control units, printers, fax machines, and set top-boxes etc. The major components of a typical embedded system are a programmable processor, a program ROM on which software is stored, and optionally application-specific hardware. A key characteristic of an embedded system is that the software is part of the system components. The software component of these systems is referred to as the *embedded software,* while the microprocessor is referred to as the *embedded processor* on which the software is executed.

Embedded systems have some unique characteristics compared to general-purpose computing systems. Due to their high-volume market demand, the manufacturing of embedded systems is very cost-sensitive. Due to time-to-market requirements, a short design cycle is desirable. In addition, many applications such as cellular phones are battery-driven, so the low power consumption requirement must be met.

With recent advancements in semiconductor processing techniques, the integration of all the system components on a single chip has become possible. For a

system that is composed of hardware and software, a hardware-software co-design approach has been used by the designers of the systems [20, 40, 44]. With this design methodology, the designers first determine which part of the functionality of the systems will be implemented in hardware and which part in software. Then the designed system will be simulated and evaluated with a co-design simulator. If the simulation results do not satisfy the design requirements -- such as power consumption, cost, and real-time constraints, etc. -- the designers may repartition the hardware and software of the system and then repeat simulation and evaluation of the new design until the design requirements and specifications are satisfied.

Given a fixed size of die on which all the components of a system are integrated, a certain amount of silicon area is dedicated to the program ROM, which is used to store the embedded software. Thus the program ROM area becomes limited. It is the designer's goal to generate high-density software code in order to fit the software code within the program ROM and to reduce software code size as much as possible, because the cost of a system increases non-linearly with the die size. In addition, many applications have strict real-time performance requirements. Producing high-performance code for embedded systems is a necessity.

In addition to code size and performance, there is another important constraint for embedded systems: power consumption. Generally, there are two main factors affecting the power consumption of embedded systems. The first factor is performance of the system. It is observed that the code that is executed faster consumes less power. The other factor is related to the instruction execution order of a given application code. An

optimal schedule of a sequence of instructions may reduce power consumption significantly.

Traditionally, in order to guarantee that the code size and performance requirements of embedded systems are satisfied, the software code usually is written manually in assembly languages. Although the assembly programming of small applications may be not relatively complicated, as the complexity of applications grows, manually programming in assembly languages becomes impractical, tedious, and error-prone. In addition, in order to meet short time-to-market cycle requirements, more efficient methods are desirable. Recently, most embedded software codes are written in a high-level language, such as C or C++, and use compiler technology to translate the high-level languages into assembly code. This is because programming in the high-level languages significantly reduces the cost and time of the software development. Furthermore, relatively less effort is required to maintain the code written in high-level languages. However, the code generated by traditional compiler techniques generally cannot satisfy the code size, performance, and power consumption requirements of embedded systems at the same time. The major reason is that traditional compiler optimization techniques classically focus on the code execution speed rather than code density and power consumption. Thus the new compiler-time optimization techniques become very important for code size, performance, and power consumption of embedded systems, although the traditional compiler optimization techniques are still effective for embedded systems.

Basically, the fundamental structure of a compiler can be classified into two parts:

- The *front-end*, which takes as input a code written in high-level languages and generates an intermediate representation of the input code, which is independent of the target machine.
- The *back-end*, which generates target-machine-dependent assembly code based on the intermediate representation. In particular, two important phases are performed in the back-end: instruction scheduling and register allocation. The phase of instruction scheduling determines instruction execution order and the register allocation phase determines the registers that will be used by each instruction.

To generate high-quality code for embedded systems in terms of code density, performance, and power consumption, this research will focus on these two compiler optimization techniques: instruction scheduling and register allocation.

1.2 Research Objectives

As mentioned in the previous section, instruction scheduling determines the execution order of each of the instructions of an application code. The instructions can be scheduled in favor of our optimization goals, such as efficient use of registers and power consumption, without affecting the correctness of the code execution. Register allocation determines which registers are used for each of the instructions. Since registers are the limited temporary storage resource in a processor, which is used to store the values of variables and temporary variables of instructions, efficient use of registers is of vital importance for high-quality code generation. These two compiler optimization techniques often affect each other. If register allocation is performed first, additional dependencies between instructions are introduced due to temporary sharing of registers. It limits the flexibility of the scheduling of instructions. If instruction scheduling is

performed first, it may create a schedule that demands more registers than available. Consequently, the benefit from instruction scheduling will be limited. The objectives of this thesis can be summarized as follows:

- Design an algorithm/heuristic to integrate instruction scheduling and register allocation. This algorithm/heuristic will minimize the number of needed registers and additional constraints for instruction scheduler due to register allocation.
- Apply the algorithm/heuristic to compiler code generation for embedded systems aiming to meet requirements for code size, high performance, and power reduction.
- Evaluate the effectiveness of the proposed algorithm.

1.3 Research Methodology

In this research, ARM7T (Advanced RISC Machines), one of the ARM series of processors, has been chosen as the target processor. The characteristics of ARM processors are high performance, low cost, and low power consumption. Each ARM7T instruction is 32-bit long. The Local C Compiler (LCC), originally developed by Fraser and Hanson [15], is used as the compiler frame work tool with which the compiler optimization techniques developed in this research will be incorporated. To achieve the objectives of this research, a data dependence analyzer and an instruction scheduler are developed first. A new register allocator is developed based on a systematic generation of register-reuse chains. Based on the analysis of the interaction between instruction scheduling and register allocation, an algorithm/heuristic that integrates instruction scheduling and register allocation is developed.

1.4 Outline of the Dissertation

This dissertation is divided into six chapters. Chapter 1 explains the research objectives and the methodology used to achieve these objectives. Chapter 2 gives the background related to this research including previous research on instruction scheduling and register allocation. This chapter also briefly describes ARM series architecture and LCC structure.

Chapter 3 presents the development of the data dependence analysis and instruction scheduler. Chapter 4 describes the integration algorithms of instruction scheduling and register allocation. It includes register-reuse chain generation, register-reuse chain merging, and merging criteria.

Chapter 5 presents a systematic approach to merge register-reuse chains that generated by the algorithms described in Chapter 4 when the number of chains is greater than the number of available registers.

Chapter 6 summarizes the research.

CHAPTER TWO

BACKGROUND

Chapter 2 provides the background on instruction scheduling and register allocation as well as the integration of both techniques. This chapter also gives a brief description of ARM architecture and Local C Compiler (LCC) structure.

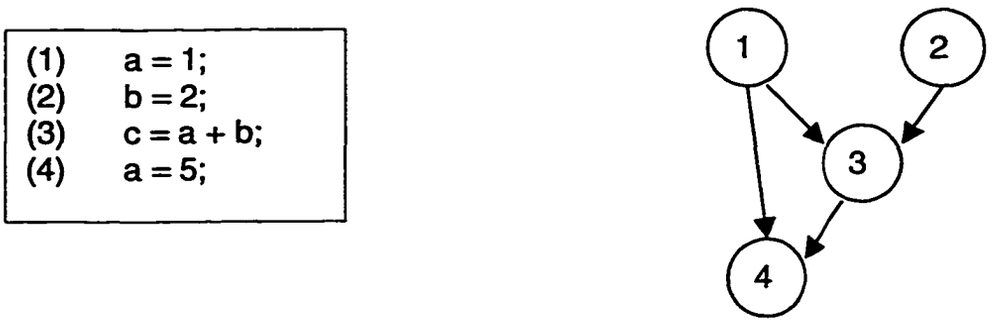
2.1 Compiler Optimization Techniques

Instruction scheduling and register allocation are very important compiler optimization techniques for embedded systems. This section is intended to give background descriptions on these techniques through some simple examples. Also it will explain how we can benefit from instruction scheduling. Before going through instruction scheduling, we first describe data dependence analysis, upon which the instruction scheduling is based.

2.1.1 Data Dependence Analysis

Data dependence analysis identifies the data dependence relationship between instructions and the constraints with which instruction scheduling must comply. The data dependencies between instructions fall into three categories: true dependence, anti-dependence, and output dependence. The alternative terminologies for these three types of data dependencies are called read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW), respectively. The data dependencies between instructions can

be visually expressed in directed acyclic graphs (DAGs), commonly called data dependence graphs. The following example in Figure 2.1 shows a piece of C code and its corresponding data dependence graph.



(a) Example C code (b) Data dependence graph

Figure 2.1 An example of data dependence graph

In the data dependence graph shown in Figure 2.1 (b), each vertex or node of the graph represents a statement. The number in a vertex represents the statement number. Each edge represents a dependence relation between two statements. Each edge could be either one of three types of dependencies. For instance, the edge $E(1, 3)$ represents a true data dependence (RAW) between statements 1 and 3, which indicates that statement 3 uses as input the value of variable a obtained from statement 1. The edge $E(3, 4)$ represents an anti-dependence (WAR) between statements 3 and 4. The value of variable a in statement 3 is read as input and updated in the following statement 4. The edge $E(1, 4)$ represents as an output dependence (WAW) between statements 1 and 4. The value of the variable a is obtained to be 1 in statement 1, and then updated to be 5 in statement 4. Whenever there is data dependence between two statements, a switch of the execution

order of these two instructions causes incorrect execution results. In other words, the execution order of two instructions must strictly comply with their data dependence relationship. Otherwise, the execution results will be wrong. For example, if we swap the execution order of statements 1 and 4, the resulting value of the variable *c* in statement 3 will be 7 instead of the correct value of 3.

2.1.2 Instruction Scheduling

Instruction scheduling is a compiler-time process to determine the execution order of a sequence of instructions, which is performed by a scheduler (part of a compiler) based on data dependence analysis. Given a sequence of instructions, the execution order does not have to be the same as the order in which the programmer writes instructions originally. However, any execution order of instructions scheduled by the scheduler must conform to the data dependence between instructions. For example in Figure 2.1, based on data dependence analysis, it is found that there are two valid possible execution orders of four statements. They may be scheduled either in the order {1, 2, 3, 4} or {2, 1, 3, 4}. Any other execution orders will violate the data dependence in above example.

A scheduler is usually developed to schedule instructions purposely in support of the programmer's optimization goals. For example, if our optimization goal is to exploit instruction parallelism to reduce total instruction execution time, the scheduler identifies that statements 1 and 2 can be executed in parallel because there is no data dependence between these two statements in the above example. The following execution order is scheduled:

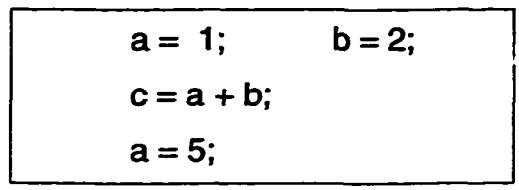


Figure 2.2 An example of paralleled code

The execution time is reduced to 3 from 4, assuming that the execution time of each statement takes one unit.

2.1.3 Register Allocation

Registers are temporary storage elements in a processor. All the operations occur in registers. If a variable is not stored in a register, it is loaded from or stored in main memory whenever it is accessed. If the number of registers is less than the number of variables, the register allocator of a compiler determines which variables are stored in registers.

A register allocation example is illustrated in Figure 2.3. Assume that there are two registers available, that variables *j* and *m* are allocated in registers, and variables *i* and *k* are not allocated in registers. The left part of the figure is an example of C code and the right part is the corresponding instructions to be executed by the processor. Since variable *i* is not allocated in register, after the assignment instruction $i = 1$, the value of variable *i* will be stored in main memory, so an extra store instruction is needed. When instruction $k = i + j$ is executed, since variable *i* is not allocated in register, a load instruction is needed to load the value of variable *i* from main memory. For the same reason, the instruction store *k* is needed to store the value of variable *k* back in main memory.

If there are three registers available, assuming that only variable *k* is not allocated in register, the corresponding instructions are shown in the lower part of the figure. Since variable *k* is not allocated in register, a memory access instruction is needed to store the value of variable *k* in main memory after instruction $k = i + j$ is finished. As a result, the number of the instructions associated with main memory access in this case is reduced to 1 from 3 in the previous case. As a result, code size and execution time are reduced.

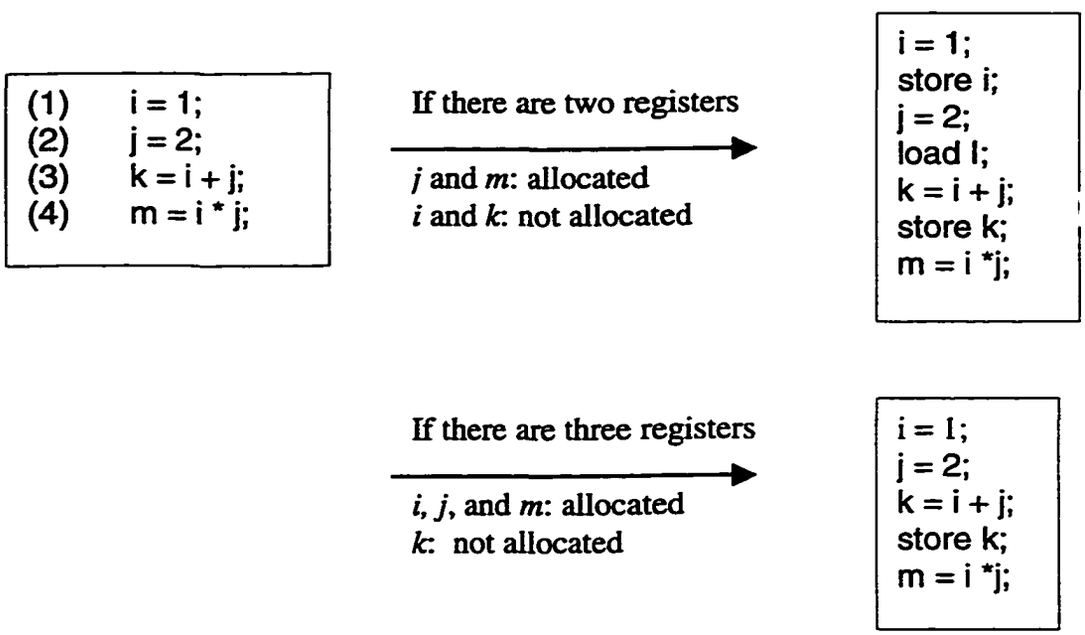


Figure 2.3 Example for register allocation

2.1.4 Live Range Analysis

Like data dependence analysis on which instruction scheduling is based, live range analysis is the basis of register allocation. Each variable in a program has a live range. The live range of a variable is the range from its definition to its last use, that is,

the range from the point where the variable becomes live to its dead point. For example in Figure 2.4, the live range of variable *h* is the range from statement 1 to statement 4, which is illustrated with an arrow below the variable name. If two variables have nonoverlapped live ranges, then they can share the same register. In other words, the second variable can safely reuse the register allocated to the first variable only when two variables have nonoverlapped live ranges. This is because a register is required to exclusively hold a live variable's value until that variable is "dead" once the register is allocated to that variable. Otherwise, the first variable's value stored in the register will be garbled by another variable that reuses the register allocated to the first variable. Consequently, wrong execution due to improper register allocation will result.

The minimum number of registers necessary can be obtained based on the live range analysis of a sequence of instruction. In the above example, variables *i* and *k* can share the same register because they have nonoverlapped live ranges. Variable *j* needs another register, and variable *h* needs another register. Thus at least three registers are needed in this example.

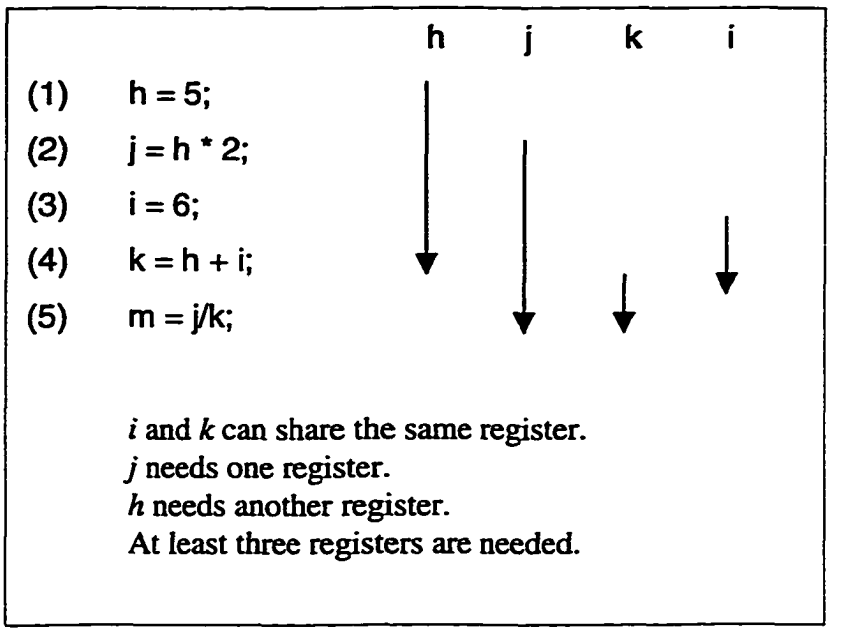


Figure 2.4 Live range analysis of variables

Obviously, the live range of a variable depends on the execution order of instructions. In other words, instruction scheduling affects register allocation. This is illustrated in Figure 2.5, which shows a different schedule for the instructions in Figure 2.4. It is easy to verify that this scheduling is a legal schedule based on the data dependence graph shown in Figure 2.5. With this schedule, variables *h* and *j* have nonoverlapped live ranges, so they can share the same register. For the same reason, variables *k* and *i* can share another register. As a result, two registers are needed, while the previous scheduling requires three registers. Thus the latter schedule saves one register compared with the original schedule.

From the above analysis, it is seen that instruction scheduling plays a very important role in the exploitation of instruction level parallelism and register allocation. In addition, instruction scheduling technique is very important to power reduction for embedded systems.

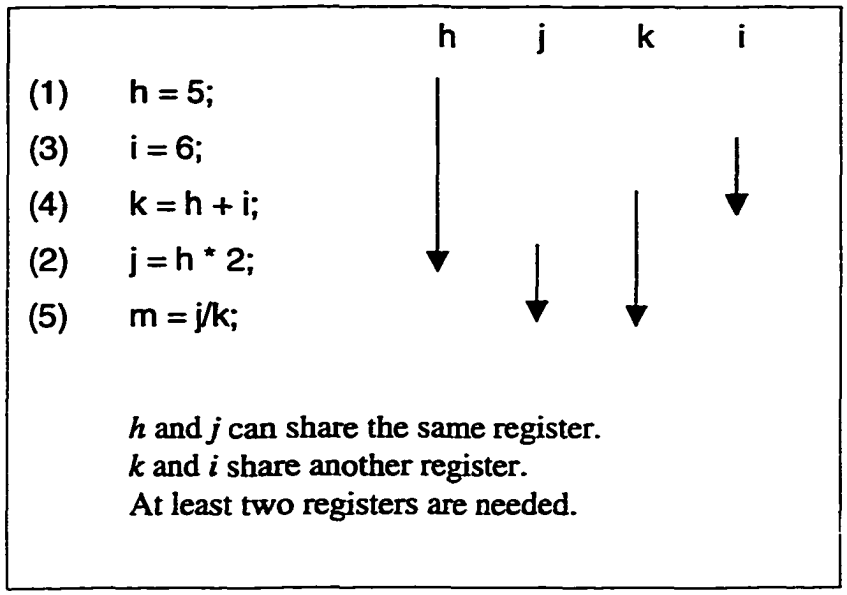


Figure 2.5 Effect of instruction scheduling on register allocation

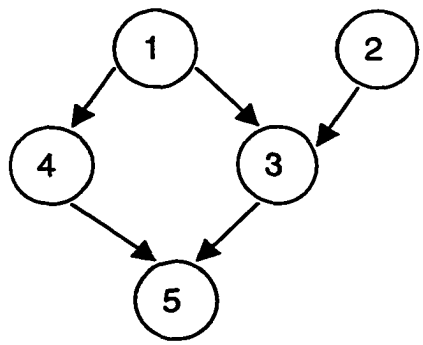


Figure 2.6 Data dependence graph

Instruction scheduling and register allocation have been two very important optimization phases not only for the compilers targeted on embedded systems but also for other modern compilers. Traditionally, one phase is performed before another phase,

which is called a phase ordering approach. In recent years it has been generally recognized that the separation between the instruction scheduling and register allocation phases leads to poor optimization for cases that are not suited to the specific phase ordering selected by the compiler [1, 31]. For instance, consider the first phase ordering approach, that is, instruction scheduling followed by register allocation. This phase ordering gives priority to instruction scheduling. It may be good for exploiting instruction-level parallelism. However, based on the previous example, it can be seen that the first schedule in Figure 2.4 stretches out some variables' live ranges compared to the second schedule in Figure 2.5. As a result, the required number of registers in the first schedule is 3, while the second schedule requires 2. If there are two registers available, one variable's value has to be spilled into main memory. Thus the first schedule relatively increases register pressure.

The second ordering approach is to perform register allocation before instruction scheduling. This approach gives priority to optimizing usage of registers, especially for the processors with a small number of registers. It may cause no spills or minimum spills. However, additional dependencies between instructions may be generated due to extra register dependencies. As a result, this generates additional constraints for instruction scheduling, thus limiting the flexibility of instruction scheduling.

It is desirable to integrate these two phases into a single phase to minimize constraints upon each other. Unfortunately, the integration of instruction scheduling and register allocation is a *NP*-Complete problem. Thus this research attempts to develop an effective heuristic algorithm for the integration problem. The register-reuse chain

generation and merging algorithms are developed primarily to minimize the constraints for instruction scheduling when optimizing of register allocation.

2.2 ARM7T (Advanced RISC Machines) Processor

This research uses the ARM7T microprocessor, one of a series of ARM processors developed by Advanced RISC Machines Ltd., as the target processor. ARM designs and licenses high-performance, low-cost, power efficient RISC microprocessors and related technology. ARM intends to establish its architecture as the standard for embedded RISC processors for use in a wide range of high volume applications in the embedded, portable, and consumer multimedia markets. The application examples of ARM processors include [41]:

- Portable: digital cellular phones, pagers and personal organizers.
- Embedded: modem, hard disc drivers, printers and automotive applications.
- Consumer multimedia: sound system, games, set-top box.

ARM7T series processors are the company's most widely licensed processors. The CPU cores of AMR7T are small, fast, low-power, 32-bit RISC processors that are primarily used in portable telecommunications. It has the ability to combine the ARM instruction set with the THUMB extension to reduce memory size and system cost. The THUMB extension delivers 32-bit RISC performance at a 16-bit system cost.

2.3 Local C Compiler (LCC)

Local C Compiler (LCC) is a retargetable compiler for ANSI C, developed by Fraser and Hanson [15]. It has been used to compile production programs since 1988. As a retargetable compiler, LCC has multiple target machines. The advantages of

retargetable compilers are that machine-specific compiler parts are isolated in modules that are readily applied to the user-desired target machines. For example, during the hardware-software co-design stage of an embedded system, the target architecture may be changed to achieve a better ratio of performance to cost. There is no need to rewrite the software compiler each time when some features are added or removed from the current architecture.

To be retargetable, LCC is organized into two major parts shown in Figure 2.7. The first part is called the *front end* that consists of a lexical analyzer and a parser. The lexical analyzer reads program source text and produce tokens. The parser takes a stream of tokens passed from the lexical analyzer and checks whether they conform to the syntax of the language. Then an intermediate representation of the source program, called directed acyclic graphs (DAGs), is generated. The intermediate representation generally is language-independent and target machine-independent. Within the front end, some target-independent optimizations are performed, such as common sub-expression elimination. The second part of LCC is called the *back end.* It takes as input the intermediate representation of source program, or DAGs, that are passed from the front end and translates them into target-dependent assembly code. When the target architecture is changed or a new architecture is considered, only the back end part of the compiler needs to be rewritten.

The optimization parts of LCC like instruction scheduling and register allocation originally distributed by Fraser and Hanson are primitive and simple. Based on their own needs, users can replace them and add their own optimization parts.

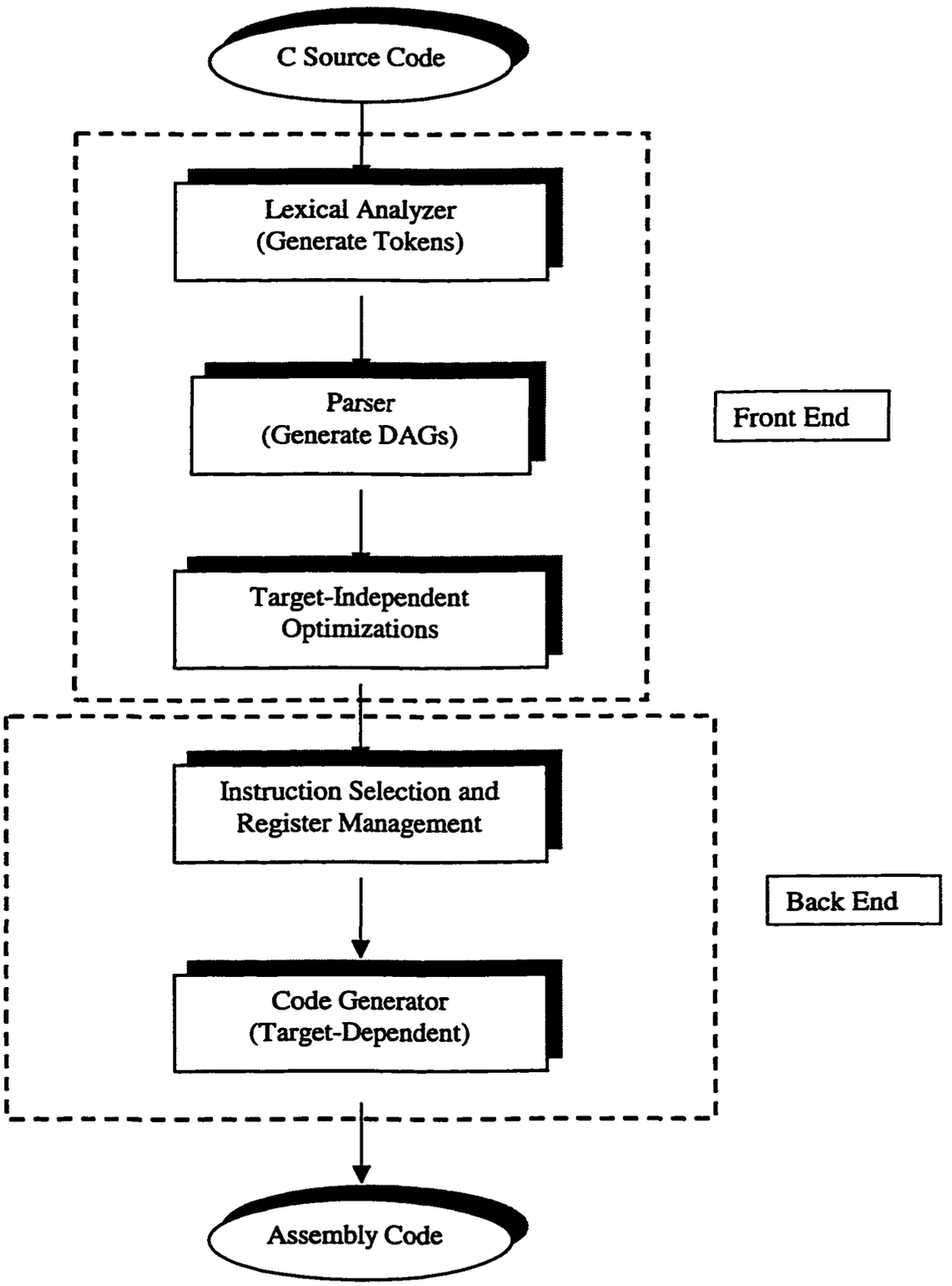


Figure 2.7 Structure of Local C Compiler (LCC)

CHAPTER THREE

DEPENDENCE ANALYSIS AND INSTRUCTION SCHEDULING

This chapter presents the dependence analysis and the instruction scheduler. In Section 1, the structure of optimizing Local C Compiler (LCC) is described. In Section 2, the data dependence analysis, the basis of the instruction scheduler, is represented. Section 3 describes the instruction scheduler developed in this research.

3.1 Structure of Optimizing LCC Compiler

Figure 3.1 shows the overall structure of the optimizing compiler developed in this research. It consists of three main phases. In the first phase, registers are allocated to variables of a program. This register allocation is optimized by each basic block. The next phase performs instruction scheduling. Dependencies between instructions are analyzed for the correct and most efficient instruction scheduling. The dependence analysis and instruction scheduling are also performed by each basic block. The last phase is the allocation of temporary registers. For this phase, the existing LCC variable register allocator is used. The variable register allocation is the main subject of Chapter 4 and Chapter 5. Therefore, detailed explanation is given in those two chapters. In this section, the implementation of dependence analysis and instruction scheduling in the LCC is explained and issues for the increase of efficiency are discussed.

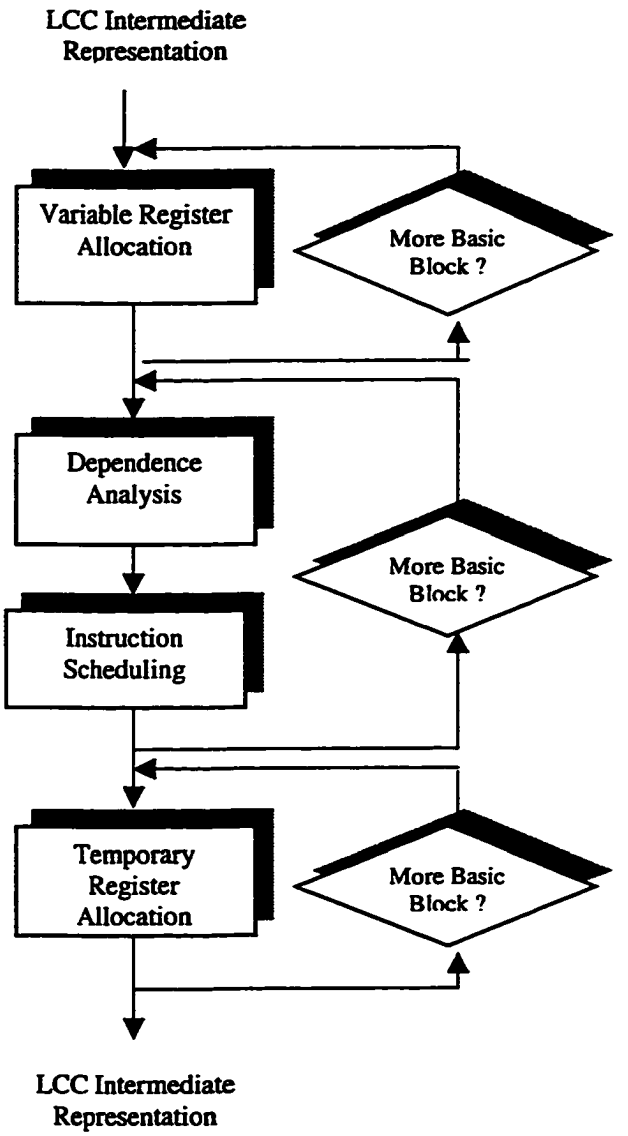
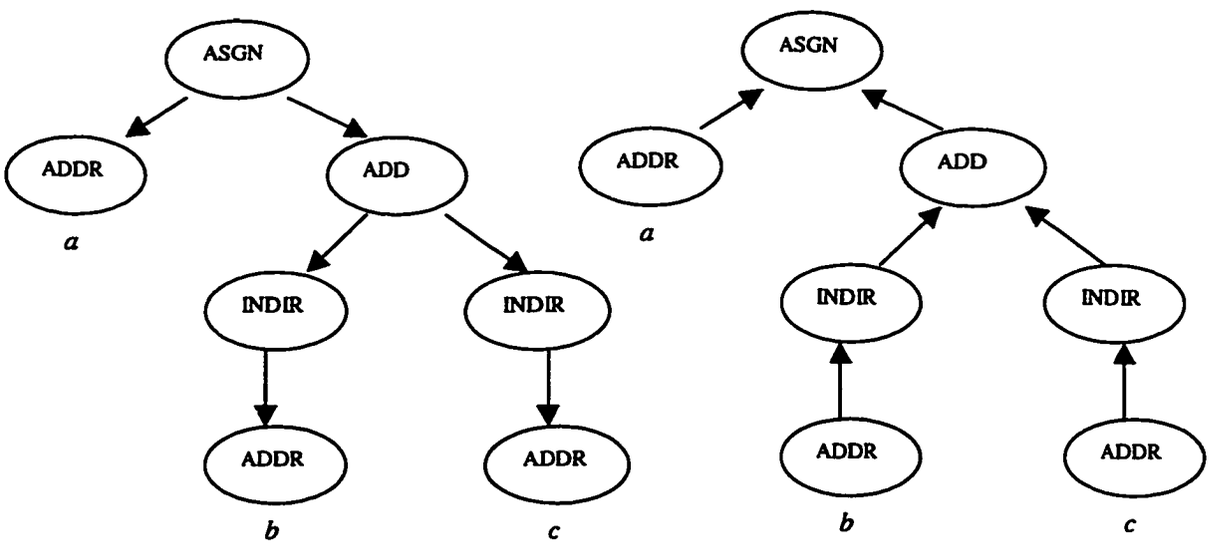


Figure 3.1 The structure of the new optimizing compiler

3.2 Dependence Analysis

3.2.1 True Dependence

Dependence represents the relationship in which a correct computation of one node depends on the result of the computation of the dependent node. For example, consider the Abstract Syntax Tree (AST) shown in Figure 3.2 (a). All nodes depend on their kids. For example, the ASGN node depends on the computation result of ADD and the address node (ADDR *a*). On the other hand, the node ADD depends on its kids, INDIR (ADDR *b*) and INDIR (ADDR *c*). Figure 3.2 (b) shows the corresponding dependence graph. The arrows in this graph show the dependence relationship. The target of the arrow depends on the source of the arrow.



(a) AST for $a = b + c$ (b) Dependence between nodes

Figure 3.2 A single AST and the corresponding dependence graph

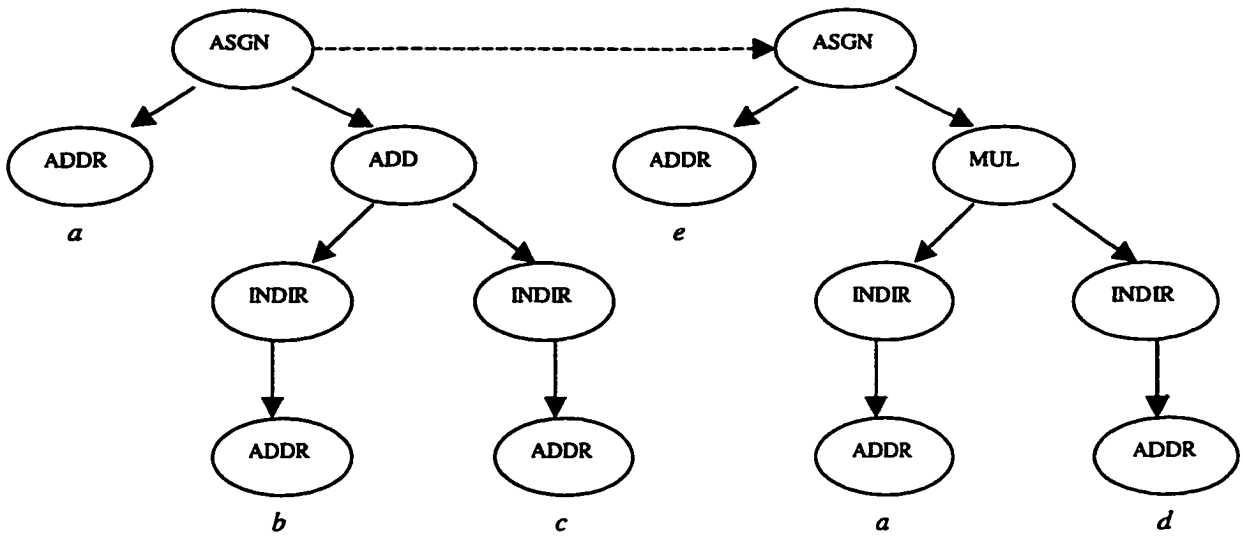
Now that the dependence graph for a single AST is explained, consider the dependence graph for multiple ASTs. Consider the ASTs shown in Figure 3.3 (a). There

are two ASTs, which represent statements $a = c + d$ and $e = a + d$, respectively. Note that the second statement depends on the first statement. In order to represent the dependence between two statements, a new edge is created from the ASGN node of the first AST to the corresponding ADDR (a) node in the second AST. This edge is illustrated with a thick line in Figure 3.3 (b).

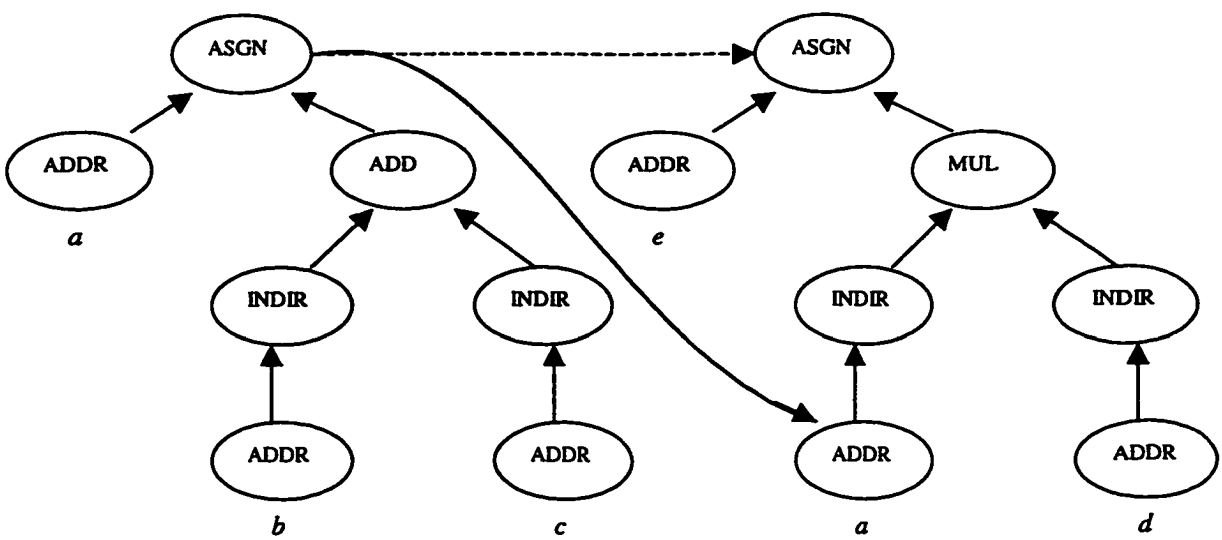
3.2.2 Anti-Dependence

Anti-dependence represents the relationship in which one statement depends on the other statement because the statement stores a value into the same memory location as the dependent statement loading a value. If the storing statement is executed earlier, before the loading statement accesses the value, a wrong value can be loaded. Consequently, the computation result can be wrong.

Figure 3.4 (a) shows a case when anti-dependence occurs. There are two ASTs representing two statements, $a = b + c$ and $c = a * d$, respectively. Note that the second INDIR node in the first AST reads data from c, while the ASGN node in the second AST stores data into c. In this case, an edge is created from the INDIR(ADDR c) node in the first AST to the ASGN(ADDR c) node in the second AST. This edge is shown in the thick line in Figure 3.4 (b).

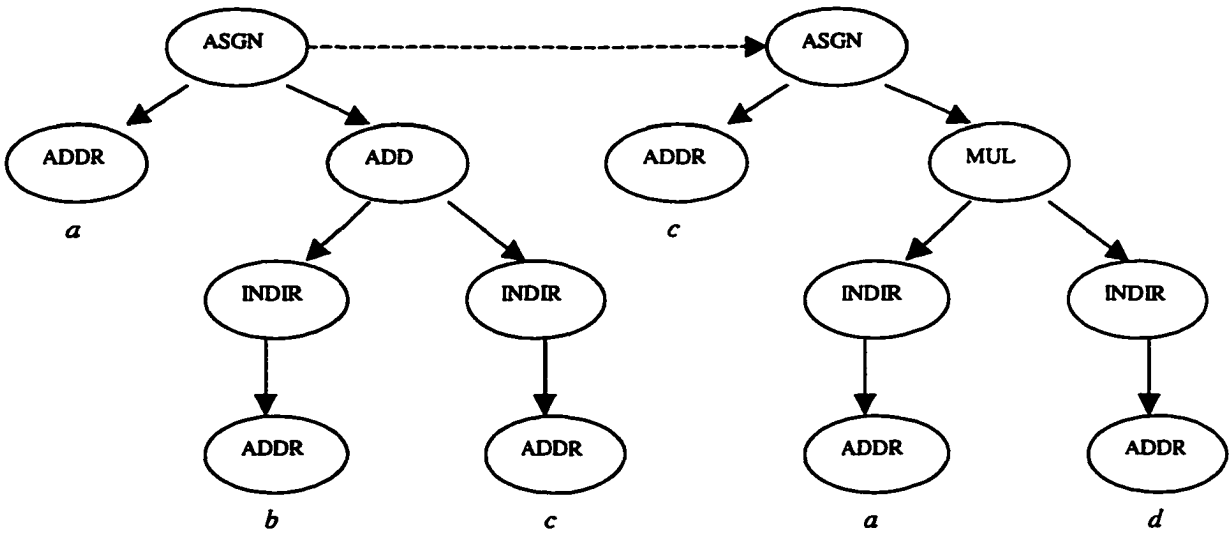


(a) ASTs for $a = b + c$ and $e = a * d$

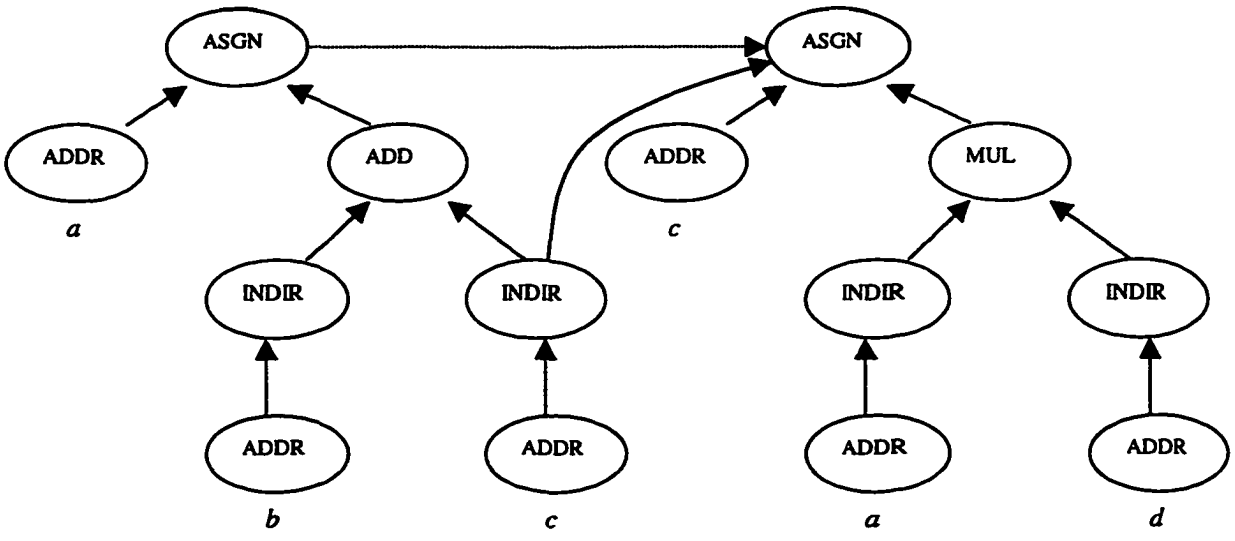


(b) Dependence graph

Figure 3.3 Multiple ASTs and the dependence graph



(a) ASTs for $a = b + c$ and $c = a * d$

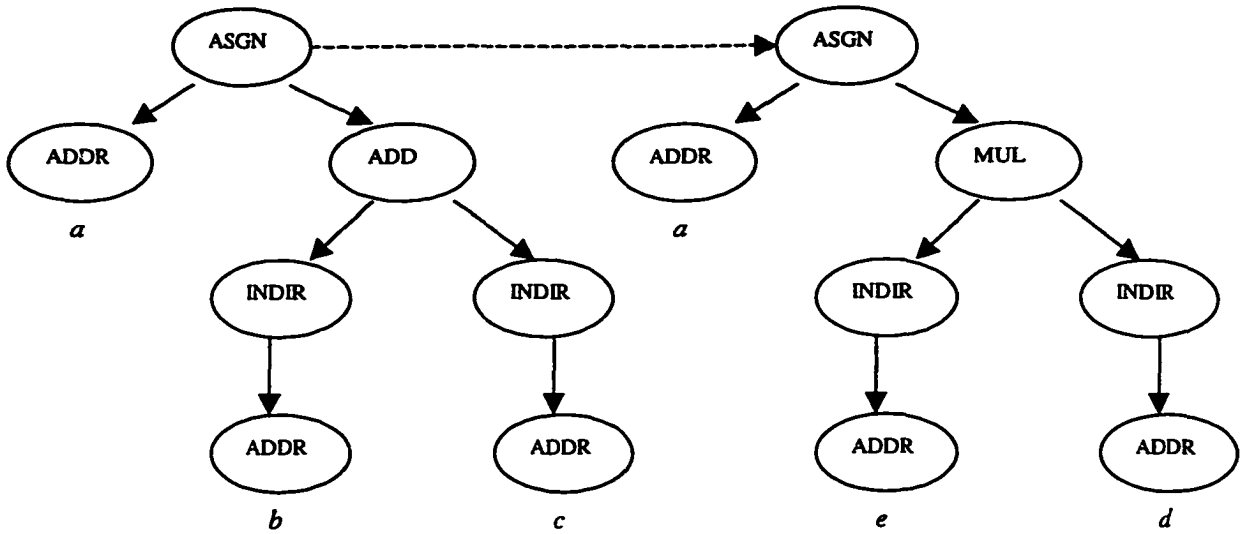


(b) Anti-dependence graph

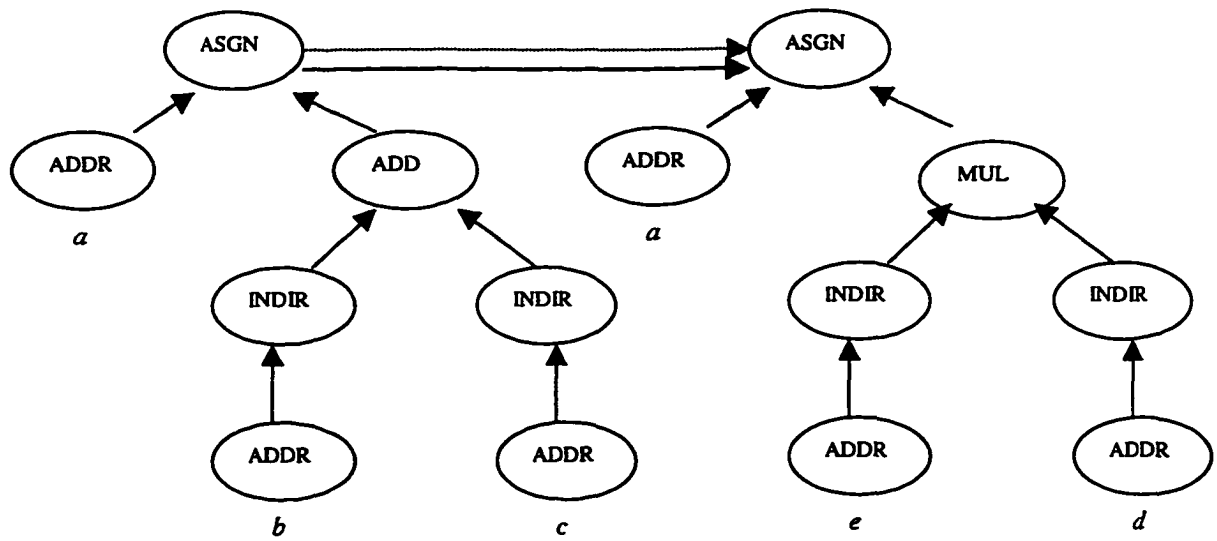
Figure 3.4 Multiple ASTs and the anti-dependence graph

3.2.3 Output Dependence

Output dependence represents the dependent relationship in which two statements store data into the same memory location. In order to store the correct value in the location, the order of the two stores must be preserved. Figure 3.5 (a) shows the case when output dependence occurs. There are two statements, $a = b + c$ and $a = e * d$. Note that both statements store data into a. In order to prevent the reordering of the two assignments, an edge is created between the two ASGN nodes. In Figure 3.5 (b), the thick line corresponds to the output dependence.



(a) ASTs for $a = b + c$ and $a = e * d$



(b) Output dependence graph

Figure 3.5 Multiple ASTs and the output dependence graph

3.2.4 Adjustment of Dependencies

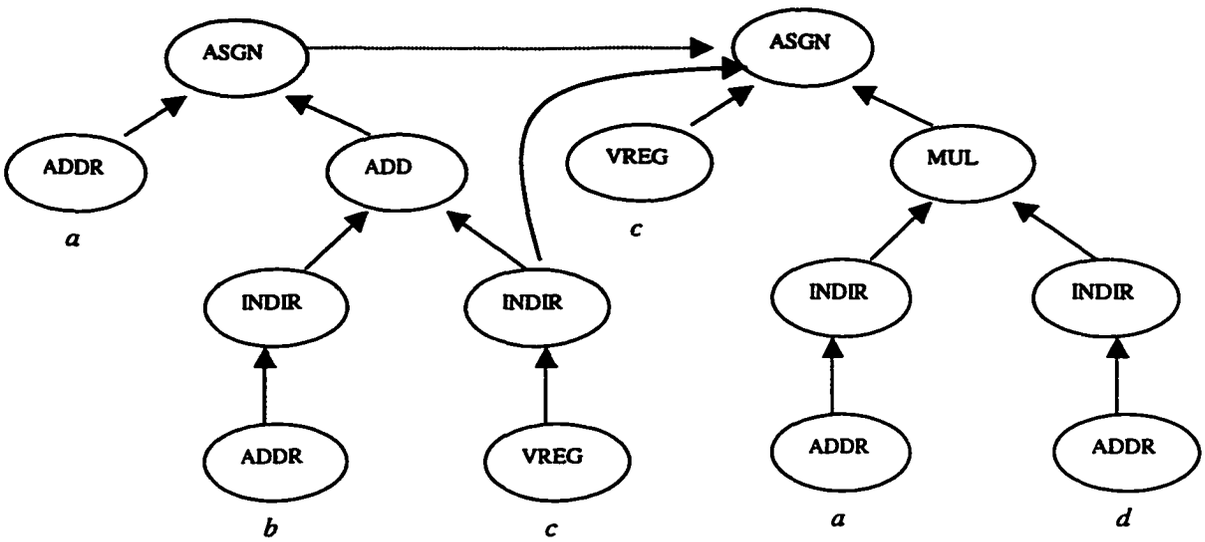
Register allocation often changes the output assembly code depending on which variables are assigned to registers. For example, Node (ASGN (ADDR symbol)) generates STORE instruction. However, if a register is allocated to the symbol, a STORE instruction is not necessary because the value is stored in the register. Therefore, after register allocation, the code needs to be labeled again so that a new code generation rule is assigned to each node. The new rule determines whether an ASGN node needs to generate STORE instruction or not. In addition, the new rule decides whether an INDIR node needs to generate a LOAD instruction or not.

Once code generation rule is changed, dependencies need to be adjusted. Consider the example of anti-dependence shown in Figure 3.4 again. There is an anti-dependence from (INDIR (ADDR *c*)) in the first tree to (ASGN (ADDR *c*)) in the second tree. Note that node (INDIR (ADDR *c*)) in the first tree generates assembly instruction `LOAD Rt, addr(c)`, and node (ASGN (ADDR *c*)) generates assembly instruction `STORE Rt, addr(c)`. Here, `Rt` represents a temporary register, and `addr(c)` represents the address of variable *c* in main memory. The anti-dependence guarantees that instruction `STORE Rt, addr(c)` is scheduled later than instruction `LOAD Rt, addr(c)`. Therefore, the correct sequence of instructions is generated.

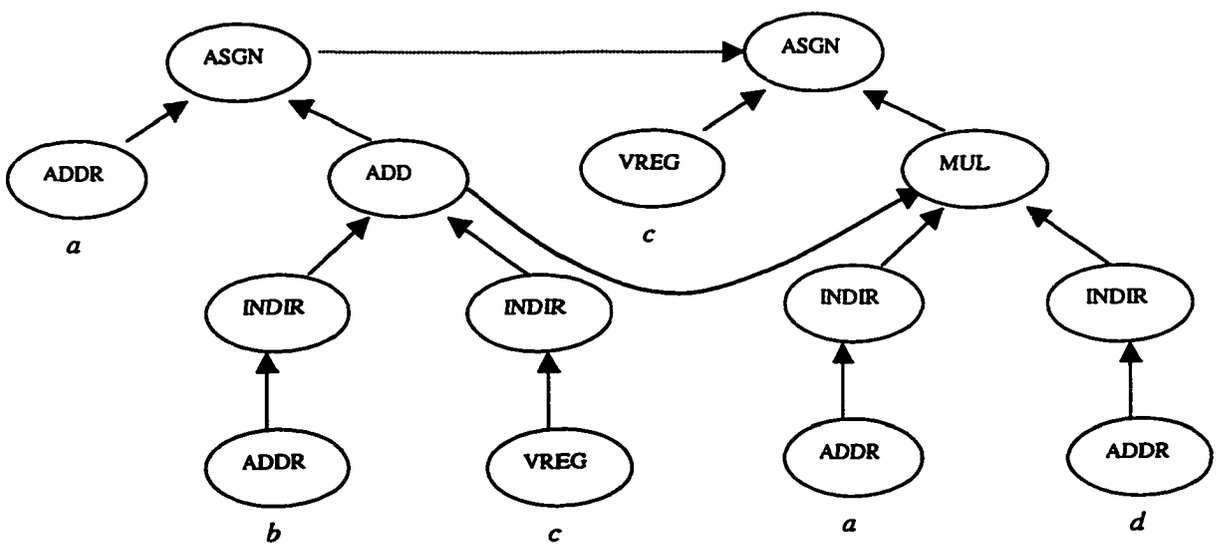
Suppose that variable *c* is assigned a register. Then, the tree is changed as shown in Figure 3.6 (a). Note that node (ADDR *c*) is changed to (VREG *c*) that represents a register is allocated to *c*. In this case, node (ASGN (VREG *c*)) does not generate an assembly instruction because variable *c* is now stored in a register. Its kid, node MUL, performs the actual assignment operation. Similarly, node (INDIR (VREG *c*)) does not generate an assembly instruction, and its kid, node ADD, performs the actual loading

operation. Therefore, an anti-dependence needs to be adjusted such that node MUL is scheduled after node (INDIR (VREG *c*)). The resulting dependence graph is shown in Figure 3.6 (b). The new edge from ADD in the first tree to MUL in the second tree is created.

Register allocation also requires the adjustment of output dependence. Consider the dependence graph shown in Figure 3.5 again. There is output dependence from node ASGN in the first tree to node ASGN in the second tree. Suppose that a register is assigned to variable so that node (ADDR *a*) is changed to (VREG *a*). Then, the two ASGN nodes do not generate any assembly instruction. Instead, their kids, ADD and MUL, perform the assignments. Note that node MUL can be scheduled as soon as its kids are scheduled. Thus it can be scheduled earlier than node ADD in the first tree. If it happens, the resulting code stores wrong value in the register. To prevent this, the output dependence needs to be adjusted so that node MUL is scheduled later than node ADD. The corresponding dependence graph is shown in Figure 3.7.



(a) Anti-dependence graph after a register is assigned to variable *c*



(b) New anti-dependence from ADD in the first tree to to MUL in the second tree

Figure 3.6 Adjustment of anti-dependence after register allocation

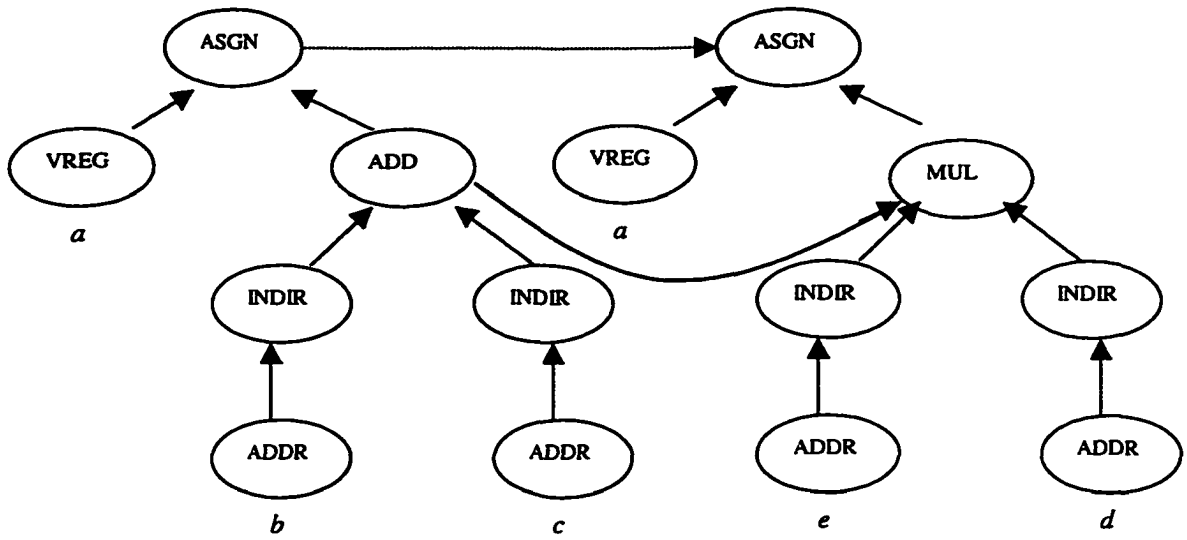


Figure 3.7 Adjustment of output dependence after register allocation

Adjustment of input dependence is similar to that of anti-dependence. For true dependence, adjustments are in general not necessary except one corner case. Function `rewriteDep( )` is called inside `rewriteAgain( )` after nodes are labeled again based on register allocation. This function takes care of one corner case for the adjustment of true dependence. Then, it calls function `adjustFalseDep( )`, shown in Figure 3.8, for the adjustment of input dependence, output dependence, and anti-dependence. Function `adjustFalseDep( )` calls three functions `adjustAntiDep( )`, `adjustOutputDep( )`, and `adjustRegDep( )`, each of which performs adjustments for anti-dependence, output dependence, and input dependence, respectively.

```
void adjustFalseDep( forest ) Node forest;
{
      for( p = forest; p; p = p->link )
            if( generic( p->op ) == ASGN ) {
                  if( p->d.antiFrom )   /* anti-dependence */
                        adjustAntiDep( p );
                  if( p->d.outputFrom ) /* output dependence */
                        adjustOutputDep( p );
                  if( p->d.regTo ) /* input dependence */
                              adjustRegDep( p );
            }
}
```

Figure 3.8 Procedure for adjustments of dependence

3.3 Instruction Scheduling

Instruction scheduling is a total (or linear) ordering of nodes in ASTs. Consider the ASTs shown in Figure 3.3 again. The dependence graph in Figure 3.3 (b) represents the partial order of the nodes, but not the total order. So the instruction scheduler transforms the dependence graph into a linked list of the nodes. Figure 3.9 shows an example of scheduled nodes. In this graph, nodes are linked in a linear order. Note, however, that all ADDR nodes are not linked in the linear list, but attached as kids of the nodes in the linear list. The reason is because the ADDR nodes do not call the LCC emitter to generate an assembly instruction. Thus, only those nodes that call the LCC emitter are included in the scheduled linked list. More details on the nodes that call the LCC emitter are illustrated in [15]

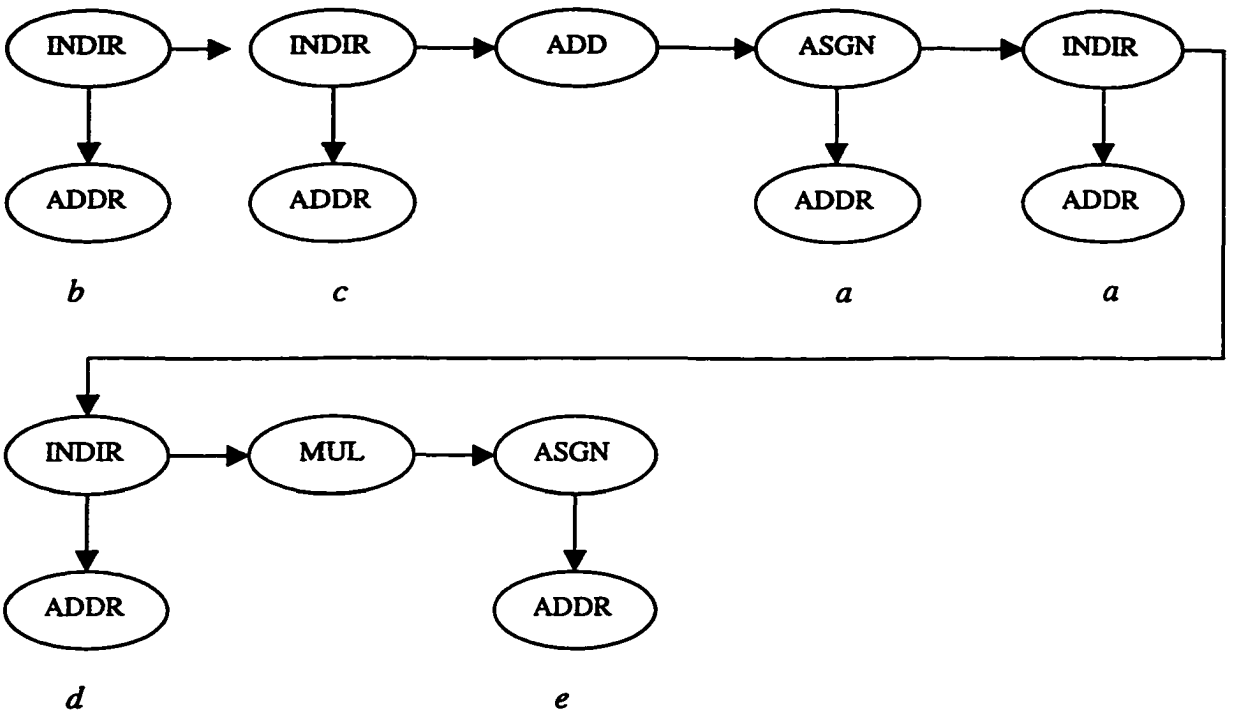


Figure 3.9 A schedule example for nodes shown in Figure 3.3

Figure 3.10 shows the structure of the instruction scheduler. The input of the scheduler is a dependence graph, and the output is the linked list of nodes in the scheduled order. The first phase of instruction scheduling is to find a set of nodes that are not dependent on any other nodes. A queue is made out of these nodes as the result of the first phase. Note that all these nodes in the queue can be scheduled next. The next phase is to select a node among the nodes in the queue and insert it into a linked list that represents the execution order of the nodes. Thus, the earlier a node is selected from the queue, the earlier the node is scheduled or executed. Once a node is selected to schedule, the next phase checks whether it has dependent nodes. If there is a dependent node, the next phase checks whether it can be scheduled next. If the node depends on any other nodes that are not scheduled at the moment, the node must be scheduled after the

dependent node is scheduled. In this case, the node cannot be inserted in the queue. Otherwise, the node is inserted in the queue, and waits for the selection phase to be inserted into the linked list. The selection and insertion phases repeat until all the nodes in the dependence graph are scheduled, and as a result the instruction queue becomes empty.

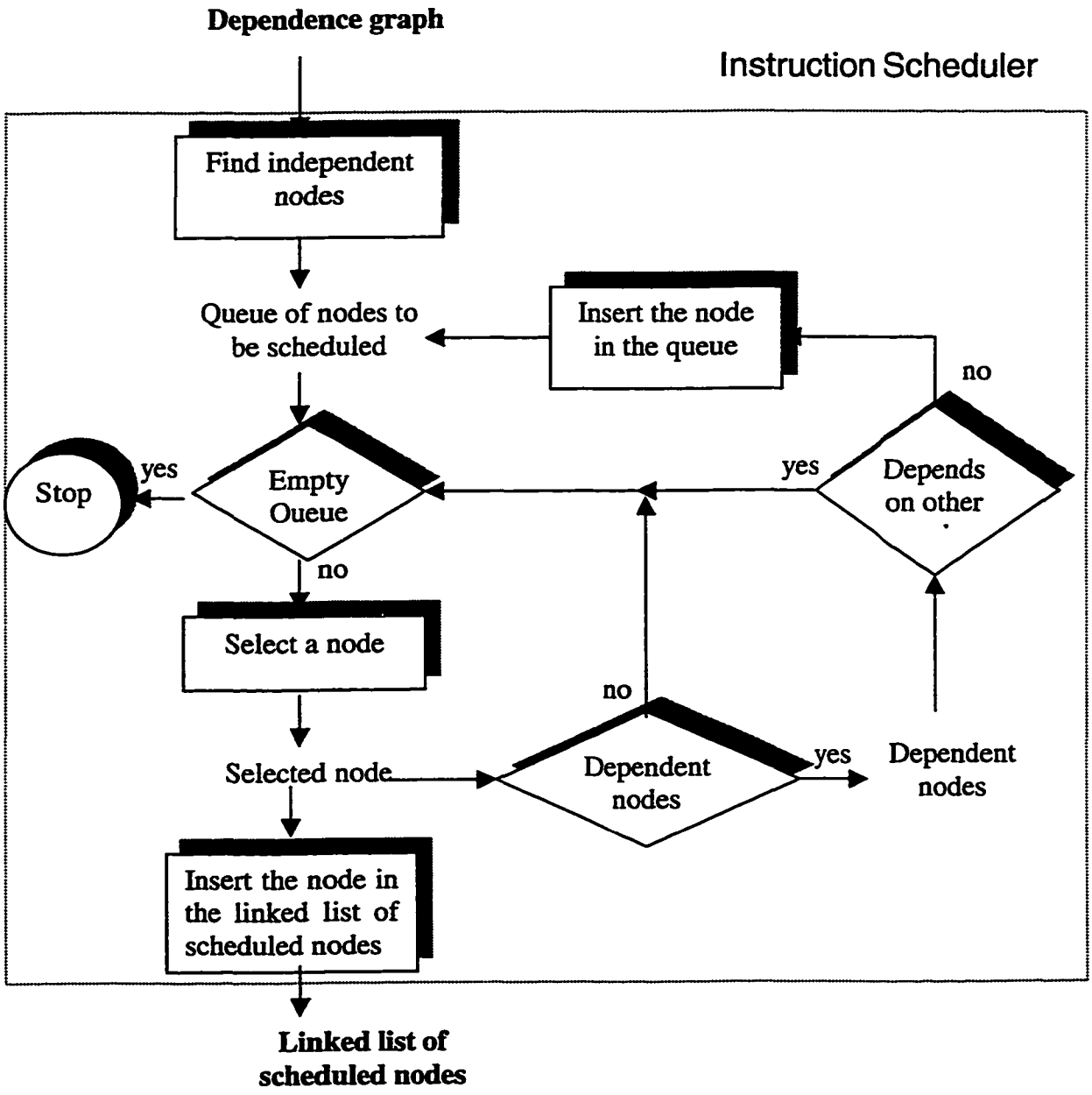


Figure 3.10 Structure of the instruction scheduler

Figure 3.11 explains the steps for how the instruction scheduler generates the scheduled nodes with a given dependence graph. Refer to the dependence graph shown in Figure 3.3 (b). The first phase of the instruction scheduler finds the nodes that are independent of any other nodes. In this example, nodes (INDIR (ADDR *b*), (INDIR (ADDR *c*)), and (INDIR (ADDR *d*)) are independent nodes. Thus, these nodes are initially inserted in the queue. The next phase is to select one of the nodes in the queue. Depending on the constraints given to the scheduler, there are many different ways to select the node. In this example, assume that the node in the top of the queue (located at the bottom of the queue in the figure) is selected first. So (INDIR (ADDR *b*)) is selected first, and then inserted in the linked list of the scheduled nodes. At this stage, only one node is scheduled. In the next step, the dependent node of (INDIR (ADDR *b*)) is found. Node (ADD) depends on node (INDIR (ADDR *b*)). Then the scheduler checks whether node (ADD) depends on any other node that is not scheduled. Note that node (ADD) depends on (INDIR (ADDR *c*)) that is still not scheduled. This implies that node (ADD) cannot be scheduled until (INDIR (ADDR *c*)) is scheduled. Thus, node (ADD) cannot be inserted into the queue. Then, the queue has two nodes (INDIR (ADDR *c*)) and (INDIR (ADDR d)). The next step is to return to select a node from the queue. At the top of the queue is node (INDIR (ADDR *c*)) that is selected to schedule. So, the linked list of the scheduled nodes contains two nodes, (INDIR (ADDR *b*)) and (INDIR (ADDR *c*)). Now that (INDIR (ADDR *c*)) is scheduled, the scheduler checks whether its dependent node is ready to be scheduled. Node (ADD) is the dependent node of (INDIR (ADDR *c*)). In order for node (ADD) to be scheduled, it must be independent of any other unscheduled nodes. The only other dependent node is (INDIR (ADDR *b*)) that is already scheduled.

Thus, node (ADD) can be scheduled, and therefore, is inserted in the queue. Thus, the queue now includes two nodes, (INDIR *d*), and (ADD). The selection and insertion steps are repeated until the queue is empty. All the steps are shown in Figure 3.11.

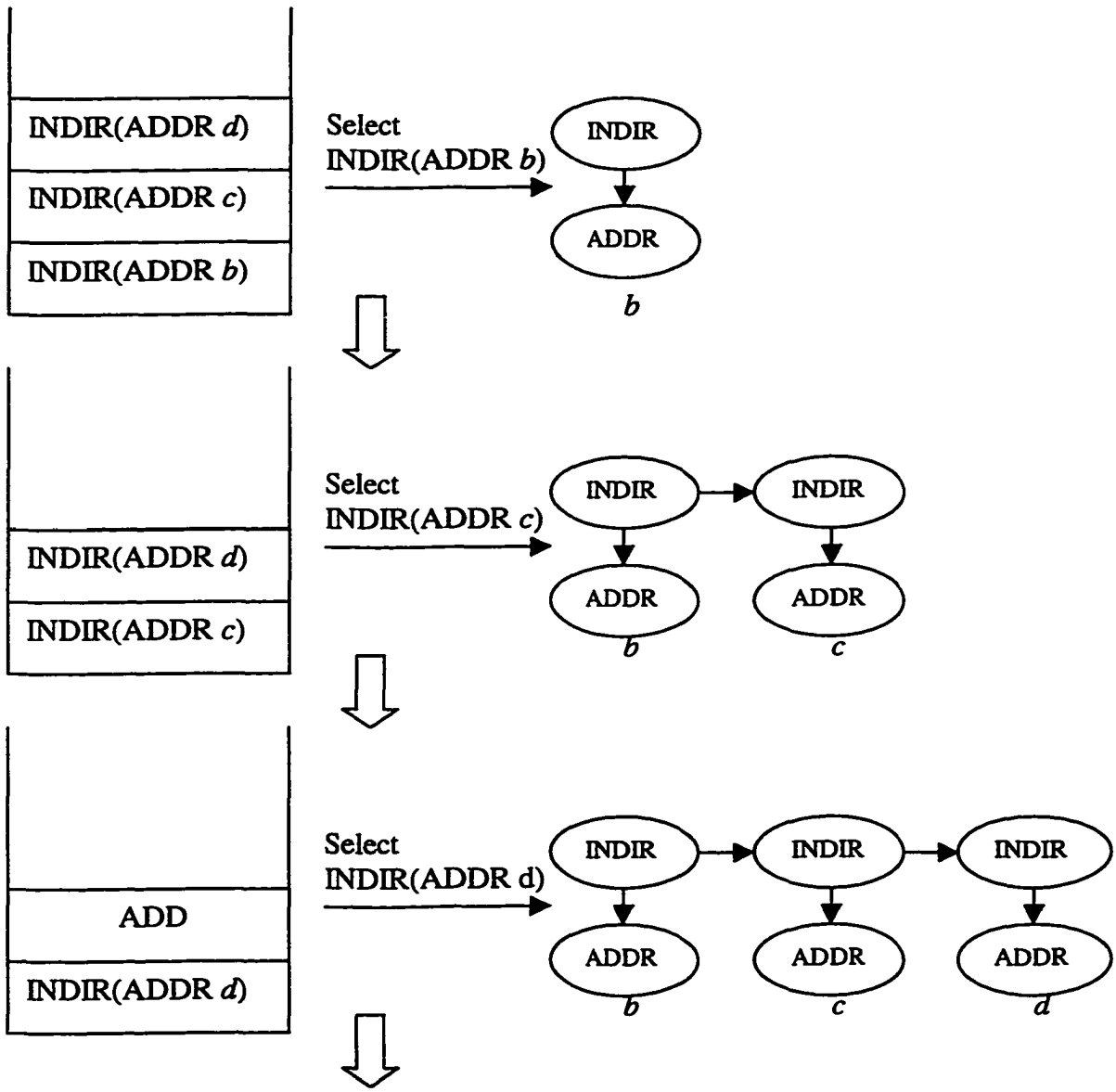


Figure 3.11 An example of scheduled nodes

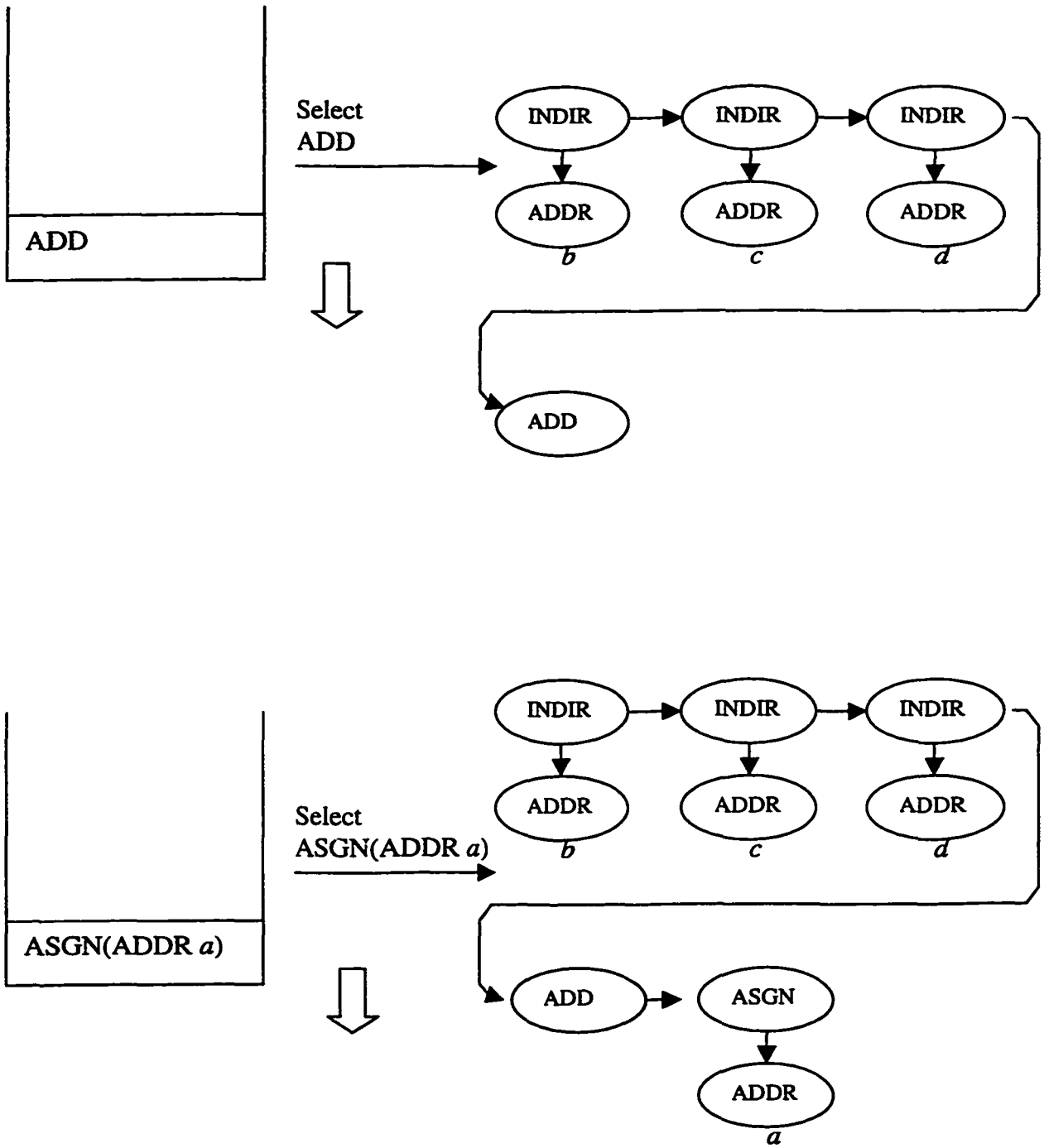


Figure 3.11 Continued

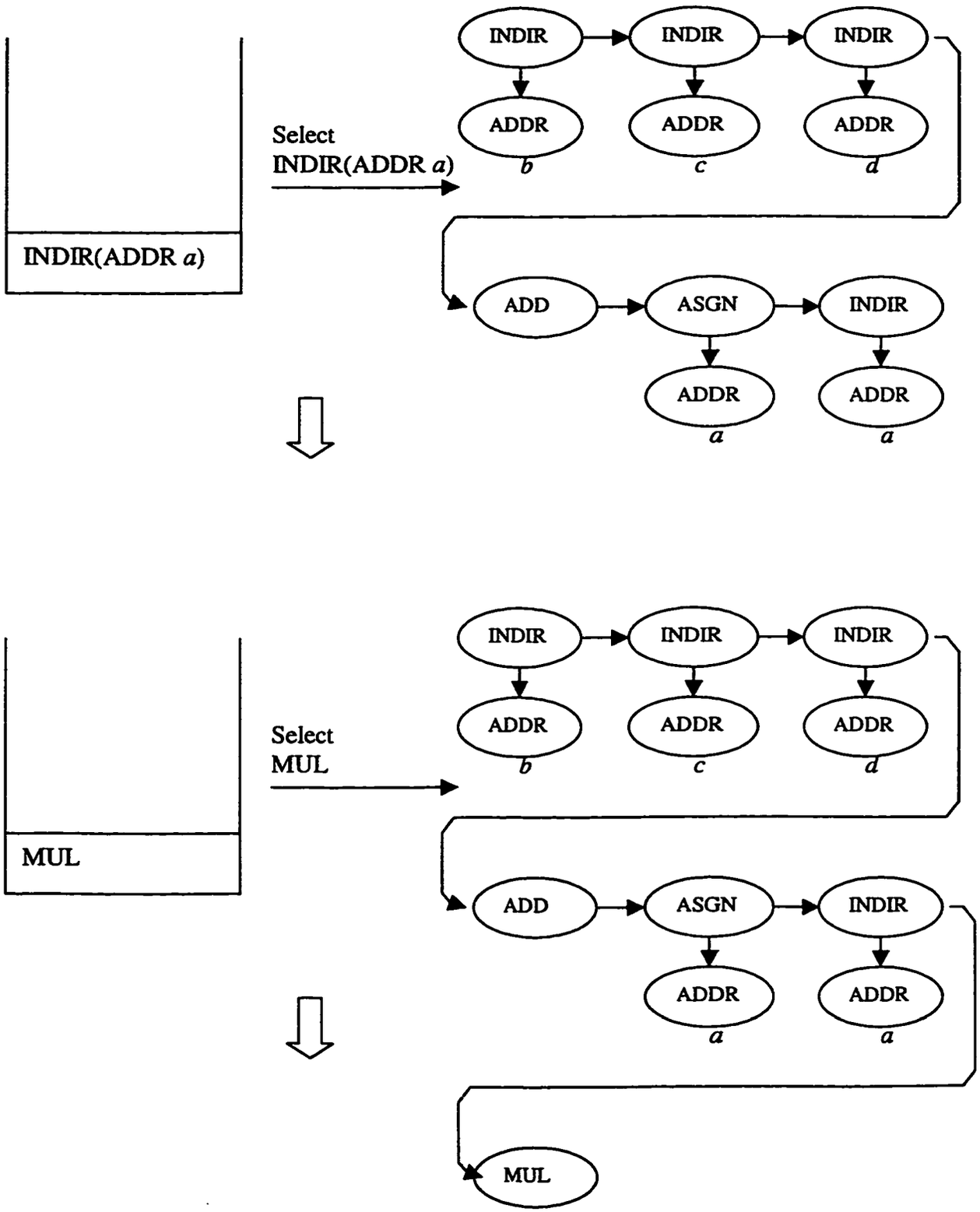


Figure 3.11 Continued

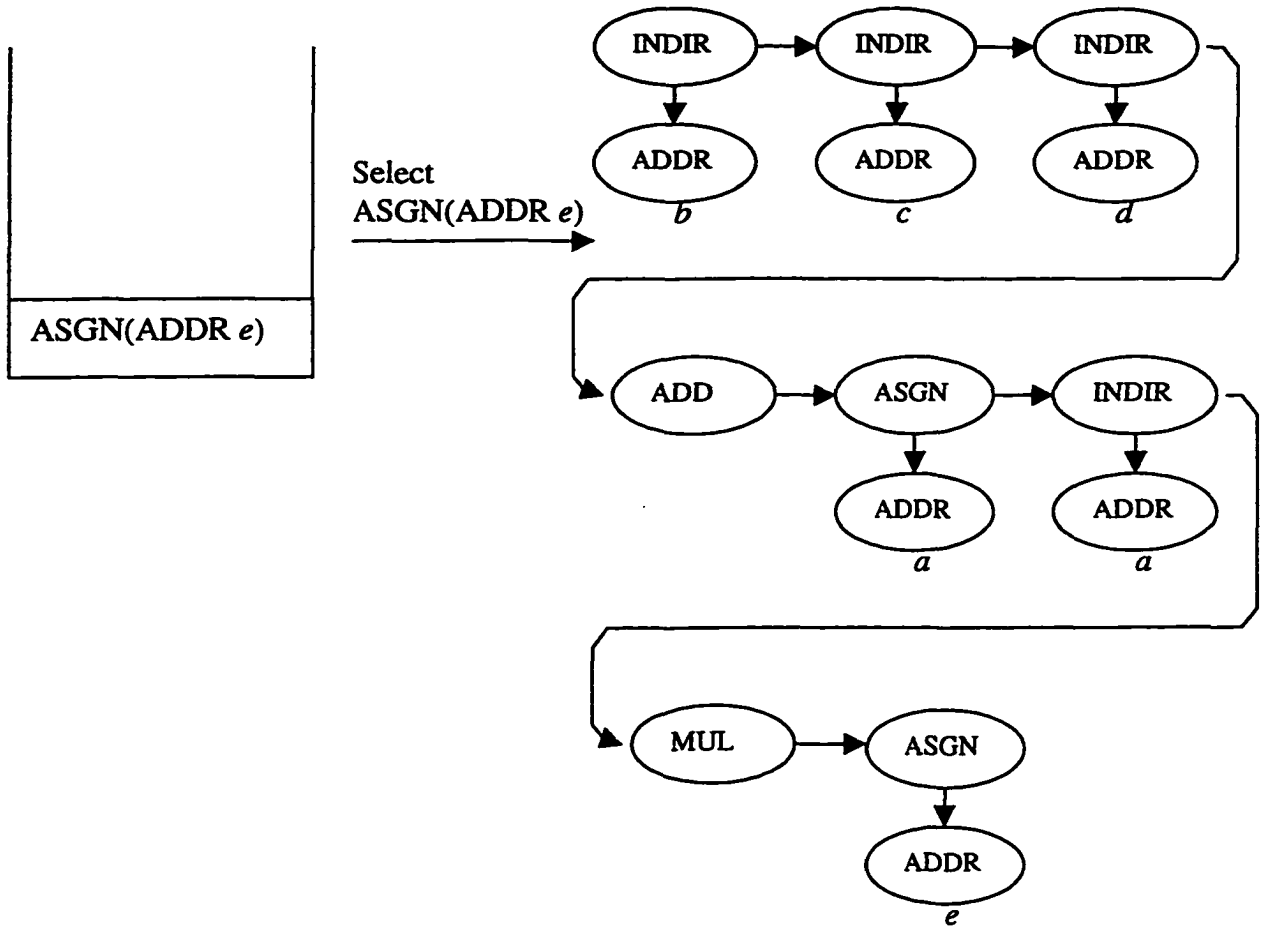


Figure 3.11 Continued

The code given in Figure 3.12 shows a rough structure of the instruction scheduler. Function `initiallyIndepInstr( )` finds all nodes that are initially independent of any other nodes. The found nodes are inserted in the queue. Then, the body of while-statement is repeatedly executed until the queue becomes empty. The first function in the body, `selectNodeInQueue( )` selects a node from the queue. Depending on various needs, this selected node can be different. In some cases, more than one node is selected. For space limitation, detailed explanation of this function is

omitted. The selected node is removed from the queue by function `removeFromQueue()`, and then inserted into the linked list of scheduled nodes. Then, function `includeDepInstrInQueue( )` searches all dependent nodes and insert them in the queue if they are not dependent on any other unscheduled nodes. The selection (`selectNodeInQueue`) and insertion (`includeDepInstrInQueue`) repeat until the instruction queue is empty, that is, `emptyInstrQueue( )` returns NULL.

```
mainScheduler( forest, next ) Node forest, next;
{
      /* initialization */
      /* find initially independent nodes and insert them in the
         queue */
      initiallyIndepInstr( forest );
      while(!(emptyInstrQueue( ))) {   /* repeat until the  queue
                                          is empty */
            sNodes = selectNodeInQueue( p );
            for( sn = sNodes; sn; sn = sn->next ) {
                  p = sn->nd;
                  p->DFScolor = SCHEDULED;
                  removeFromQueue( p );
                  /* insert p into scheduled node list */
                  includeDepInstrInQueue( p );
            }
      }
      /* reset data structure for opcode generation */
}
```

Figure 3.12 A basic structure of the scheduler

In the insertion phase, there are three important functions, `includeDepInstrInQueue( )`, `insertInstrInQueue( )`, and

`putInstrInQueue( )`. Function `includeDepInstrInQueue( )` searches all the nodes that depend on node *p*. Then, for each of the dependent nodes, `insertInstrInQueue( )` is called. The main functionality of `insertInstrInQueue( )` is to check whether node *p* depends on any other unscheduled nodes. If it is false, it calls `putInstrInQueue( )` function in order to insert the node in the instruction queue. Function `putInstrInQueue( )` puts the node in the instruction queue and updates all the data structure correspondingly.

When a node in an instruction queue is selected and scheduled, its dependent nodes are searched. If a dependent node does not depend on any other nodes that are not scheduled, the node can be inserted in the queue. Function `includeDepInstrInQueue( )`, shown in Figure 3.13, performs the search of dependent nodes. The first phase of the function is to check whether its parent node is prevented from being included in the instruction queue. In the second phase, it searches true dependent nodes that are stored in the linked list pointed by *dDependent*. In the next phase, it searches false dependent nodes by calling function, `includeFalseDepInstrInQueue( )`.

```
includeDepInstrInQueue( p ) Node p;
{
      /* prevent its parent from being included in the
            instruction queue */
      for( dn = p->dDependent; dn; dn = dn->next )
            insertInstrInQueue( dn->nd, p );

      includeFalseDepInstrInQueue( p );
}
```

Figure 3.13 Function for including dependent nodes in the queue

Function `includeFalseDepInstrInQueue( )` shown in Figure 3.14 searches false dependent nodes. Note that there are three different types of false dependencies: anti-dependence, output dependence, and true dependence. These dependent nodes are pointed by the fields, *d.antiTo*, *d.outputTo*, and *d.regTo*, respectively. Function `insertASGNnodeInQueue( )` checks whether the dependent node is also dependent on any other unscheduled nodes. If it is false, then the dependent node is inserted in the instruction queue.

```
void includeFalseDepInstrInQueue( p ) Node p;
{
      Node q;
      if( q = p->d.outputTo ) insertASGNnodeInQueue( q, p );
      if( q = p->d.antiTo ) insertASGNnodeInQueue( q, p );
      if( q = p->d.regTo ) insertASGNnodeInQueue( q, p );
}
```

Figure 3.14 Function for including false dependent nodes in the queue

When `includeDepInsrInQueue( )` finds a node to be inserted in the instruction queue, it calls function `insertInstrInQueue( )` shown in Figure 3.15. First, this function checks whether it is an instruction node or not by evaluating *x.inst* field. If the node is an instruction node, it can be inserted in the queue. So function `insertXNodeInQueue( )` is called. If the node is not an instruction node, check whether the node is an ADDR node or a VREG node. If it is true, the conditions specific to the ADDR/VREG node are checked, and insert the node in the queue if the conditions are satisfied. If the node is not an ADDR/VREG node, function `includeDepInsrInQueue( )` is recursively called to search its dependent nodes.

```
void insertInstrInQueue( nd, depNode )
Node nd;
Node depNode;    /* nd is dependent upon depNode */
{
      if( nd->x.inst )
            insertXNodeInQueue( nd, depNode );
      else if( !isAddrOpOrVreg( nd->op ))
            includeDepInstrInQueue( nd );
      else
            /* if ( AddrOp or VREG ), insert the node with special
                  condition */
}
```

Figure 3.15 Function for inserting a node in the queue

Function `insertXNodeInQueue( )`, shown in Figure 3.16, checks whether a node depends on any other nodes that are not scheduled yet. If it is not true, it calls function `putInstrInQueue( )` to insert the node in the instruction queue. This check is performed by function `isAnyOtherXKids( )`. If there is such a node, it is necessary to check whether it is scheduled, that is, *DFScolor* = *SCHEDULED*. It is also necessary to check whether the node falsely depends on any other nodes that are not scheduled yet. The examination for false dependence is performed by function `isFalseDepOnNotScheduledNode( )`. This function checks anti-dependence, output dependence, and input dependence through register reuse. Once all these examinations are passed, function `putInstrInQueue( )` is called. If function `isAnyOtherXKids( )` returns NULL (that is, there is no other kids that are instruction nodes), the false dependence is the only necessary check. Thus, function `isFalseDepOnNotScheduledNode( )` is called for the examination.

```
void insertXNodeInQueue( nd, depNode )
Node nd;
Node depNode;    /* nd is dependent upon depNode */
{
      Node q;
      if( ( q = isAnyOtherXKids( nd, depNode )))
            if ( q->x.inst && getOpIndex( q, q->x.inst )
               && q->DFScolor == SCHEDULED )
                  if( !isFalseDepOnNotScheduledNode( nd ) )
                        putInstrInQueue( nd );
      else if( !isFalseDepOnNotScheduledNode( nd ))
            putInstrInQueue( nd );
}
```

Figure 3.16 Function for inserting Xnode in the queue

When function `putInstrInQueue( )`, shown in Figure 3.17, is called, the first step is to check whether the node is already scheduled, or inserted in the queue. This check prevents a node from being inserted twice in the instruction queue. The second step inserts the node in the instruction queue. This step includes the establishment of new links after the insertion as well as the update of data structure. The final step checks whether it is an assembly-generating node or not. If it is true, it searches its dependent nodes by calling `includeDepInstrInQueue( )`. In this way, the search and insertion process repeats until an assembly-generating node is inserted in the queue.

```
void putInstrInQueue( p ) Node p;
{
      if( p->DFScolor == QUEUED || p->DFScolor == SCHEDULED)
            return;
      /* insert Node p in the instruction queue */
      if( !getOpIndex( p, p->x.inst ) )
            includeDepInstrInQueue( p );
}
```

Figure 3.17 Function for putting a node in the queue

Function `selectNodeInQueue( )`, shown in Figure 3.18, is able to choose various scheduling techniques based on variable *scheduleOption*. Currently, three different scheduling techniques are implemented. The first one is to select the instruction that minimizes the hamming distance of opcode between consecutive instructions. This technique is useful for reducing power consumption due to frequent change between consecutive opcode values. The second scheduling technique is designed for VLIW architecture that can execute multiple instructions simultaneously. The third scheduling is to simply select the first instruction node in the queue.

```
void SelectedNode selectNodeInQueue( )
{
    switch( scheduleOption )
    {
        case BIT_CHANGE_REDUCTION:
           sn = callMinBitChangeOpInQueue( sn );
           break;
        case VLIW:
           sn = callVliwSchedule( sn );
           break;
        default:
           sn = firstNodeInQueue( );
           break;
    }
}
```

Figure 3.18 Function for selecting a node from a queue

Various instruction scheduling techniques can be combined together and executed simultaneously. In order to support the combination of multiple instruction scheduling techniques, `callMinBitChangeOpInQueue` or `callVliwSchedule( )` shown in Figure 3.19 supports the case when there are some nodes selected by the previous

scheduler. For example, see function `callVliwSchedule( )`. When there is a previously selected node (that is, *sn* is not NULL), the function attaches the selected node by calling `vliwSchedule( )` to the tail of the selected node list.

```
SelectedNode callVliwSchedule( sn ) SelectedNode sn;
{
    if( sn ) {
        SelectedNode lsn;
        lsn = lastSelectedNode( sn );
        lsn->next = vliwSchedule( );
    }
    else
        sn = vliwSchedule( );
    return sn;
}
```

Figure 3.19 Function for calling VLIW scheduling

CHAPTER FOUR

REGISTER ALLOCATION

This chapter presents an improved register allocation algorithm developed in this research. The major algorithms include register-reuse chain generation, chain merging, and the merging criteria. A discussion of the previous research on register-reuse chain in [7, 8] is also provided. For convenience of descriptions, some formal definitions related to those algorithms are given.

4.1 Background

As mentioned in Chapter 2, instruction scheduling and register allocation are the two major compiler optimization techniques. In most research efforts, these two techniques are studied separately [7, 8, 30, 31]. One phase is performed before the other. The resulting approaches are called phase ordering solutions. However, these two optimizations often significantly influence each other. Optimization in one phase adversely affects the optimization in the other phase. The first phase ordering approach (that is, instruction scheduling followed by register allocation) gives priority to instruction scheduling. An instruction scheduler decides the live range of a variable and consequently places significant constraints on register allocation. Therefore, even efficient instruction scheduling can degrade the overall optimization if the scheduler places too many constraints on register allocation, resulting in poor register allocation. Some variables'

values may spill in main memory. As a result, performance is lowered due to main memory access latency, and code size increases correspondingly due to extra memory access instructions generated.

On the other hand, the second approach, register allocation followed by instruction scheduling, gives priority to register allocation. Register allocation affects instruction scheduling because it often creates additional dependencies that add constraints to the scheduler. A common approach to register allocation is graph-coloring. The graph-coloring approach formulates a register allocation problem as a graph-coloring problem. Each vertex of the graph represents a variable in the program. If two variables' live ranges are overlapped, an edge exists between two vertices. The graph-coloring approach attempts to assign a color to each vertex in the graph with a minimum number of colors used, such that no two vertices between which there is an edge has the same colors. The variables with the same color can share a register.

In order to develop an optimizing compiler that would be efficient for both a scheduler and a register allocator, recent research has been focused on the integration of these two techniques [7, 8]. Berson, Gupta, and Soffa make a promising contribution by proposing register allocation based on register-reuse chains [8]. A register-reuse chain is defined as an ordered set of instructions that use the same destination registers. Thus, register allocation in [8] is a procedure for decomposing a dependence graph into register-reuse chains. Each reuse chain requires a register so that the number of necessary registers is the same as the number of register-reuse chains. If the number of chains is greater than the number of registers, dependencies are added to the dependence graph that leads to the reduction of the number of register-reuse chains (see details in later sections). Since the

addition of dependencies generates additional restrictions for an instruction scheduler, efficient heuristics are proposed in [8] to reduce unnecessary restriction.

Although the main idea of the register-reuse chain approach is promising, the method proposed by [8] can still be improved. This is because it does not have a systematic approach to derive the best register-reuse chains and consequently it can result in an inappropriate selection of register-reuse chains. Another improvement is needed because the efficiency of the previous heuristic can be lowered when statements have various execution times. The previous method is optimized assuming that every statement in a program has the same execution time. However, it is often the case that different statements can have different execution times, because different statements can have different types of operations as well as different number of operations.

The register allocation technique proposed in this research follows the framework in [7,8], and improves the efficiency of the technique. The first step is to find a register allocation that is optimal in the sense that no additional dependencies are created. This optimal register allocation sometimes requires a large number of registers that is greater than the number of available registers. For this case, a heuristic is proposed to reduce the number of necessary registers while attempting to minimize the additional dependencies.

4.2 Register Allocation Based on Register-Reuse Chains

4.2.1 Definitions

For convenience of description, some definitions that are used in the algorithms are introduced as follows:

Definition: A *partial ordering* relation P on a set S is a relation on S such that

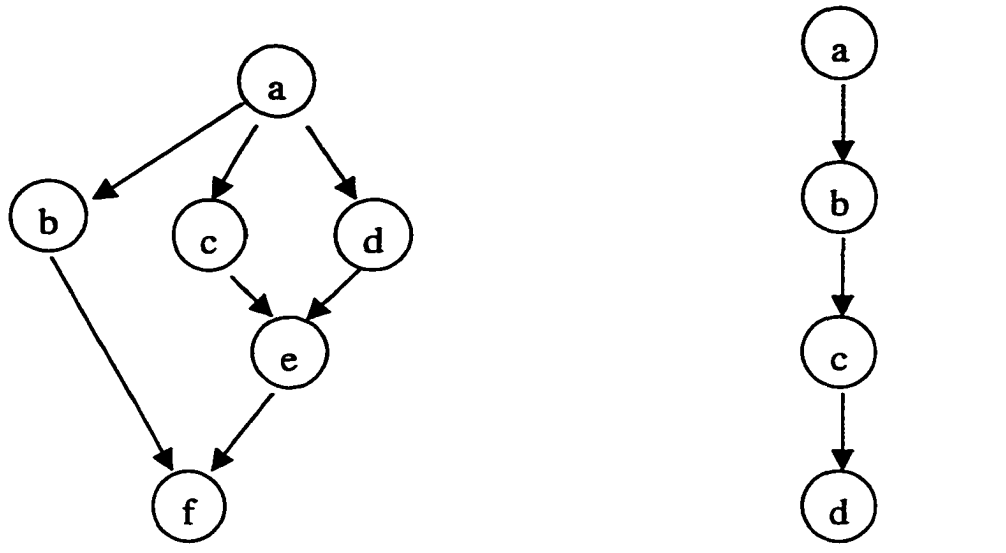
(1) (x, x) is $\notin P$ for any x in S,

(2) if $(x, y) \in P$, and $x \neq y$ then (y, x) is $\notin P$,

(3) if (x, y) and $(y, z) \in P$, then $(x, z) \in P$.

Condition 3 says P is a transitive relation. Condition 1 is called the irreflective law, and Condition 2 is called the antisymmetry law. Thus, a partial ordering is an irreflective, antisymmetric, and transitive relation [10].

According to the above definition, a data dependence graph is a partially ordered set whose elements are the nodes of the data dependent graph. The relation between the nodes in a data dependence graph can be described as "ancestor" or "descendant" relations. It is easily seen that a data dependence graph satisfies the three conditions of a partial ordering relation. For the example shown in 4.1 (a), there is a relation of "ancestor" between nodes *a* and *e*; that is, node *a* is the ancestor of node *e*. This relation can be obtained by applying the transitive condition (Condition 3), because the node *a* is the ancestor of node *c* and node *c* is the ancestor of node *e*.



(a) Data dependent graph (partial ordering) (b) Linear partial ordering

Figure 4.1 Examples of partial ordering and linear partial ordering.

Definition: A partial ordering P of a set X is *linear* if for each two elements x and y in a set X, either $(x, y) \in P$, or $(y, x) \in P$.

An example of linear partial ordering is shown in 4.1 (b) in which set $\{a, b, c, d\}$ forms a linear ordered set because there exists a relation "ancestor" between any two nodes in the set.

Definition: In a partially ordered set, a subset whose elements are linearly ordered (relative to each other) is called a *chain*.

In other words, a chain is a subset of a partially ordered set and its elements are linearly ordered. Since a data dependence graph is a partially ordered set, it could be decomposed into distinct chains. For example, subsets $\{a, e, f\}$ and $\{b\}$ are two distinct chains in the dependence graph in Figure 4.1(a).

Definition: A *register-reuse chain* is a chain in a data dependence graph in which all the nodes share the same register.

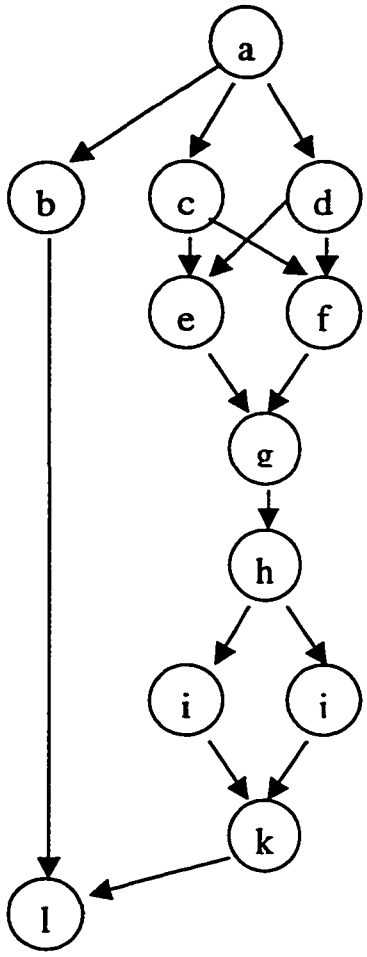
For example, by definition, chains {*a*, *e*, *f*} and {*b*} can be two different register-reuse chains in Figure 4.1 (a), respectively, if nodes *a*, *e*, and *f* share the same register and node *b* uses another register.

4.2.2 Previous Approach

In [8], the source code shown in Figure 4.2 (a) is used to explain the register allocation based on register-reuse chains. Figure 4.2 (b) shows the corresponding dependence graph in which each node corresponds to a statement in the source code. The character in the node represents the name of the variable assigned in the statement. Figure 4.2 (c) shows the register-reuse graph derived from the dependence graph. In this graph, the nodes are the same as those in the dependence graph and an edge shows the possibility of the register-reuse; that is, the successor (destination of the edge) can reuse the register of the predecessor (source of the edge). Let *E*(*a*, *b*) denote the edge from node *a* to *b*. This edge represents that *b* can kill *a*; that is, *b* can reuse the register assigned to *a*. Similarly, *E*(*c*, *f*) represents that *f* can reuse the register for *c*. Note that *f* has another incoming edge from *d*. This means that *f* can reuse both registers for *d* and *c*. However, *f* can reuse only one register. Thus, it is necessary to decide which register *f* reuses. Removing one of the edges coming into *f* can represent this decision. For example, if *c* is chosen to be reused by *f*, then removing edge *E*(*d*, *f*) can represent this decision.

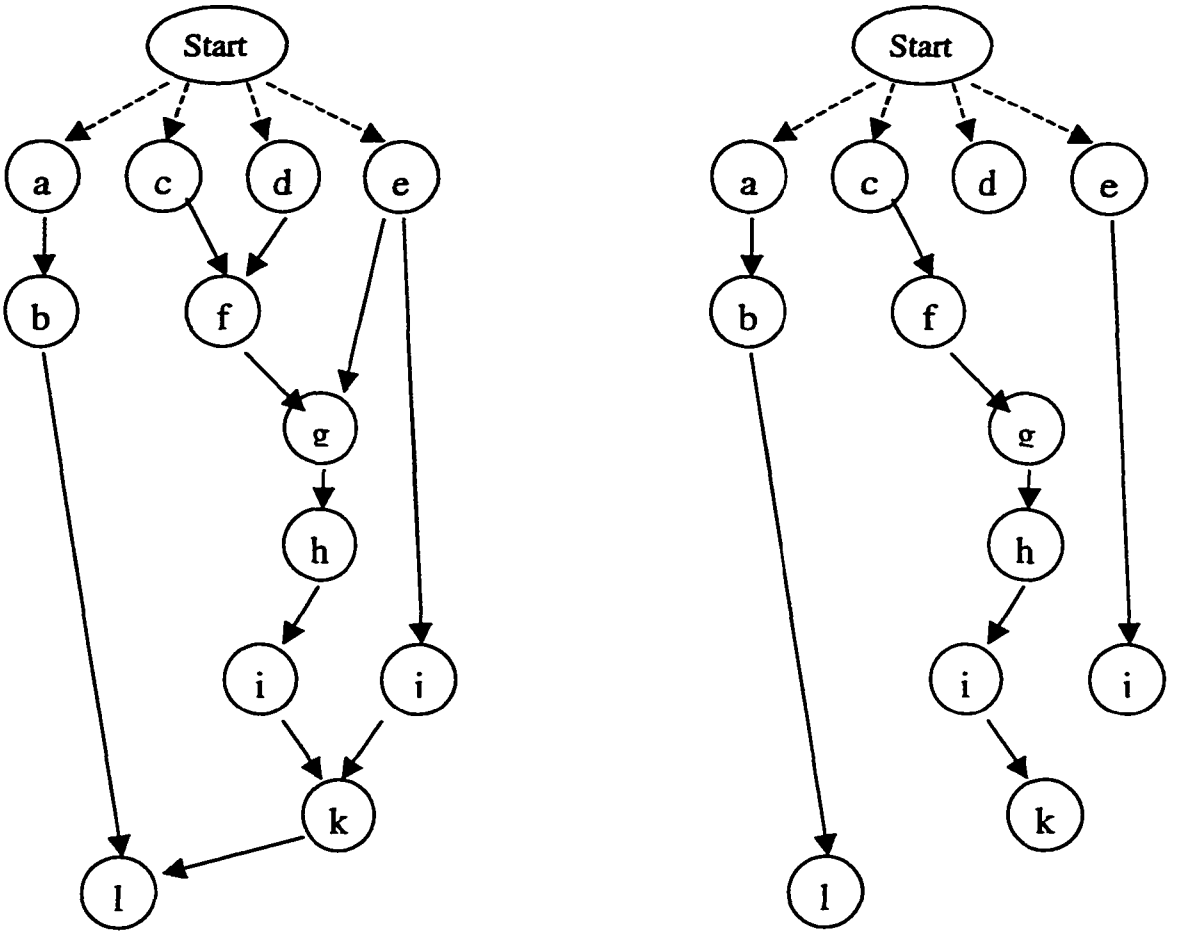
```
a = 5;
b = 2 * a;
c = a + 1;
d = a – 3;
e = c * d;
f = c – d;
g = e / f;
h = g + 5;
i = h * 2;
j = h + 4;
k = i / j;
l = b + k;
```

(a) Source code



(b) Dependence graph

Figure 4.2 Register allocation example given in [8]



(c) Register-reuse graph

(d) Register-reuse chains

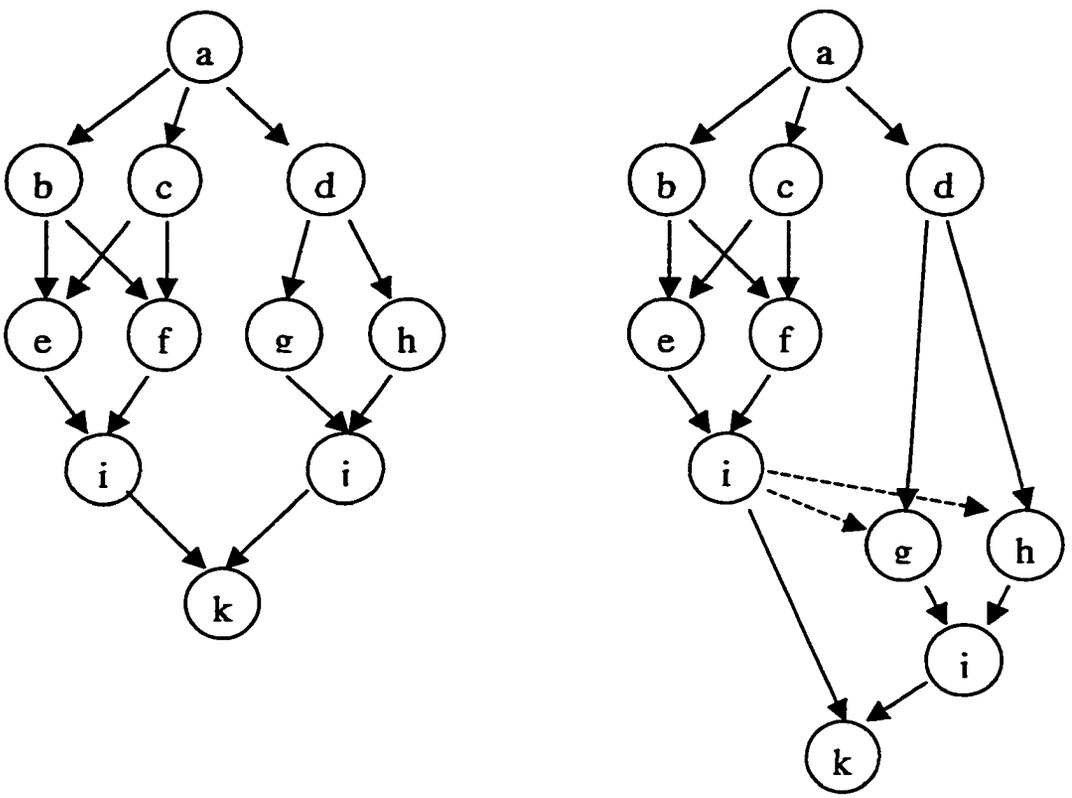
Figure 4.2 Continued

In general, in order to use the register-reuse graph for register allocation, a register-reuse graph needs to be transformed into another graph in which each node has at most one predecessor and one successor. Figure 4.2 (d) shows such a graph transformed from Figure 4.2 (c). By removing edges $E(d, f)$, $E(e, g)$, $E(j, k)$ and $E(k, l)$, this graph forms a set of chains in which all nodes have one predecessor and one successor at most. This graph can be used for register allocation in such a way that each chain is mapped to a

register. Thus, all statements in a chain are assigned to the same register. In this figure, *a*, *b*, and *l* are assigned to the same register, while *c*, *f*, *g*, *h*, *i*, and *k* are assigned to the same register. These chains in this graph are called register-reuse chains in [7, 8].

Since each chain is mapped to one register, the number of chains corresponds to the number of necessary registers. For example, the graph in Figure 4.2 (d) requires four registers. If the number of chains is greater than the number of registers, it is necessary to reduce the number of chains. In order to reduce the number of chains, [8] proposes to add dependencies in the dependence graph. For example, consider a new dependence graph as shown in Figure 4.3 (a). Five register-reuse chains are derived in [8]. Suppose that there are only four available registers. Then, [8] suggests to add dependencies from *i* to *g* and *i* to *h* as shown in Figure 4.3 (b). With the new dependencies, [8] can derive four new reuse chains. More details on the addition of dependencies can be found in [8].

There are many different ways to add dependencies. So optimization is necessary to select which additional dependencies need to be added. In [8], the criterion for the addition of new dependencies is the length of the critical path in the dependence graph. For example, the added dependencies in Figure 4.3 (b) increase the length of the critical path by one. In [8], a method is developed to add dependencies that attempt to minimize the increase of the critical path length. In addition, further optimizations are developed in [8] for the integration of instruction scheduling with the register allocation, the generation of register spill/reload instructions, and the optimization across basic blocks. Since the additional optimizations are not the interest of this research, detailed explanation is omitted.



(a) Dependence graph (b) Additional dependencies shown in dotted lines

Figure 4.3 Creation of dependencies for register allocation

4.2.3 Possible Improvements

This chapter concentrates on the generation of the register-reuse graph, reuse chains, and the reduction of reuse chains. Improvements are attempted based on the following observations. In the generation of the register-reuse graph, the selection of a possible killing node can affect the efficiency of a scheduler. Recall that the killing node is the node that can reuse the register assigned to its predecessor. The previous research does not have a systematic approach to select the killing node, and consequently can

degrade the efficiency of a compiler. For example, consider the following dependence graph shown in Figure 4.4.

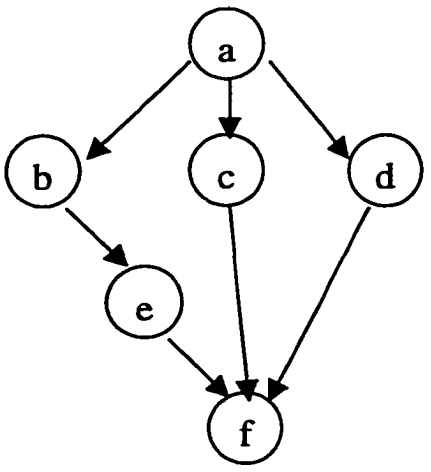


Figure 4.4 Example of dependence graph

Suppose that *b* is selected to reuse the register for *a*. This forces that *c* and *d* must be computed no later than *b*. Otherwise, *a* is not available for the computation of *c* and *d* because *a* is already replaced by *b*. Assume that there are only two functional units. Since *c* and *d* cannot be executed later than *b*, a scheduler must force *c* and *d* to be executed immediately after the execution of *a*. Then, node *b*, *e*, and *f* must be executed sequentially due to dependencies between them. This requires 5 steps to complete the computation. Suppose that *c* is selected to reuse the register of *a*. Then, *b* and *d* need to be scheduled right after *a*. In the next step, *c* and *e* can be scheduled simultaneously. Finally, *f* is scheduled. This requires four steps to complete the computation. This example shows the importance of the selection of the initial reuse graph.

In [8], dependencies are added for reducing the number of register reuse chains. In this phase, it is often the case that there is more than one choice in the selection of dependencies. [8] uses the criterion based on the increase of the length of the critical path in the dependence graph. This criterion is useful when each statement requires the same execution time. However, each statement computes different operations and therefore can have different execution times. In addition, each statement can have a different number of operations that can further differentiate the execution time of a statement. This research proposes a new criterion for the reduction of register-reuse chains. Based on the new criterion, a heuristic is proposed to reduce the register-reuse chains. The new heuristic is designed to be used efficiently for the general case when each statement can have different execution time.

4.3 Register-Reuse Chain and Dependence Analysis

Register allocation is a procedure to decide which variables are stored in registers and which of them share the registers. If two variables are assigned to the same register, their live ranges must not be overlapped. If registers are allocated before instructions are scheduled, register allocation, in general, forces an ordering in the live ranges of variables, resulting in the creation of additional dependencies between instructions that access the variables. However, these additional dependencies can be avoided if the forced ordering complies with existing dependence relationship. This section investigates register allocation that does not create any additional dependencies. First, the creation of dependencies due to register allocation is studied and then an algorithm is proposed to allocate registers without the creation of additional dependencies.

4.3.1 Generation of Dependence Due to Register Allocation

The execution results of a given C code do not change with the schedules that comply with the data dependencies. However, when a C code is translated into assembly code, register allocation for each variable is involved and additional dependencies may be introduced by register allocation. Therefore, when instructions are scheduled, these extra dependencies due to register allocation should be taken into consideration in addition to normal data dependencies.

Consider the example shown in Figure 4.5. The left column in the figure shows a segment of C code. The corresponding data dependence graph is given in Figure 4.7. Assume that variables *a, b, e,* and *f* are allocated to register R3, variable *c* to register R4, and variable *d* to register R5. The corresponding assembly code is shown in the middle column in Figure 4.5. The execution results based on the assembly code are shown in the right column. Note that the computation result of *f* is 80, which is incorrect. The reason is that when instruction (1) is finished, variable *b* reuses register R3 originally allocated to variable *a*. When variables *c* and *d* use of the value of variable *a*, it is already replaced with the value of variable *b* and no longer available. In order to correctly execute the code, instructions (3) and (4) must be executed before instruction (2). To force the execution ordering, it is necessary to create additional dependencies from instruction (3) to (2) and from (4) to (2). The new dependencies are shown with dashed arrows in the data dependence graph (see Figure 4.7). The additional dependencies force instruction (2) moved behind both instructions (3) and (4), which is shown in Figure 4.6. The correct execution results are shown in the right column of the figure.

(1)	a = 5;	R3 = 5	(5)
(2)	b = a + 1;	R3 = R3 + 1	(6)
(3)	c = 2 * a;	R4 = 2 * R3	(12)
(4)	d = a / 3;	R5 = R3 / 3	(2)
(5)	e = b + c + d;	R3 = R3 + R4 + R5	(20)
(6)	f = 4 * e;	R3 = 4 * R3	(80)

Figure 4.5 Example C code and corresponding assembly code

(1)	a = 5;	R3 = 5	(5)
(3)	c = 2 * a;	R4 = 2 * R3	(10)
(4)	d = a / 3;	R5 = R3 / 3	(1.67)
(2)	b = a + 1;	R3 = R3 + 1	(6)
(5)	e = b + c + d;	R3 = R3 + R4 + R5	(17.67)
(6)	f = 4 * e;	R3 = 4 * R3	(70.68)

Figure 4.6 C code and assembly code after rescheduling

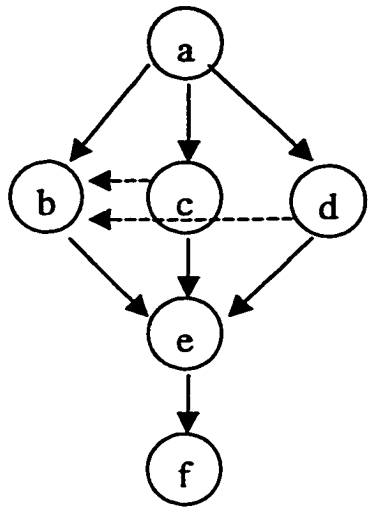


Figure 4.7 Data dependence graph

In general, if a node in a dependence graph has multiple successors, additional dependencies need to be created when one of its successors reuse the same register assigned to its predecessor. In Figure 4.7, node *a* has multiple successors: *b*, *c*, and *d*. If *b* is selected to reuse *a*, additional dependencies need to be created from *c* to *b* and from *d* to *b*. Since the new dependence is necessary to guarantee the values of predecessors are available as an input to successors, it is called an *input dependence* in this research. The new dependence is different from a normal dependence in the sense that it requires the dependent node to be executed no earlier than the node, but does not require the dependent node to be executed strictly later. If there are multiple functional units, a node can be executed simultaneously with its dependent node. The following proposition summarizes the discussion on the creation of input dependence by register allocation.

Proposition 4.1 If a node in a dependence graph has multiple successors and one of the registers assigned to the node is reused by one of its successors, then an input dependence

is created from the other successors to the successor that reuses the register. The input dependence forces the scheduler to order the execution of other successors no later than the execution of the node that reuses the register.

Consider the dependence graph shown in Figure 4.7 again. If the variables *a*, *e*, and *f* are allocated in the register R3, and the variables *b*, *c*, and *d* in three different registers, say, R4, R5, and R6, variable *e* reuses the register of variable *a*. Based on the data dependence analysis, instruction (5) will be executed behind instructions (2), (3), and (4) which use the value of the variable *a*. This implies that at the point where the instruction (5) is executed variable *a* is guaranteed to be dead, so the register allocated to variable *a* can be reused safely by variable *e* regardless of the execution order among the three instructions (2), (3), and (4). In this way no additional dependence is generated.

Suppose that three functional units are available. Then, *b*, *c*, and *d* can be executed simultaneously. When *b* reuses the register assigned to *a*, variables *c* and *d* have already accessed the value of *a*. Therefore, the computation for *c* and *d* can generate the correct result. However, if there is only one functional unit, the input dependence behaves exactly the same as a normal dependence. Therefore, instruction (2) has to be moved behind instructions (3) and (4) as shown in Figure 4.6.

Consider another dependence graph shown in Figure 4.8. Suppose that a register allocator assigns the same register to *a* and *c*. In addition, *b* and *d* share the same register. Since *c* reuses *a*, an input dependence from *d* to *c* is generated. Similarly, another input dependence from *c* to *d* is generated because *d* reuses *b*. The generated dependencies are shown as dotted lines in Figure 4.8. With the two input dependencies between *c* and *d*, these two nodes must be executed simultaneously. If there is only one functional unit, the

simultaneous execution is impossible. Then, register allocation needs to be changed because either the register of *a* cannot be reused by *c* or *b* cannot be reused by *d*. If *c* (or *d*) does not reuse the register of *a* (or *b*) but is assigned to a new register, the input dependence from *d* to *c* (or from *c* to *d*) is removed. Therefore, by computing in the order of *a*, *b*, *d* and *c* (or *b*, *a*, *d* and *c*), the correct result is obtained. In this case, however, note that three registers are necessary because *b* and *d* share one register while *a* and *c* need different registers.

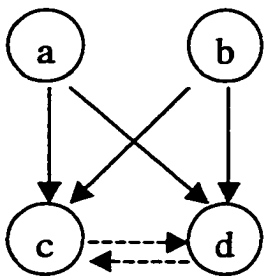


Figure 4.8 An example for additional dependence forcing simultaneous execution of *c* and *d*

4.3.2 Generation of Register-Reuse Chains without Additional Dependencies

An input dependence is generated because a register is reused by another variable even though the variable is still alive. Therefore, in order to prevent additional dependence, the end of the live range needs to be found. However, when an instruction scheduling is not fixed, it is impossible to find the exact end. However, it is often possible to derive the latest possible end at which the variable is guaranteed to be dead. The first phase of register allocation is to find the latest possible end of a live range.

Definition Let *a* be a node in a dependence graph. An ultimate *killing node* of *a* is a node through which all paths starting from *a* pass.

In Figure 4.7 *e* is an ultimate killing node of *a* because all paths starting from *a* path through *e*. Similarly, *f* is also an ultimately killing node of *a*. In general, there may be more than one ultimate killing nodes.

Definition The *earliest ultimate killing (EUK) node* of node *a* is the ultimate killing node that is the ancestor of all other ultimate killing nodes of *a*. The *EUK* node of *a* is denoted by *EUK* (*a*).

In Figure 4.7 nodes *b*, *c*, and d have two ultimate killing nodes, *e* and *f*. Node *e* is the *EUK* node because the node *e* is the ancestor of *f*. For example, *e* is the first node that guarantees that *a* is dead. In fact, *a* is dead earlier than *e* because *a* is dead when *b*, *c*, and *d* are executed. However, it is not decided which one among *b*, *c*, and *d* are executed later before the instruction scheduling is fixed. Thus, no node among *b*, *c*, and *d* guarantees that *a* is dead. Only *e* guarantees that *a* is dead because *b*, *c*, and *d* are already executed when *e* is executing. In general, the *EUK* node is the first node that guarantees the end of the live range. The *starting nodes* in a data dependence graph are the nodes without any ancestor.

Register allocation without the generation of additional dependence is possible by taking advantage of the property of an *EUK* node. Since a variable is guaranteed to be dead at *EUK*, registers can be reused at its *EUK* without additional dependencies. In addition, any node succeeding the *EUK* node also can reuse the register. Therefore, the register allocation algorithm proposed in this section recursively finds its *EUK* node and

assigns the same register to the *EUK* node. If the *EUK* node is already visited from another node and assigned to a register, a successor of *EUK* is chosen.

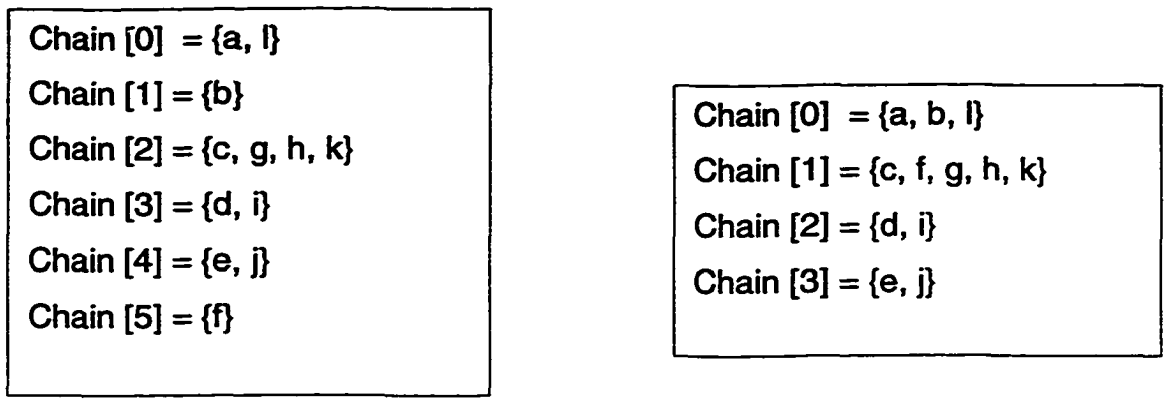
A data dependence graph from [8] shown in Figure 4.2 (b) is an example that demonstrates the register allocation algorithm. Initially, the register-reuse chain generation is started with the root of the input DAG. When there is only one starting node in the DAG, the starting node is chosen as the root. If there are multiple starting nodes in the DAG, a "virtual root", whose kids or dependent nodes are those starting nodes, is created. But the "virtual root" is not included in any chain and just functions as the beginning node in Breadth First Search (BFS) order. The algorithm starts with root node *a* and searches its *EUK* node. Note that *a* has three edges incident into nodes, *b, c,* and *d.* The three paths merge at node *l*, that is, *l* is the *EUK* of *a*. This means that *l* is the first node that guarantees *a* is dead so that *l* can reuse the register assigned to *a* without additional dependencies. The register allocation algorithm assigns the same register to both *a* and *l*. Since the node *l* does not have any dependent node, the recursive search of *EUK* node stops, and the algorithm starts with another node.

For the selection of the next node, nodes are visited in BFS order. Therefore, node *b* is visited next. Then node *l* is found as its *EUK* because the node *l* is the only dependent node of *b*. However, node *l* is already visited and assigned to a register, so it cannot reuse the register of *b*. Since node *l* does not have any dependent node, the search procedure stops. Thus *b* is the only variable that is assigned to the second register.

Now another new search for the third register is initiated starting with node *c,* the next visited node based on BFS. The *EUK* searching finds *g* as the *EUK* node of *c* and it is not visited. Thus, *g* is selected to share a register with *c*. Then, recursively finding the

EUK of node *g*, *h* is found. Thus, *h* is selected to reuse the register of *g*. Then *k* is found as the *EUK* node of *h*, and *k* reuses the register of *h*. Finally, node *l* is found as the *EUK* of *k*, but it is already assigned to the first register. So, the searching stops, and *c*, *h*, *g*, and *k* are selected to share the third register.

The forth search starts with *d* and finds *g* as its *EUK* node. However, *g* is assigned to the previous register. In this case, searching finds the first unvisited successor of *g* is *i* and then *i* is selected to reuse the register of *d*. Then, searching begins with node *i* and finds *k* as its *EUK* node. However, it is already assigned. The searching finds its successors. All successors are already assigned to registers. So the searching stops, and *d* and *i* are selected to share the same register. By visiting all the nodes that are not assigned and recursively searching its *EUK*, *e* and *j* are selected to share a register, and *f* is assigned to a new register. Figure 4.9 (a) shows the result of the register allocation algorithm. A chain is used to collect all the nodes that share the same register. Then each chain is mapped to an available register. These chains are called *register-reuse chains.*



(a) Before merge (b) After merge

Figure 4.9 Resulting register reuse chains

Figure 4.10 shows the algorithm for the generation of the register-reuse chains. The algorithm consists of two main functions, a driver function and a work horse function. The driver function `Build_Register_Reuse_Chains (DAG)` takes data dependence graphs or DAGs as input. It visits each node in the DAGs in BFS order. If a node named *p* in the DAGs is not visited yet, it creates a new chain with *p* as the head node of the chain. Then the work horse function `Build_A_Register_Reuse_Chain (p)` is called by the driver function whenever a new chain is initiated. The functionality of `Build_A_Register_Reuse_Chain (p)` is to find the *EUK* node of *p* and returns it. If the *EUK* node exists and not visited yet, it is attached to the chain. If the *EUK* node is already visited (that is, included in the other chain), the first unvisited successor of the *EUK* node is found in BFS order. If such a node exists, it is attached to the chain. Repeat this search-and-attach process recursively

until no more nodes can be attached to the current chain. The driver function stops when all the nodes in DAGs are visited.

```
Build_Register_Reuse_Chains ( DAG )
{
      Node p  = visit DAG in Breadth First Search ( BFS ) order;
      if p is not visited
            attachNodeToChain ( p );
            Build_A_Register_Reuse_Chain ( p );
            chainIndex ++;              /* increment chain index */
}
chain Build_A_Register_Reuse_Chain ( Node p )
{
      /* This function is used to build a chain */
      if (Node q = findEarliestUltimateKillingNode ( p ) )
            if( q is not visited ) {
            /* attach node q to current register reuse chain */
                  attachNodeToChain ( q );
                  Build_A_Register_Reuse_Chain ( q );
      }
      else  /* q is visited */
            if ( q = firstUnvisitedDescendantNode ( q ) ) {
                  attachNodeToChain ( q );
                  Build_A_Register_Reuse_Chain ( q );
            }
}
```

Figure 4.10 Register reuse chain generation algorithm.

The reuse chains created by this algorithm do not create any additional dependencies. Therefore, these chains are called *independent register-reuse chains*. The number of chains corresponds to the necessary number of registers. This algorithm always generates the minimum number of chains. However, there may be a different set of chains that have the same minimal number of chains for a given dependence graph. The

complexity of this algorithm is $O(|V|^3)$, where $|V|$ is the number of nodes in the dependence graph.

4.4 Register-Reuse Chain Merging

The number of independent register-reuse chains derived in the previous section can often exceed the number of available registers. For this case, it is necessary to develop an algorithm to reduce the number of register-reuse chains. Recall that the independent register-reuse chains are the minimal chains that do not create any dependencies. Thus, the reduction of the number of chains always creates new dependencies, resulting in additional constraints to an instruction scheduler. This section develops an algorithm that aims to reduce the number of chains while minimizing additional constraints to an instruction scheduler.

To reduce the number of register-reuse chains, we need to combine the chains. This process is called chain merging. For example, in the previous section, there are six chains generated. If there are five registers in a processor, we combine two of the six chains into a single chain such that the number of chains after combining becomes five, which is equal to the number of registers available in the processor. Recall that independent register-reuse chains are generated without introducing any additional dependency, and the number of chains generated is minimized. Thus the merging of any such two chains must create additional dependence, which in turn adds extra constraints for instruction scheduling. The merge problem becomes another optimization problem in minimizing additional constraints for instruction scheduling. In this section, an optimal chain merge algorithm and merging criterion are presented.

The chain merge problem can be formulated as:

INPUT: independent register reuse chains.

OUTPUT: merged chains.

CONSTRAINTS:

- The number of merged chains is less than or equal to the number of registers available.
- The merging complies with the existing data dependencies.

OBJECTIVE: minimize additional constraints for instruction scheduling.

4.4.1 Criterion of Chain Merging

One way to reduce the number of chains is to merge chains. When selecting the chains to be merged, many different combinations of chains are possible. The selection of chains affects an instruction scheduler because different merging chains result in different additional dependencies. If a preferred scheduling criterion is available at the register allocation phase, the best possible chains can be chosen based on the criterion. However, it is often the case that a desirable scheduling scheme is not available. For this case, this section proposes a generic criterion that can be applicable to any scheduler.

Definition 4.1 Given a dependence graph, *the number of schedules* is the number of possible orderings of the nodes.

Consider the dependence graph shown in Figure 4.12 (a). Dependencies between nodes force only parts of the order of nodes, but not the total order. So many different orderings of nodes are possible. For example, the following orders all comply with the dependencies.

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow k$

$a \rightarrow c \rightarrow b \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow k$

$a \rightarrow d \rightarrow c \rightarrow b \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow k$

However, the following order is not possible because it violates the dependence from *a* to *b*.

$b \rightarrow a \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow k$

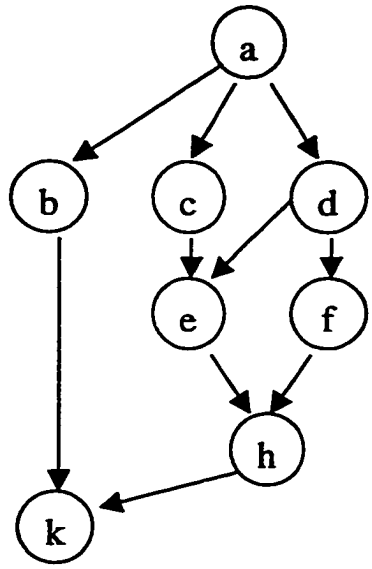
In this dependence graph, there are 30 different possible orderings of the nodes.

Among the possible schedules (or orders of nodes), an instruction scheduler selects the order that best suits the target architecture. The addition of dependencies by register allocation reduces the number of schedules, and consequently reduces the choices that can be made by the instruction scheduler. The more schedules a dependence graph has, the more choices the instruction scheduler has, so it is desirable to avoid the reduction of the number of schedules due to register allocation. Thus, the number of schedules should be used as the criterion to decide the efficiency of a register allocator.

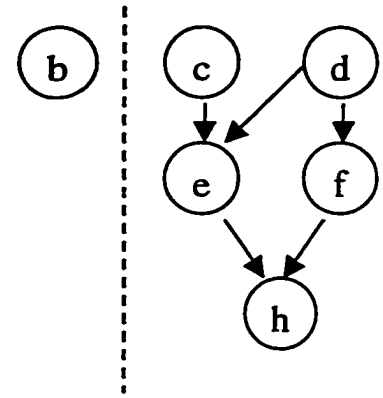
The following algorithm shown in Figure 4.11 computes the number of schedules for a given dependence graph:

```
int Compute_Num_Schedules ( DAG )
{
      If ( DAG is linearly ordered ) return 1;
      /* Remove all the nodes with fixed schedules from DAG */
      Remaining_DAG = Remove_Nodes ( DAG );
      if ( Remaining_DAG can be divided into two disjoint
            subgraphs )
      {
            Divide the Remaining_DAG into LEFT and RIGHT
            subgraphs;
            S = C(|LEFT| + |RIGHT|, |LEFT|);
            S_l = Compute_Num_Schedules ( LEFT );
            S_r = Compute_Num_Schedules ( RIGHT );
            S = S * S_l * S_r;
      }
      else
      {
            S = 0;
            for ( i = 0; i < num_starting_nodes; i++)
            {
                  /* Suppose the starting node i is scheduled first
                  among all the starting nodes */
                  Temp_DAG = Remove_Starting_Node ( Remaining_DAG,
                  i );
                  S += Compute_Num_Schedules ( Temp_DAG );
            }
      }
      return S;
}
```

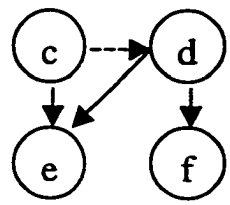
Figure 4.11 Algorithm for calculation of the number of schedules

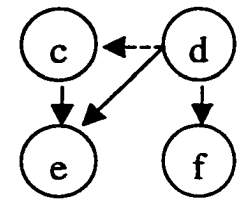


(a) Data dependence graph.

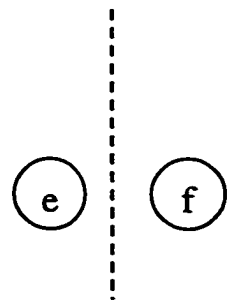


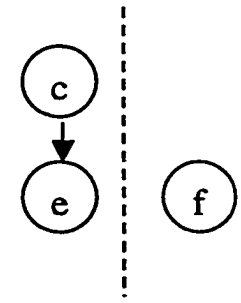
(b) Remaining dependence graph





(c) Two cases for *RIGHT* subgraph
(*c* is scheduled first vs *d* is scheduled first)





(d) The remaining graphs for the two cases in (c)

Figure 4.12 An example of calculation of the number of schedules

Suppose that there is only one processor available. To simplify the description of the algorithm, assume that each node takes one time slot to execute. In Figure 4.12 (a), there are 8 nodes in the graph. Thus the total execution time required is 8 time slots. Function `Compute_Num_Schedules( )` checks whether the input graph is linearly ordered. If it is true, it returns 1 because one schedule is possible for a linearly ordered graph. Next, `Remove_Nodes( )` removes the nodes whose orders are fixed. Note that the removal of the nodes with fixed schedule does not change the number of schedules. For example, nodes *a* and *k* have fixed execution time. In other words, node *a* must be executed before all other nodes, that is, at the first time slot; node *k* must be executed after all the nodes, that is, it is executed last. Therefore, function `Remove_Nodes( )` removes these two nodes from the graph. The remaining graph is shown in Figure 4.12 (b). Then, the if-clause checks whether the remaining graph can be divided into two disjoint subgraphs between which there is no edge. If it is true, the body of the if-statement is executed. Let the two subgraphs be denoted by *LEFT* and *RIGHT*. In the example, *LEFT* = {*b*} and *RIGHT* = {*c, d, e, f, h*}. Therefore, the original problem is reduced into a simpler problem with two smaller sizes of subgraphs *LEFT* and *RIGHT*. Since these two subgraphs are disjoint, the total number of schedules can be formulated as:

$$S(GRAPH) = C(|LEFT|+|RIGHT|,|LEFT|) * S(LEFT) * S(RIGHT) \qquad (4.1)$$

where *S* denotes the number of possible schedules for a given graph, |*LEFT*| is the number of nodes in the set or subgraph *LEFT*, |*RIGHT*| is the number of nodes in the set or

subgraph *RIGHT*, and $C(|LEFT|+|RIGHT|, |LEFT|)$ is the number of possibilities in choosing $|LEFT|$ elements out of $|LEFT|+|RIGHT|$ elements, that is,

$$C(|LEFT|+|RIGHT|,|LEFT|) = \frac{(|LEFT|+|RIGHT|)!}{|LEFT|!|RIGHT|!}$$

The term $C(|LEFT|+|RIGHT|, |LEFT|)$ can be interpreted as the number of schedules for a graph without considering how the nodes in subgraphs *LEFT* and *RIGHT* are scheduled individually. In other words, the term represents the number of ways in which the nodes in *LEFT* occupy the entire available time slots of $|LEFT| + |RIGHT|$. For example in Figure 4.12 (b), there are six nodes in the remaining graph, one in the *LEFT* subgraph, five nodes in the *RIGHT* graph. Six time slots are needed in total. Ordering of these nodes can be considered as allocating these nodes in six time slots. When scheduling these six nodes, the node *b* in *LEFT* can be allocated to any time slot from 1 to 6 while the five nodes in *RIGHT* occupy five remaining time slots. As a result, there are six ways the nodes in *LEFT* occupy six available time slots. This can be obtained by

$$C(|\{b\}| + |\{c, d, e, f, h\}|, |\{b\}|) = C(1+5, 1) = C(6,1) = 6$$

Now, it is necessary to compute *S*(*LEFT*) and *S*(*RIGHT*) that are the numbers of schedules of *LEFT* and *RIGHT* subgraphs, respectively. Thus, the problem of finding the number of schedules is decomposed into a smaller problem with two subgraphs. *S*(*LEFT*) and *S*(*RIGHT*) can be computed by recursively calling `Compute_Num_Schedules( )` function. Consider *S(LEFT)*, that is, the number of schedules of the left graph. Since there is only one element, only one schedule is possible (that is, totally ordered). Thus,

`Compute_Num_Schedules( )` returns 1. Consider the subgraph *RIGHT* = {*c, d, e, f, h*}. Node *h* should be always scheduled last, so `Remove_Nodes ( )` can remove this node. The resulting graph is shown in Figure 4.12 (c). There are two starting nodes and therefore their schedules are not fixed. These starting nodes cannot be removed, and the remaining graph cannot be decomposed into two disjoint subgraphs. So the test of the if-statement fails and the body of the else-statement is to be executed. In this case, a straightforward decomposition is impossible. In order to decompose the problem, it is necessary to arbitrarily fix the schedule of some node. Consider the node that can be scheduled as the first node. Note that only a starting node can be scheduled as the first node. Then, a subproblem is defined to calculate the number of schedules for each case that one of the starting nodes is scheduled first. In each subproblem, the chosen starting node can be removed because its schedule is fixed. Then, `Compute_Num_Schedules()` function is called again. Once every subproblem is solved, the total number of the schedules is simply the summation of the numbers of the schedules for all subproblems.

For this example, there are two starting nodes *c* and *d*. So, one of these two nodes can be chosen as the first node. Suppose that starting node *c* is scheduled first as shown in Figure 4.12 (c). Then, this node can be removed from the graph. In addition, node *d* can also be removed because its schedule is fixed as the second node. The resulting graph is shown in Figure 4.12 (d). Now, the graph can be decomposed into two disjoint subgraphs, and the number of schedules can be computed from Equation (4.1). The other subproblem addresses the case when starting node *d* is chosen to be the first node (see Figure 4.12 (c)). Then this node can be removed from the graph and the resulting graph can be

decomposed into two disjoint subgraphs (see Figure 4.12 (d)). Again, the number of schedules can be computed from Equation (4.1). The total number of schedules is the summation of the numbers of the schedules for the two cases:

$S(RIGHT) = S(\{c, d, e, f, h\})$

$= S(\{c, d, e, f\} | c$ is scheduled first) + $S(\{c, d, e, f\} | d$ is scheduled first)

$= S(\{e, f\}) + S(\{c, e, f\}) = C(2, 1) * 1 * 1 + C(3, 1) * 1 * 1$

$= 2 + 3 = 5$

Since the subgraph *LEFT* has only one node $\{b\}$, the number of schedules is one. Substituting $S(LEFT)$ and $S(RIGHT)$ into the previous formula gives:

$S(GRAPH) = C(|LEFT|+|RIGHT|, |LEFT|) * S(LEFT) * S(RIGHT)$

$= C(6, 1) * 1 * 5 = 30$

In summary, the algorithm of computing the number of schedules follows a "divide" and "conquer" strategy. In the "divide" phase, all the nodes with fixed schedules are removed. If the removal leads to disjoint subgraphs, the "conquer" phase computes the number of schedules for each of the subgraphs. If the graph cannot be divided into disjoint subgraphs, the problem is decomposed into multiple subproblems, each of which considers the case when one of the starting nodes is scheduled first. Then, the "conquer" phase solves the subproblems.

4.4.2 Heuristics for Chain Merging

Merging multiple chains into a single chain can reduce the number of register-reuse chains. In the merge of chains, optimization is necessary because there are many possible combinations for selecting the chains to be merged. Thus, this section proposes a heuristic that reduces the search space for selecting the chains to be merged.

In order to reduce complexity, the proposed heuristic merges only a pair of chains at a time. In addition, further reduction is made by additional constraints in the selection of the chains to be merged. In the proposed heuristic, two chains can be merged only if the first node of one chain is adjacent to a node in the other chain, that is, there is an edge incident to the first node of one chain from a node in the other chain. For example, in Figure 4.9 (a), *chain*[0] and *chain*[1] can be merged because the first node of *chain*[1] is *b* that is adjacent to *a* in *chain*[0]. Similarly, *chain*[2] can be merged with *chain*[0]. However, *chain*[4] cannot be merged with *chain*[0] because the first node, *e*, is not adjacent to any nodes in *chain*[0].

Visiting nodes in the dependence graph in BFS order can effectively perform the search for the candidate pairs. When a node is visited, check whether its dependent node is the first node of a separate chain. If it is true, the two chains that include the two nodes are merged. If there are more than one such node, the node that results in the greatest number of schedules is selected. For example, consider the chains in Figure 4.2 (a) again. The node *a* has three dependent nodes, *b*, *c*, and *d*, each of which is the first node of a chain. If the chain with *b* is merged with *a*, the resulting graph has 64 schedules. Note that the additional dependencies from *c* to *b* and from *d* to *b* need to be counted when the number of schedules is calculated. Similarly, if chain *c* is merged, the graph has 8 schedules. If chain *d* is merged, the graph has 8 schedules. Therefore, chain *b* is selected to be merged with chain *a*. The resulting register-reuse chains are shown in Figure 4.9 (b).

```
MERGE_CHAINS( independent_chains, DAG )
{
      /* initialization of merged_chains */
      merged_chains = independent_chains;
      while ( number_of_chains > number_of_registers or
                  all the nodes in DAG are visited ) {
            Node p <- visit DAG in BFS order;
            number_of_chains = merge ( merged_chains, p );
      }
      return merged_chains;
}

int merge ( merged_chains, Node p )
{
      if ( p has multiple dependent nodes ) {
            /* choose one dependent node that is the best choice
*/
            chosen_node = select_best_choice ( p );
            if (chosen_node exists)
                  merge_two_chains (p, chosen_node, merged_chains);
      }
      /* count the number of merged_chains */
      return count_chains ( merged_chains );
}
```

Figure 4.13 Register-reuse chain merge algorithm

```
Node select_best_choice ( Node p )
{
      Compare all the dependent nodes of the node p. Select the
      best dependent node d based on the following criteria:
      (1)   the node d is the head node of a chain
      (2)   the node d can reuse p's register without violating
            data dependence
      (3)   choosing the node d gives maximum number of schedules
      (4)   the chain containing the node d is the shortest chain
      If there are more than one candidate dependent nodes, just
      select one arbitrarily.
      return d;
}

merge_two_chains (p, chosen_node, merged_chains)
{
      /* find chains containing nodes p and chosen_node */
      ref_chain = find_chain ( merged_chains, p );
      dep_chain = find_chain (merged_chains, chosen_node);
      /* find the next node of p in ref_chain */
      next_node = find_next_node ( p );
      /* find the chain between p and next_node in dep_chain */
      temp_chain   =   remove_chain_between_nodes(   p,   next_node,
dep_chain);
      /* check whether the merge violates data dependence */
      if (merge is allowable)
            merge temp_chain with ref_chain;
      generate additional dependencies;
}
```

Figure 4.13 Continued

CHAPTER FIVE

SYSTEMATIC MERGE OF REGISTER-REUSE CHAINS

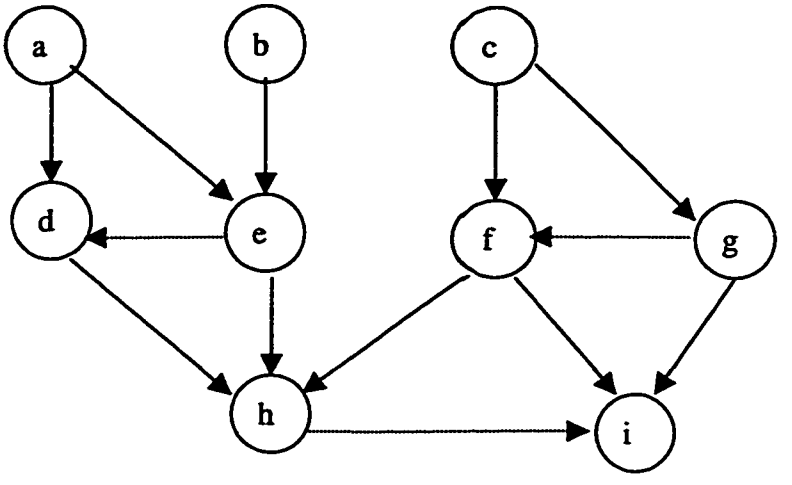
Chapter 4 proposes an optimal register allocation that does not create additional dependencies. The optimal register allocation often requires more registers than those available in a processor. Thus, a simple and intuitive method is proposed to reduce the number of required registers. In this chapter, a systematic method is investigated in order to reduce the resister requirements effectively.

5.1 The Conflict Graph

In order to represent the relationship between register-reuse chains that can be merged or not, a conflict graph is derived from register-reuse chains. In this graph, each node corresponds to a register-reuse chain, and an edge represents that two chains corresponding to the two nodes connected by the edge cannot be merged. The edge has a direction that represents the direction of the merge in which one chain reuses the other chain. If two chains can be merged in only one direction, a unidirectional edge is connected between two nodes. If two nodes cannot be merged in both directions, a bidirectional edge is connected to the corresponding nodes.

Consider the example shown in Figure 5.1. A dependence graph is shown in Figure 5.1 (a), and possible register-reuse chains given in Figure 5.1 (b). The input

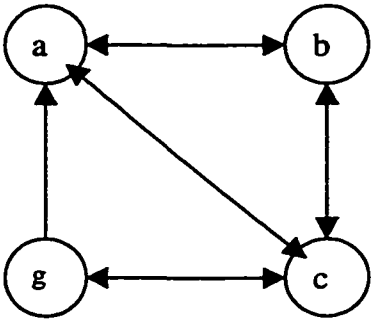
dependencies due to the reuses of registers are shown as dotted lines in Figure 5.1 (a). Figure 5.1 (c) shows the corresponding conflict graph. (The derivation of this graph is explained later.) The character in the node is the first node of the corresponding register-reuse chain. For example, node *a* in the conflict graph represents the register-reuse chain starting with node *a*. A bidirectional edge between *a* and *b* represents that chain *a* and chain *b* cannot be merged. In addition, this figure shows that chain *b* and chain *c*, chain *c* and chain *g*, and chain *a* and chain *c* cannot be merged, respectively. There is a unidirectional edge from g to *a*. This represents that the two chains can be merged, but only in one way: if chain *a* reuses the register assigned to chain *g* after all the computations for chain *g* is completed. There is no edge between *b* and *g*. Thus, *b* and *g* can be merged in both directions.



(a) Dependence graph

Chain[0] = { a, d, h}
Chain[1] = {b, e}
Chain[2] = {c, f, i}
Chain[3] = {g}

(b) Register-reuse chains



(c) Corresponding conflict graph

Figure 5.1 Dependence graph and conflict graph

Once the conflict graph is derived, it is relatively easy to decide the chains to be merged. In order to draw the conflict graph from given register-reuse chains, it is necessary to analyze whether the register-reuse chains can be merged or not. For the analysis, all possible relationships between register-reuse chains are investigated and then analyzed to determine whether they can be merged or not. For the simplicity of analysis,

only the merge of two chains is considered first. In addition, the merge of compact chains (defined below) is analyzed. General cases, such as the merge of multiple noncompact chains, are addressed in the next section.

Definition 5.1 A register-reuse chain is compact if all adjacent nodes in the register-reuse chain are also adjacent in the dependence graph.

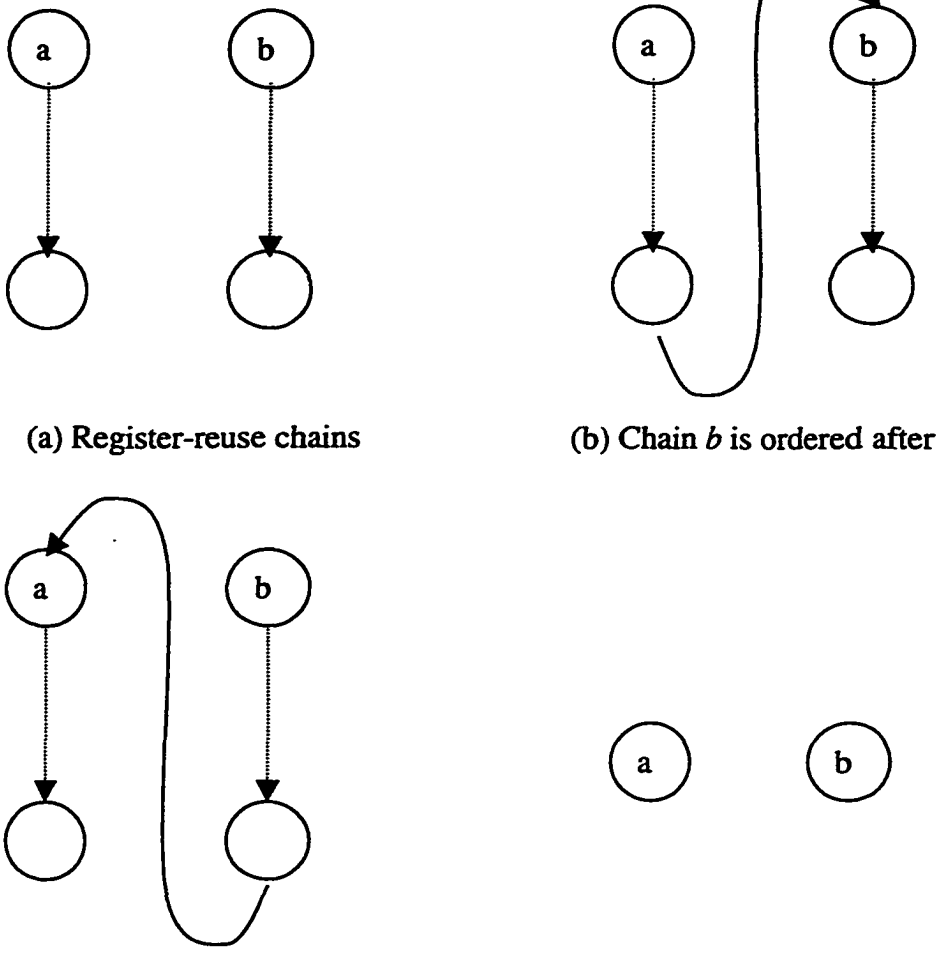
Consider the register-reuse chains of Figure 4.9 (b) again. *chain*[0] contains nodes *a, b*, and *l*. *chain*[0] is compact because *b* is adjacent to *a* in the dependence graph and *l* is adjacent to b in the dependence graph. *chain*[1] contains nodes *c, f, g, h*, and *k*. *chain*[1] is not compact because *k* is adjacent to *h* in the chain, but not adjacent in the dependence graph. *chain*[2] contains nodes *d* and *i*. This chain is not compact because nodes *d* and *i* are not adjacent in the dependence graph. Similarly, *chain*[3] is not compact because *j* is not adjacent to *e* in the dependence graph.

A compact chain does not contain any dead period. Thus, the register assigned to the compact chain cannot be reused by other variables until the last nodes in the chain are computed. However, the register assigned to a noncompact chain can be used by other variables before the last node in the chain is computed. This is possible because the noncompact chain has a dead period. For example in Figure 4.9 (b), the register assigned to *chain*[3] can be reused by other variables or chains after the completion of *e* and before the computation of *j*.

In Section 5.1, a chain is always assumed to be compact. This assumption simplifies the analysis of the possibility of chain merging. Note that compact chain does not include any dead period from the first node to the last node. Therefore, it is impossible to insert any node in the middle of a compact chain. If a new node is included

in this chain, it must be included either before the first node of the chain, or after the last node. Therefore, if two compact chains are merged, the only possibility is that the last node of one chain is connected to the first node of the other chain. Depending on the chains to be put first, there are two ways (or directions) to merge two compact chains. Until explicitly specified in Section 5.2, all chains are assumed to be compact. The merge of noncompact chains is explained later in Section 5.2.

Consider the first case as shown in Figure 5.2. There are two register-reuse chains, each of which starts with nodes *a* and *b*, respectively. Only the starting node and the ending nodes are shown in this register-reuse chain. The dotted line represents a path between two nodes. This graph shows the case when there is no path connecting these two chains. In this case, there are two ways to merge these two chains. One way is to order chain *a* before chain *b* as shown in Figure 5.2 (b). The other is to order chain *b* before chain *a* as shown in Figure 5.2 (c). So it is possible to merge the two chains in both ways. Therefore, there is no hazard in merging these two chains, and the corresponding conflict graph has no edges between the two chains as shown in Figure 5.2 (d).



(a) Register-reuse chains

(b) Chain *b* is ordered after chain *a*

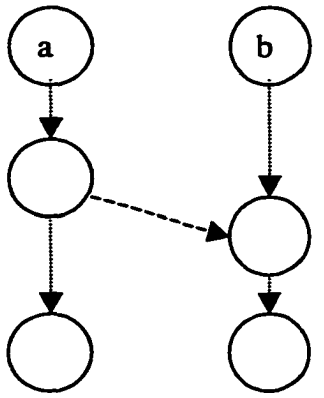
(c) Chain *a* is ordered after chain *b*

(d) Corresponding conflict graph

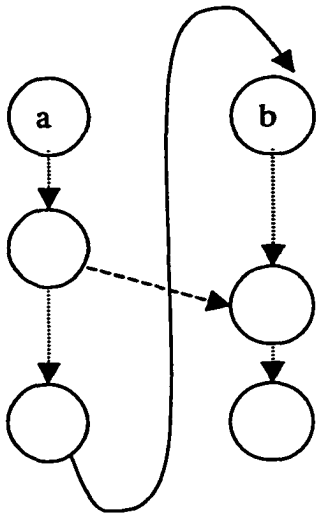
Figure 5.2 No dependence between register-reuse chains

Consider the second case as shown in Figure 5.3. This is the case when there is a path from one chain to another chain. Note that the path is shown in a dashed line that represents a path on which there are multiple nodes. In this case, the merge as shown in Figure 5.3 (b) is possible. However, the merge as shown in Figure 5.3 (c) is always impossible. This is because the created dependency due to merge violates the existing

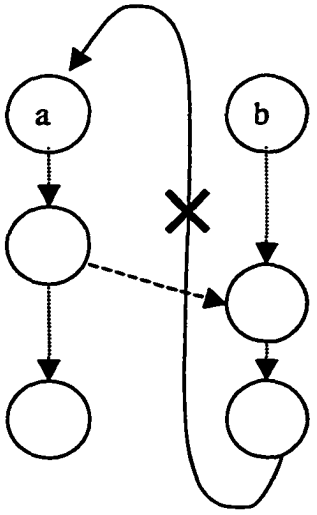
dependency. In other words, the dependence graph makes a cycle. In order to represent the impossibility of chain merging from chain *b* to chain *a*, the conflict graph contains edge from node *b* to node *a*. This is shown in Figure 5.3 (d).



(a) Register-reuse chains



(b) Chain *b* is ordered after chain *a*

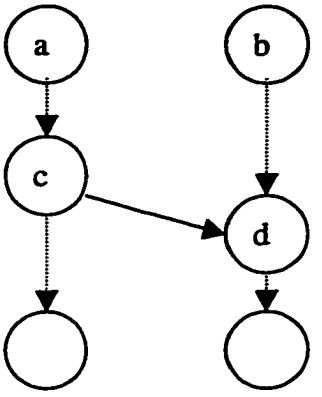


(c) Chain *a* is ordered after chain *b*

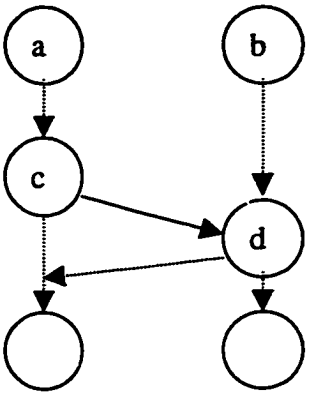
(d) Corresponding conflict graph

Figure 5.3 Unidirectional path from chain *a* to chain *b*

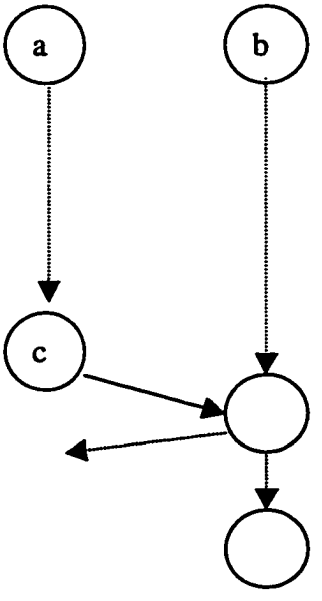
Consider a special case of the example in the above figure. Suppose that the path from chain *a* to chain *b* is a single edge, that is, the node in chain *b* is a successor of the node in chain *a*. This case is shown in Figure 5.4. Note that the path from chain *a* to chain *b* is drawn in a solid line that represents a single edge. Let *c* and *d* denote the two nodes that are connected by the edge. In this case, there is always a path from chain *b* to chain *a*. This is because whenever the register of *c* is reused by other nodes, a new dependence is created from all the successors of c to the node that reuses the register. So the input dependence is shown in Figure 5.4 (b). This case belongs to the third case when there are paths in both directions between the two chains. The only exception is the case as shown in Figure 5.4 (c). In this case, node *c* is the last node of chain *a*. Thus, the input dependence does not go back to chain *a*.



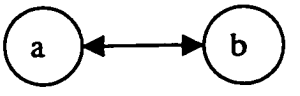
(a) Register-reuse chains



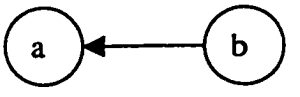
(b) Input dependence from *d* to chain *a*



(c) The last node of chain *a* is adjacent to chain *b*



(d) Conflict graph for (b)



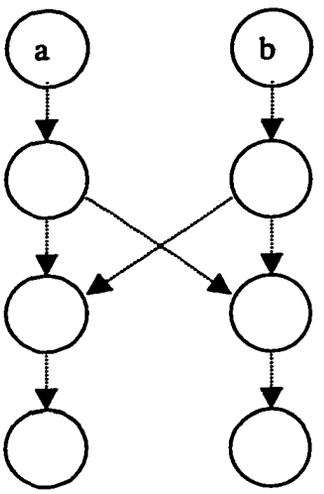
(e) Conflict graph for (c)

Figure 5.4 Chain *b* is adjacent to chain *a*

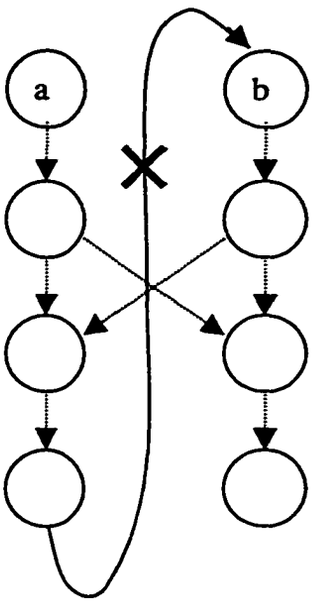
Consider the case when there are paths in both directions between two chains. This case is shown in Figure 5.5 (a). In this case, chain merging is impossible in both directions. This is shown in Figure 5.5 (b) and Figure 5.5 (c). In both figures, the chain merging creates a new dependence that makes a cycle in the graph. Note that the cycle implies that dependence is violated. Therefore, the corresponding conflict graph needs to have a bidirectional edge that represents merging as impossible in both directions.

Consider the case when there are paths from two chains crossing later. First, consider the case shown in Figure 5.6 (a). In this case, a path from chain *a* crosses to the adjacent node of the last node of chain *b*. In this case, the merge of chain *a* before chain *b* is possible (Figure 5.6 (b)), but the merge in the opposite direction violates the existing dependence (Figure 5.6 (c)). This is due to the created input dependence as a dotted line in the figure. Note that a cycle in the dependence graph is made due to the input dependence. The corresponding conflict graph is shown in Figure 5.6 (d).

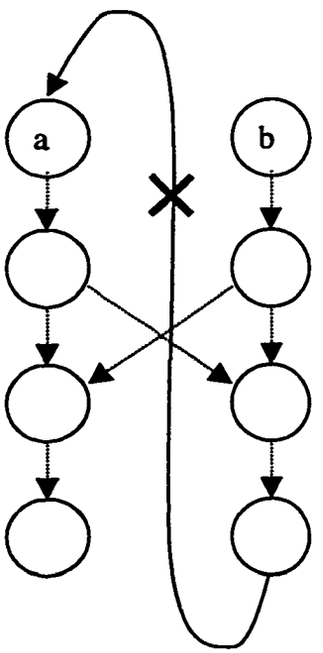
Consider the other cases for the paths from two chains cross each other. Figure 5.7 (a) shows the case when the path from chain *a* crosses a successor of chain *b*. In this case, however, the successor is not the successor of the last node of chain *b*. In this case, merging in both directions is possible. Consider the case when chain *a* is merged before chain *b*. Figure 5.7 (b) shows the created new input dependence. Note that there is no cycle in this graph. So, the merging is possible. Figure 5.7 (c) shows the corresponding conflict graph that has no edge between the two nodes.



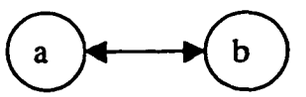
(a) Register-reuse chains



(b) Chain *b* is ordered after chain *a*

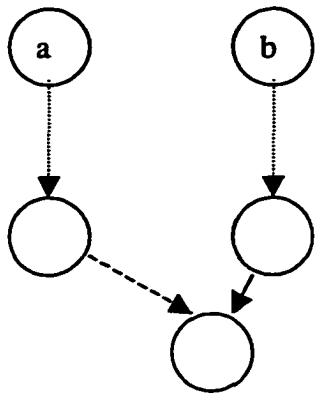


(c) Chain *a* is ordered after chain *b*

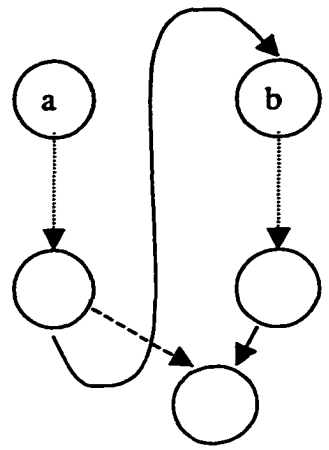


(d) Corresponding conflict graph

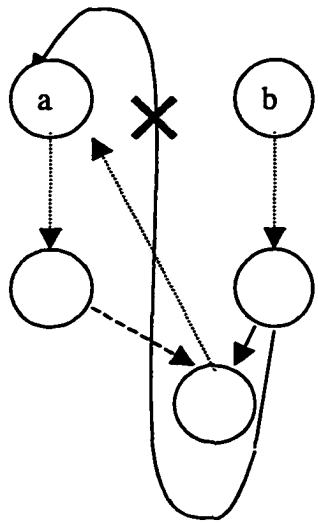
Figure 5.5 Bidirectional path from chain *a* to chain *b*



(a) Register-reuse chains



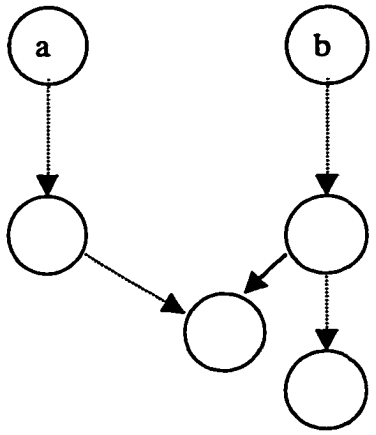
(b) Chain *b* is ordered after chain *a*



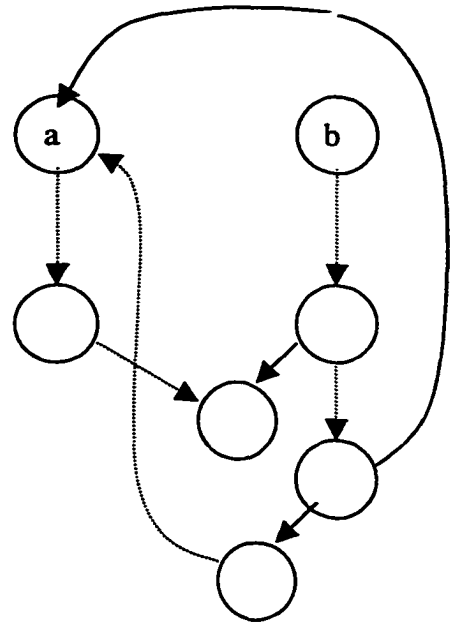
(c) Chain *a* is ordered after chain *b*

(d) Corresponding conflict graph

Figure 5.6 A path from chain *a* crosses a successor of chain *b*



(a) Register-reuse chains



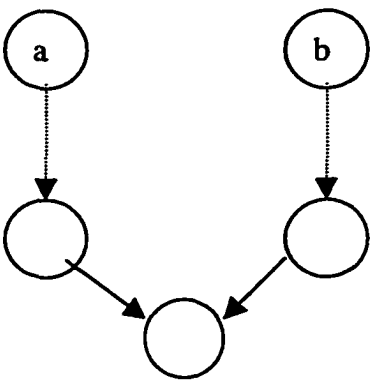
(b) Chain *a* is ordered after chain *b*

(c) Corresponding conflict graph

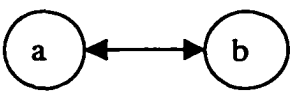
Figure 5.7 A path from chain *a* crosses a successor of an intermediate node of chain *b*

Consider the other cases for the paths from two chains that cross each other. Figure 5.8 (a) shows the case when the successor of the last node of chain *a* is also a successor of the last node of chain *b*. In this chain, merging is impossible in both directions. Figure 5.8 (b) shows the corresponding conflict graph. Figure 5.8 (c) shows the case when a successor of an intermediate node of chain *a* is the same as a successor of an intermediate node of chain *b*. In this case, merging is possible in both directions. So, the corresponding conflict graph does not have any edge (Figure 5.8 (d)). Figure 5.8 (e)

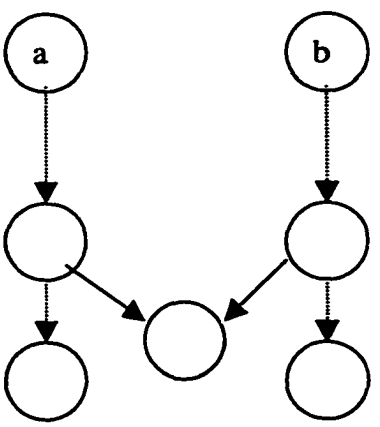
shows the case when a path from chain *a* crosses a path from chain *b*. Note that this path is not a single edge. In this case, two chains can be merged in both directions. So, the corresponding conflict graph is shown in Figure 5.8 (f). The last case is shown in Figure 5.8 (g). Here, there is a node from which two paths go to both chain *a* and chain *b*. The incoming paths do not prevent chain merging. So, the corresponding conflict graph has no edge between the two nodes (Figure 5.8 (h)).



(a) Register-reuse chains



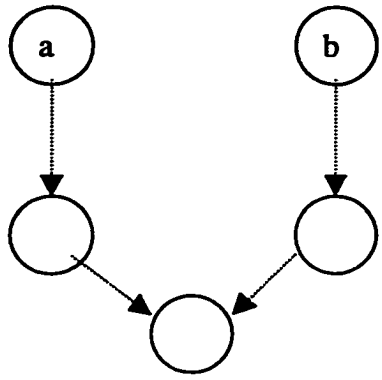
(b) Conflict graph for (a)



(c) Register-reuse chains

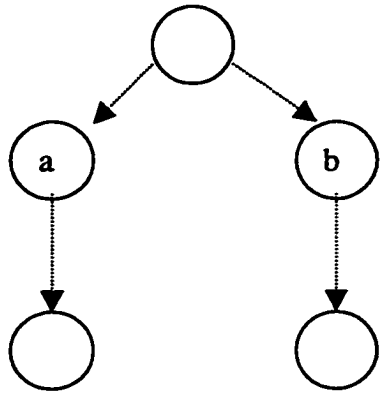
(d) Conflict graph for (c)

Figure 5.8 Various cases for paths between chain *a* and chain *b*



(e) Register-reuse chains

(f) Conflict graph for (e)



(g) Register-reuse chains

(h) Conflict graph for (e)

Figure 5.8 Continued

Consider the example shown in Figure 5.1 again. Chain *a* and chain *b* have paths in both directions. The edge from *a* to *e* forms a path from chain *a* to chain *b*, while the input dependence from *e* to *d* forms another path from chain *b* to chain *a*. Therefore, the corresponding nodes in the conflict graph have the bidirectional edge. Chain *a* and chain *c* also have paths in both directions. The path $h \rightarrow i$ goes from chain *a* to chain *c*, while the path $f \rightarrow h$ goes from chain *c* to chain *a*. Thus, the corresponding nodes have the bidirectional edge. Similarly, chain *b* and chain *c* have paths in both directions, and chain *g* and chain *c* have paths in both directions. Thus, the corresponding nodes have bidirectional edges, respectively. From chain *g* to chain *a*, there is a path, $g \rightarrow f \rightarrow h$. However, there is no path from chain *a* to chain *g*. Thus, the corresponding nodes have a unidirectional edge from *g* to *a*. There is no path between chain *b* and chain *g*. Thus, these two chains can be merged in both directions, and the corresponding nodes do not have an edge connecting them.

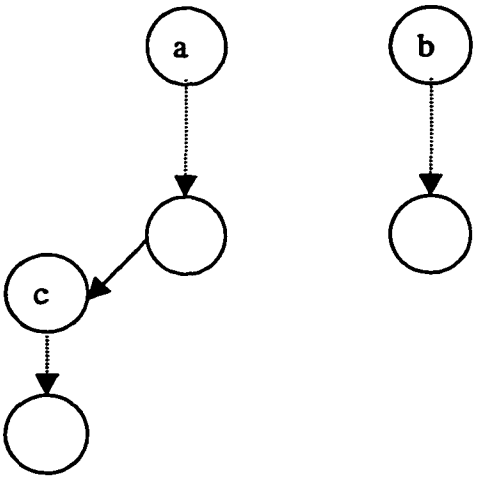
5.2 Merging Algorithm

Two register-reuse chains can be merged if the corresponding nodes in the conflict graph do not have a bidirectional edge. Once the conflict graph is driven, it is easy to check whether two chains can be merged or not. However, even with the given conflict graph, it is not trivial to check whether more than two nodes can be merged or not. This is because the edges in the conflict graph can be changed when two chains are merged. So, when two nodes are merged, it is necessary to analyze the relationship between chains again, and redraw the conflict graph.

Consider the dependence graph shown in Figure 5.9 (a) and the register-reuse chains in Figure 5.9 (b). Figure 5.9 (c) shows the corresponding conflict graph. This figure shows that chain *a* and chain *b* as well as chain *b* and chain *c* can be merged both directions. Chain *a* and chain *c* can be merged in only one direction. Suppose that chain *a* and chain *b* are merged as shown in Figure 5.9 (d). Then, the new dependencies are generated as shown in Figure 5.9 (e). Note that the new input dependency from chain *c* to chain *b* is created. So, the new dependency prevents chain *c* from being merged from chain *b* in both directions. Instead, only one direction is possible. Figure 5.9 (f) shows the corresponding conflict graph. Node *a-b* represents the chain resulting from the merge of chain *a* and chain *b*. In Figure 5.9 (e), chain *a-b* and chain *c* have paths in both directions; a bidirectional edge is necessary between these two nodes.

Definition 5.2 A merged conflict graph is a graph derived from a conflict graph as follows:

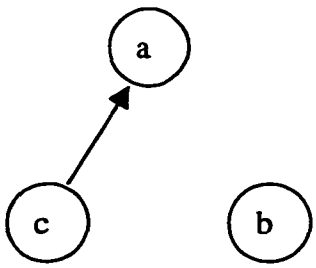
- Multiple nodes $\{v_1, v_2, \ldots, v_m\}$ in the conflict graph can be merged into a single node in the merged conflict graph.
- Suppose that nodes $\{v_1, v_2, \ldots, v_m\}$ are merged into a single node s_1 in the merged conflict graph, and nodes $\{u_1, u_2, \ldots, u_n\}$ are merged into a single node s_2 in the merged conflict graph. Then, the conflict graph has an edge from s_1 to s_2 only if there is an edge from one of $\{v_1, v_2, \ldots, v_m\}$ to one of $\{u_1, u_2, \ldots, u_n\}$. In addition, the conflict has an edge from s_2 to s_1 only if there is an edge from one of $\{u_1, u_2, \ldots, u_n\}$ to one of $\{v_1, v_2, \ldots, v_m\}$.



(a) Dependence graph

Chain *a*: a→ - - →a'
Chain *b*: b→ - - →b'
Chain *c*: c→ - -→c'

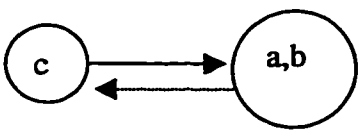
(b) Register-reuse chains



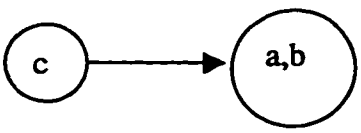
(c) Conflict graph corresponding to (a)

Chain *a,b*: a→ - - →a' →b →- - →b'
Chain *c*: c→ - - →c'

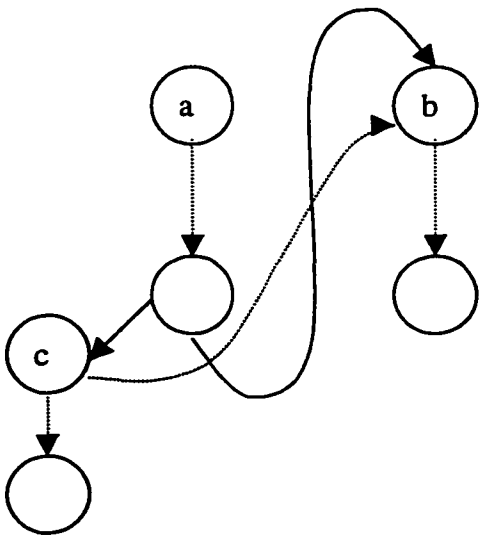
(d) Merged register-reuse chains



(f) Conflict graph after merge



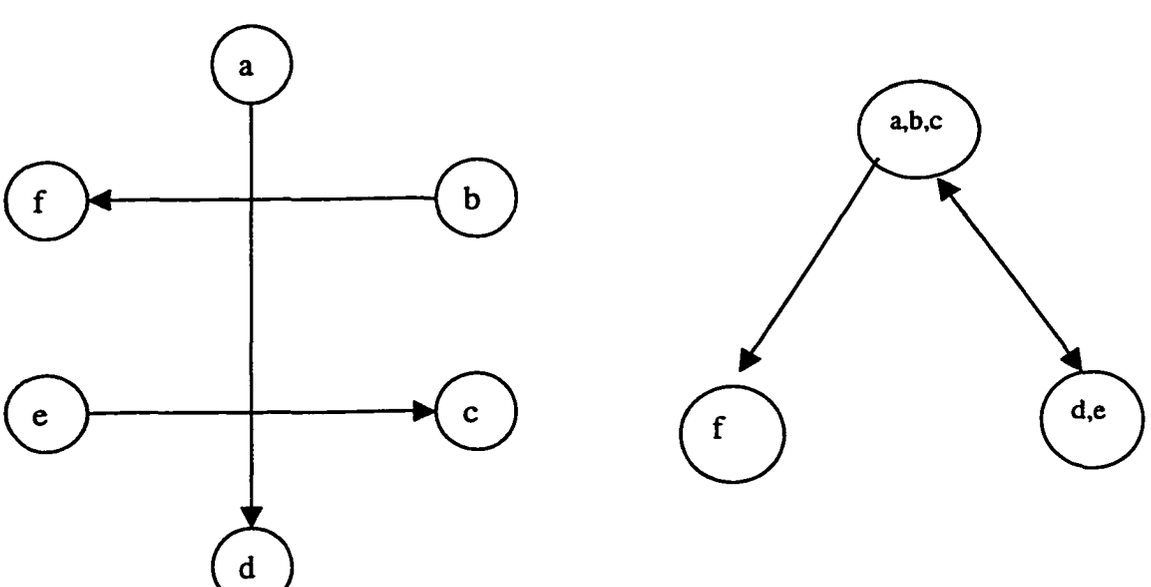
(g) Merged conflict graph



(e) Dependence graph after merge

Figure 5.9 Conflict graph after chain merge

Consider the conflict graph shown in Figure 5.10 (a). Suppose that chain *a*, chain *b*, and chain *c* are merged as well as chain *d* and chain *e* are merged. The resulting merged conflict graph is shown in Figure 5.10 (b). Node *a-b-c* results from the merge of nodes a, *b*, and *c*, while node *d-e* results from the merge of nodes *d* and *e*. A bidirectional edge is necessary between nodes *a-b-c* and *d-e* because the original graph has edge from *a* to *d* as well as *e* to *c*. A unidirectional edge from *a-b-c* to *f* is necessary because of the edge from *b* to *f*. No edge is necessary between *d-e* and *f* because there is no edge between *d* and *f* as well as *e* and *f*.



(a) Original conflict graph (b) Merged conflict graph

Figure 5.10 Merged conflict graph

Figure 5.11 shows two ways to generate the merged conflict graph. In this figure, RRC, CG, M-RRC, and M-CG represent register-reuse chains, a conflict graph, merged register-reuse chains, and a merged conflict graph, respectively. There are two paths from the register-reuse chains to the merged conflict graph. One path goes through M-RRC while the other goes through CG. The first path via M-RRC represents that the merged register-reuse chains are generated first, and then the corresponding conflict graph is generated. The other path represents that the conflict graph of the original reuse chains is generated first, and then the merged conflict graph is generated. The first path always generates the right merged conflict graph. However, this path requires additional computation because the computation for the derivation of the original conflict graph is wasted for the derivation of M-CG. The second path is more computationally effective. In addition, there is another advantage that is explained later. Unfortunately, the merged conflict graph is not always correct. Consider the register-reuse chains shown in Figure 5.9 again. The conflict graph shown in Figure 5.9 (f) is the one following the first path in Figure 5.11. If the conflict graph is derived following the second path, the resulting conflict graph is shown in Figure 5.9 (g). Note that there is only unidirectional edge because the original conflict graph (Figure 5.9 (c)) has an edge neither from *c* to *a* nor from *c* to *b*. Note that this merged conflict graph is wrong because it is different from the one in Figure 5.9 (f).

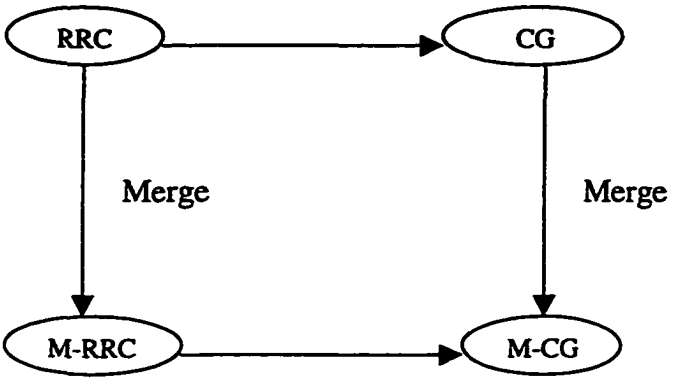


Figure 5.11 Generation of merged conflict graph

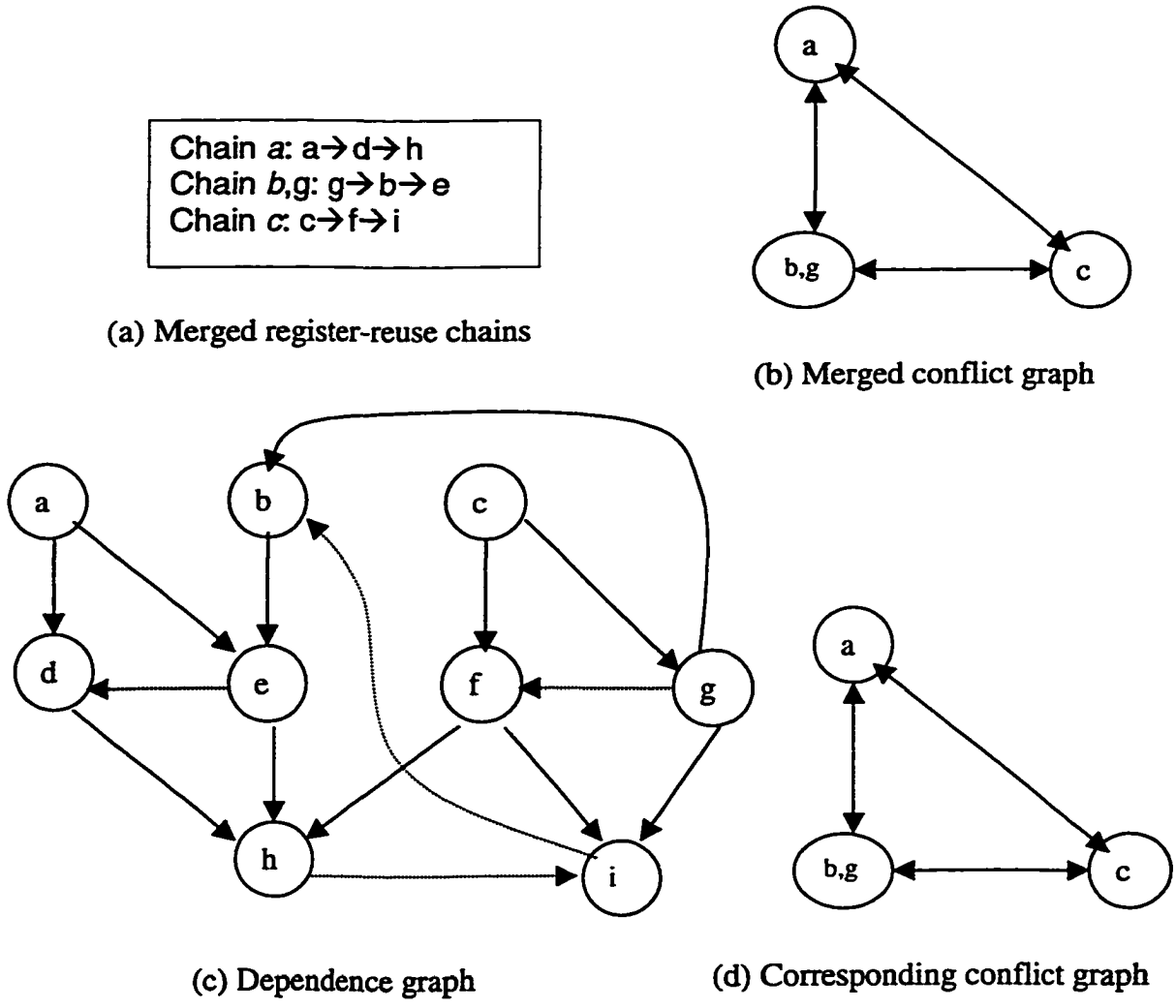
The above example shows that, in general, the merged conflict graph cannot be directly driven from the original conflict graph. However, there is a special type of register-reuse chain (and the corresponding conflict graphs) that allows the direct derivation of the merged conflict graph. In this research, such register-reuse chains, called dependence-conservative register-reuse chains, are formally defined as follows:

Definition 5.3 (Dependence-conservative register-reuse chains) Register-reuse chains are called dependence-conservative if they satisfy the following:

- when sets of chains are grouped and each set is merged into a single chain, the corresponding conflict graph can be derived by merging the corresponding nodes of original conflict graph.

Consider the example in Figure 5.1 again. Suppose that chain *b* and chain *g* are merged. Figure 5.12 (a) shows the corresponding register-reuse chains. The merged conflict graph based on Definition 5.2 is shown in Figure 5.12 (b). Figure 5.12 (c) shows the additional dependencies created by the merge. The corresponding conflict graph is shown in Figure 5.12 (d). Note that this graph is the same as the conflict graph shown in

Figure 5.12 (b). This shows that the register-reuse chains are dependence-conservative register-reuse chains.



(a) Merged register-reuse chains

(b) Merged conflict graph

(c) Dependence graph

(d) Corresponding conflict graph

Figure 5.12 Dependence-conservative register-reuse chains

The dependence-conservative register-reuse chains can reduce the complexity of the derivation of the conflict graph after merging register-reuse chains. Another important property of the dependence-conservative register-reuse chains is that the merge of more than two chains can be easily shown in the corresponding conflict graph. The following

theorem shows how to check whether register-reuse chains can be merged or not from the conflict graph:

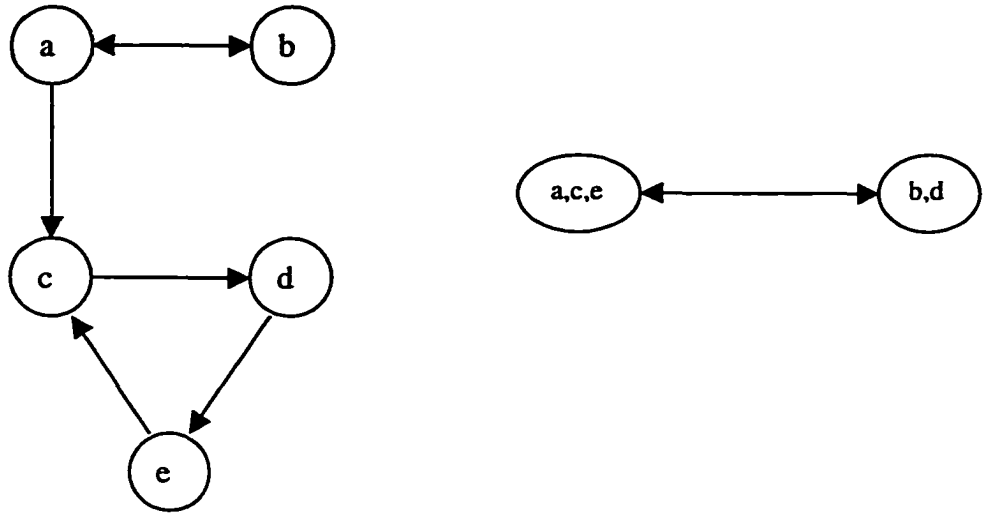
Theorem 5.1 In dependence-conservative register-reuse chains, a set of chains can be merged if the corresponding nodes in the conflict graph do not have edges making a cycle.

(Proof by induction) Suppose that a set of nodes $\{v_1, v_2, \ldots, v_n\}$ does not have edges making a cycle. Then, the first two nodes v_1 and v_2 can be merged because the connecting edge must be not bidirectional if it exists. Assume that n-1 nodes $v_1, v_2, \ldots, v_{n-1}$ can be merged. Consider the new conflict graph resulting from the merge of $v_1, v_2, \ldots, v_{n-1}$. Let s denotes the merged node in the new conflict graph. Then, the edge from s to v_n must be not bidirectional if it exists. Otherwise, the set of nodes $\{v_1, v_2, \ldots, v_n\}$ have edges forming a cycle. Therefore, s and v_n can be merged. Thus, nodes $v_1, v_2, \ldots, v_n$ can be merged.

Corollary 5.1 In dependence-conservative register-reuse chains, multiple sets of nodes can be merged simultaneously if each set does not have edges making a cycle.

(Proof) From Theorem 5.1, one such set can be merged. The merge does not create any additional dependence. Thus, it does not affect the merge of other sets. Thus, other sets can also be merged.

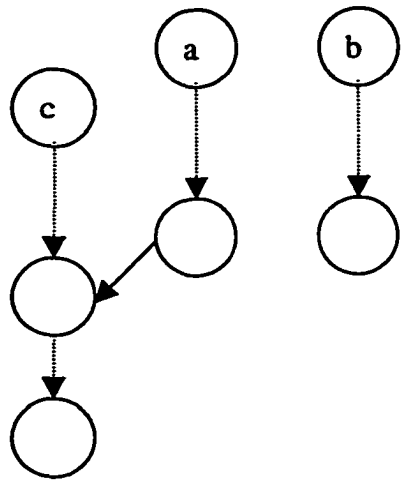
Consider the following graph as shown in Figure 5.13 (a). Nodes *a* and *b* cannot be merged because they are connected as a bidirectional edge. Nodes *c, d,* and *e* cannot be merged into a single node because they make a cycle. All the other nodes can be merged. Figure 5.13 (b) shows an example of possible merges. Nodes *a, c,* and *e* are merged into a single chain, while nodes *b* and *d* are merged into another chain.



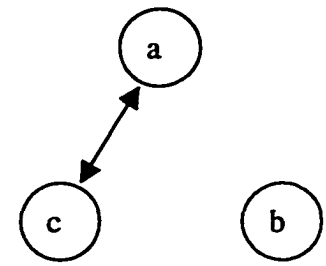
(a) Directed conflict graph (b) An example of a merged conflict graph

Figure 5.13 Chain merge based on Theorem 5.1 and Corollary 5.1

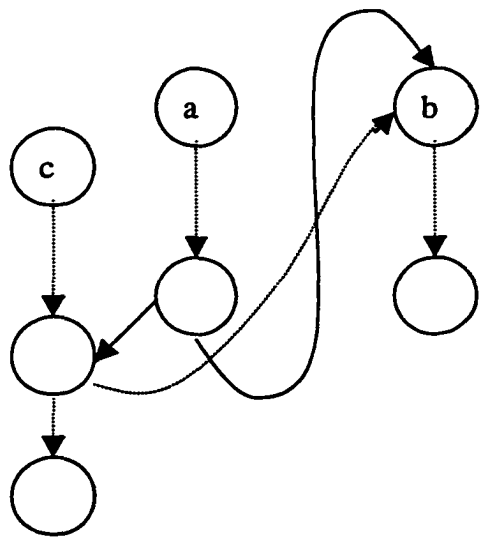
In order to use the results of Theorem 5.1 and Collorary 5.1, register-reuse chains must be dependence-conservative. So, we must consider how to convert register-reuse chains to be dependence-conservative. First, consider the case when the merge of two chains changes the resulting conflict graph. In fact, the example shown in Figure 5.9 is the only possible case, that is, the first node of chain *c* is adjacent to the last node of chain *a*. Suppose it is other than the first node of chain *c*. This case is shown in Figure 5.14. Note that an intermediate node of chain *c* is adjacent to the last node of chain *a*. The corresponding conflict graph is shown in Figure 5.14 (b). Note that chain *a* and chain *c* cannot merge so that there is a bidirectional edge between *a* and *c*. Figure 5.14 (c) shows the merge of two chains *a* and *b*, and the creation of the new input dependence from chain *c* to chain *b*. The resulting conflict graph is shown in Figure 5.14 (d).



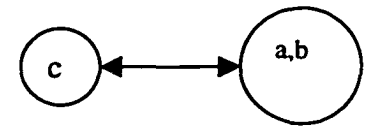
(a) Dependence graph



(b) Conflict graph



(c) Dependence graph after merging



(d) Conflict graph after merging

Figure 5.14 Generation of the merged conflict graph

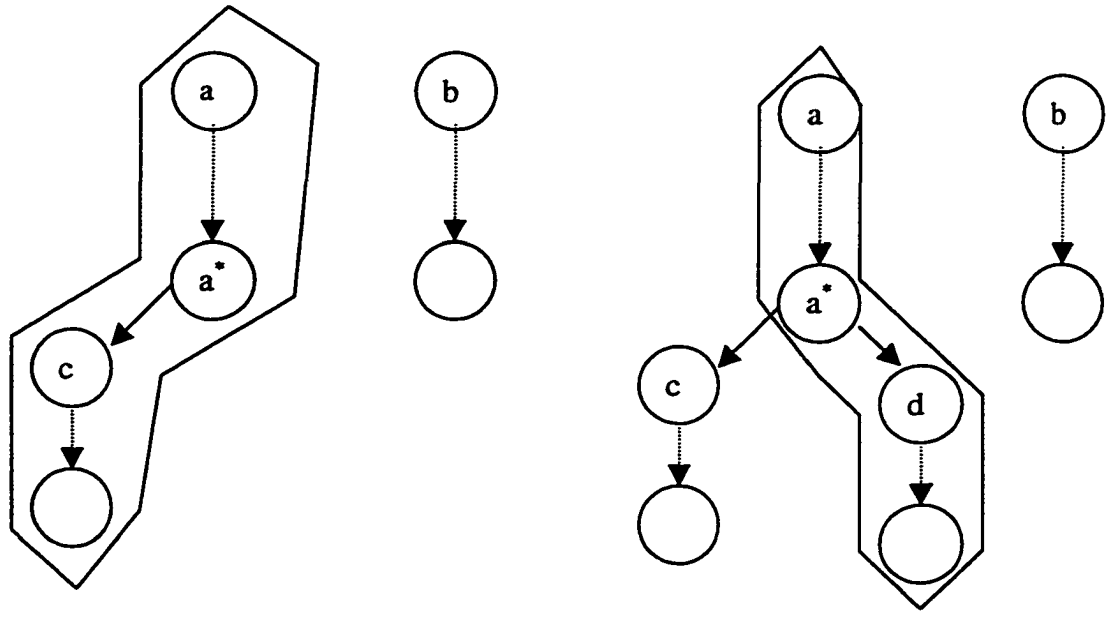
So far, the merging of compact chains has been the considered case. Noncompact chains can be considered as merged compact chains. Note that Theorem 5.1 and Corollary 5.1 are true for the merged compact chains. Therefore, they are applicable to noncompact chains.

Proposition 5.1 The merge procedure, called `init_merge ( )`, discussed in the previous section, generates dependence-conservative registe-reuse chains.

(Proof) The merge procedure, called `init_merge ( )`, discussed in the previous section, removes the case of Figure 5.9. If all the missing parts of dependence graphs are drawn, there are two possibilities. One case is that c is the only successor of the last node of chain a, and the other case is that there are successors other than c. The first case is redrawn in Figure 5.15 (a) while the other case is shown in Figure 5.15 (b). Note that a^* represents the last node in chain a. In the first case, the optimal chain does not allow a^* and c in the different register-reuse chains. Thus, the case shown in Figure 5.9 is impossible. In the second case as shown in Figure 5.15 (b), `init_merge ( )` procedure merges either chain c or chain d with chain a. Without loss of generality, assume that chain d is merged with chain a. Then, again the case is the same as Figure 5.14 (a), but is different from that in Figure 5.9 (a). Therefore, the case like Figure 5.9 (a) is not allowed after `init_merge ( )` procedure is called. Therefore, all register-reuse chains are dependence-conservative.

The analysis for compact chains can be easily generalized for noncompact chains. A noncompact chain can be made by merging compact chains. In order to draw the conflict graph for noncompact chains, the compact chains that make the noncompact

chains are analyzed. If an edge is needed for any compacting chain, an edge is drawn for the noncompact chain.



(a) c is the only successor of a^* (b) Both c and d are successors of a^*

Figure 5.15 Avoidance of the case as shown in Figure 5.9

5.3 Register Allocation Algorithm Based on Coloring of Conflict Graph

This section finalizes the register allocation procedure. Given the conflict graph, register allocation can be formulated as a graph-coloring problem.

Problem 5.1 (Graph coloring problem formulation for register allocation over directed conflict graph) Given a conflict graph, colors are assigned to all the nodes such that:

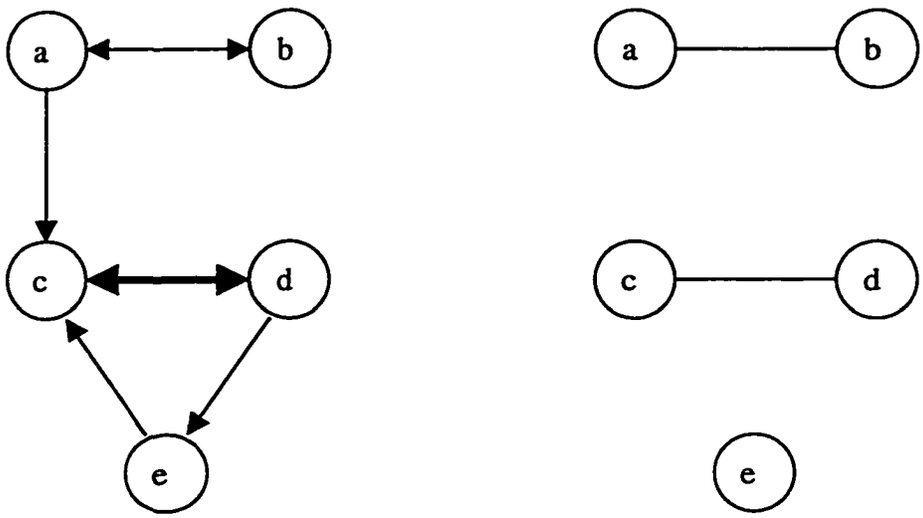
- Constraints: Nodes cannot have the same color if they have connecting edges that make a cycle.
- Objective: Minimize the number of colors.

After the graph-coloring problem is solved, colors are assigned to each node. Register-reuse chains whose corresponding nodes are assigned to the same color are to be merged.

The above graph coloring problem is formulated on the conflict graph that is a directed graph. In order to reduce the search space, the directed graph is converted into an undirected graph. In this conversion, a bidirectional edge in the original directed conflict graph is converted to an edge in the new undirected conflict graph, and a unidirectional edge is removed in the undirected graph. However, if a unidirectional edge is a part of a cycle, one of the unidirectional edges needs to be changed to a bidirectional edge. In this selection, the number of schedules is used as the criterion.

Consider the conflict graph shown in 5.13 (a). Nodes *c*, *d*, and *e* make a cycle. Therefore, these nodes cannot be merged into a single node. Among the three edges making the cycle, one needs to be chosen and converted to a bidirectional edge. In this selection, it is necessary to compute the number of schedules when the corresponding

chains are merged. Then, the chain that maximizes the number of schedules is selected. In this example, assume that the edge between *c* and *d* is chosen and converted to a bidirectional edge (see Figure 15.6 (a)). Now that there is no cycle made only by unidirectional edges, the conflict graph is converted into an undirected graph. All unidirectional edges are removed while bidirectional edges remain as undirected edges. The resulting graph is shown in Figure 5.16 (b).



(a) Edge (*c,d*) is converted to a bidirectional edge

(b) Converted undirected conflict graph

Figure 5.16 Conversion of a directed conflict graph into an undirected conflict graph

Now that an undirected conflict graph is derived, register allocation is formulated as graph coloring problem over the undirected conflict graph:

Problem 5.2 (Graph coloring problem formulation for register allocation over undirected conflict graph) Given a conflict graph, colors assigned to all the nodes such that:

- Constraints: Two nodes cannot have the same color if there is an edge connecting them.
- Objective: Minimize the number of colors.

Although graph coloring is a *NP*-Complete problem, there are many efficient heuristics. The number of colors corresponds to the number of required registers. If the number of colors is less than the number of registers available in a target processor, not all of the chain mergings are necessary. For this case, some of the merged chains can be decomposed into the original separate chains. Recall that chain merging always reduces the number of schedules. Therefore, decomposition of merged chains can increase the number of schedules.

Problem 5.3 (Decomposition of merged chains) Decompose merged chains into the original separate chains.

- Constraint: the number of chains is less than or equal to the number of available registers.
- Objective: maximize the number of schedules after chain merging.

This problem can be converted into a 0-1 knapsack problem. First, define V_i as follows:

V_i = the number of schedules without merging any chains - the number of schedules after merging the chains assigned to the ith color.

W_i = the number of register-reuse chains assigned to the ith color $- 1$.

The decomposition problem can be formulated as the following problem:

Problem 5.4 (0-1 knapsack problem formulation for the decomposition of merged chains) Decompose merged chains into the original separate chains.

- Constraint: Summation of W_i is less than or equal to the number of available registers.
- Objective: Minimize the summation of V_i

The above problem is exactly the same as a 0-1 knapsack problem, and a dynamic programming can be used to solve the problem.

Figure 5.17 summarizes the overall register allocation procedures proposed in this research. Given a dependence graph, the first step is to generate register-reuse chains that are optimal in the sense that no additional dependencies are created. The next step is the initial merge algorithm that combines the optimal register-reuse chains by visiting the nodes in the dependence graph in BFS order. This procedure effectively reduces the number of chains without a significant increase of dependencies. In addition, the resulting reuse chains become dependence-conservative. The next step is to generate a conflict graph that is a directed graph. In order to use the graph-coloring problem, the directed graph is converted into a undirected graph in the next procedure. Once the undirected conflict graph is obtained, the graph-coloring algorithm is performed to assign colors to each chain. The chains that are assigned to the same color are merged into a single chain, and consequently allocated to the same registers. For the case when the number of colors is less than the number available registers, the next procedure is called in order to decompose the merged chains into the original separate chains. For this procedure, the dynamic programming algorithm for solving the 0-1 knapsack problem is used.

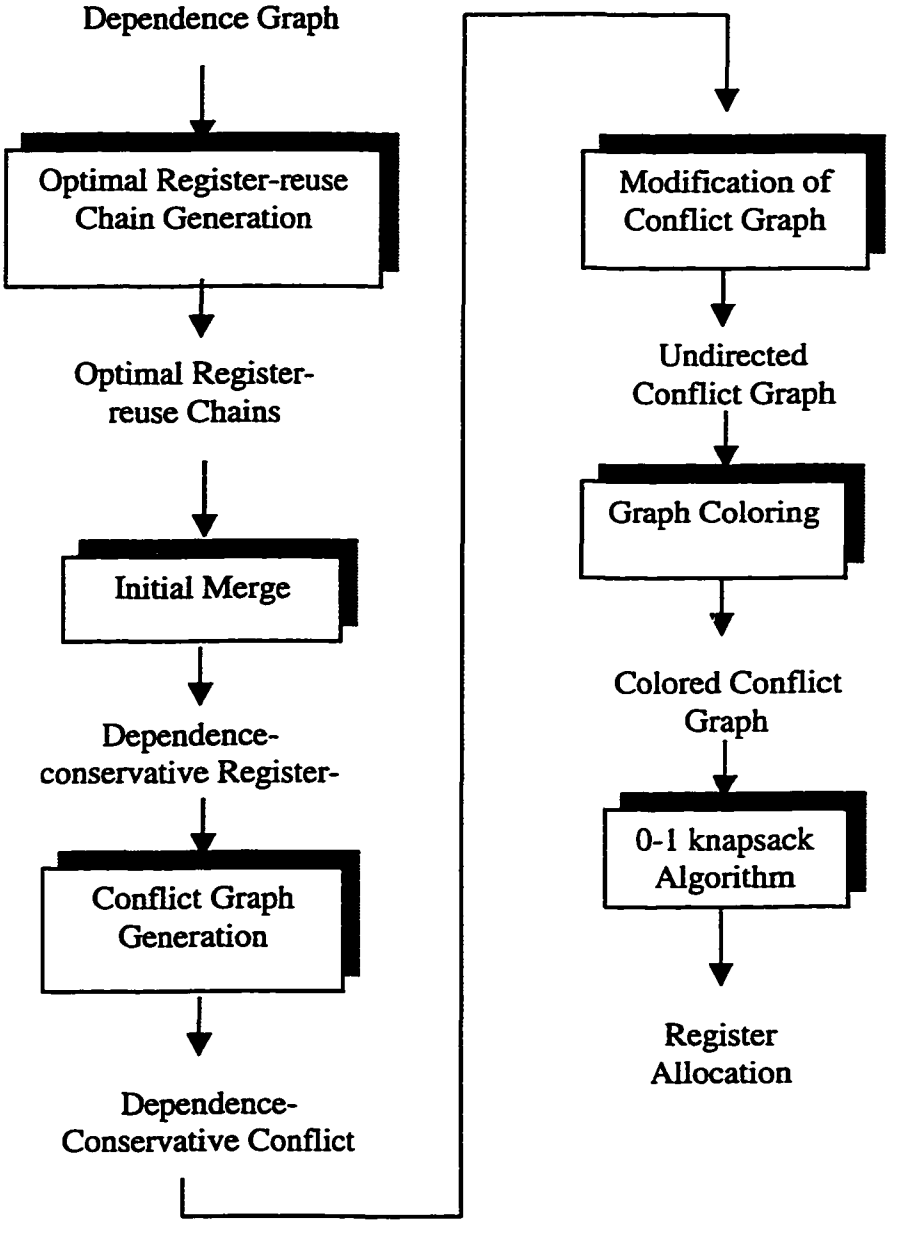


Figure 5.17 Complete procedure for register allocation

CHAPTER SIX

CONCLUSIONS

This research develops a register allocation technique that can be integrated into instruction scheduling. The result of this form of register allocation is more efficient than the traditional register allocation based on the graph-coloring algorithm. The main reason for this is that the proposed technique is performed before instruction scheduling, while the traditional register allocation is performed after instruction scheduling that gives additional constraints to the selection of registers. On the other hand, the proposed technique can add constraints to instruction scheduling when there are not enough registers. As a result, the efficiency of the scheduler can be degraded. The optimal register allocation algorithm developed in Chapter 4 does not create additional dependencies, and consequently, no additional constraints are added to instruction scheduling. Although the number of the registers required resulting from the optimal register allocation is large, it fits into the available registers for most cases. When a processor does not have enough registers, the register allocation algorithm needs to reduce the number of required registers. The heuristic proposed in this research attempts to prevent unnecessary constraints to instruction scheduling while reducing the number of necessary registers. As a result, the proposed heuristic adds fewer constraints than previous approaches. The proposed register allocation analyzes the dependence generated by the register allocation

and attempts to minimize the generated dependencies. Note that instruction scheduling can be considered as a sequence of generation of dependences until all instructions have strictly ordered by the dependencies. The main job of a scheduler is to choose the dependencies that are preferred by a target processor. Thus, the proposed register allocation can be easily integrated with instruction scheduling with a minor modification such that the register allocation attempts to minimize generated dependencies that are not preferred.

REFERENCES

[1] Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[2] J. Allen. Computer Architecture for Digital Signal Processing. *Proceedings of the IEEE*, 73(5): 852-873, May 1985.

[3] *ARM Architecture Reference Manual.* Document Number: ARM DDI 0100B. Advanced RISC Machines Ltd (ARM), 1996.

[4] Guido Araujo and Sharad Malik. Optimal Code Generation for Embedded Memory Non-homogeneous Register Architectures. In *Proceedings of International Symposium on Systems Synthesis*, pp. 36-41, 1995.

[5] Guido Araujo, Srinivas Devadas, Kurt Keutzer, Stan Liao, Sharad Malik, Ashok Sudarsanam, Steve Tjiang, and Albert Wang. Challenges in Code Generation for Embedded Processors. Chapter 3, pp. 48-64, in P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors.* Boston, Mass.: Kluwer Academic Publishers, ISBN 0-7923-9577-8, 1995.

[6] Steve Beaty. *Instruction Scheduling Using Genetic Algorithms.* Ph.D. Thesis. Colorado State University, 1991.

[7] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource Spackling: A Frame Work for Integrating Register Allocation in Local And Global Schedulers. In *proc. of IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, papers 135-146, 1994.

[8] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures. In *proc. of IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243-254, 1993.

[9] D. G. Bradles, S. J. Eggers and R. R. Henry, Integrating Register Allocation and Instruction Scheduling for RISCs, Proc. *Fourth International Conf. On ASPLOS*, Santa Clara, CA, April 8-11, pp.122-131, 1991.

[10] Thomas H. Cormen, Charles E. Lerserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT press, 1989.

[11] Tai Myoung Chung. *CHARTS: Compiler for Hard Real-Time Systems*, Ph.D. Thesis. Purdue University, 1995.

[12] G. T. Chatin. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction*. Vol. 17, No. 6, pp 98-105, June, 1982.

[13] Kevin Dowd. *High Performance Computing*. O'Reilly & Associates, Inc. ISBN 1-56592-032-5, 1993.

[14] J. R. Ellis, *A Compiler for VLIW Architectures*, MIT Press, 1985.

[15] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc., ISBN 0-8053-1670-1, 1995.

[16] Jack G. Ganssle. *The Art of Programming Embedded Systems*. San Diego, CA, Academic Press, Inc., 1992.

[17] Lal George and Andrew W. Appel. Iterated register coalescing. In *ACM Transactions on Programming Languages and Systems*. Vol. 18, No. 3, pp 300-324, May 1996.

[18] J. R. Goodman and W. Hsu, Code Scheduling and Register Allocation in Large Basic Blocks, *Proc. of the ACM Supercomputing Conference*, pp.442-452, 1998.

[19] Rajiv Gupta, Mary Lou Soffa, and Denise Ombres. Efficient Register Allocation via Coloring Using Clique Separators. In ACM Transactions on Programming Languages and Systems. Vol. 16, No. 3, pp 370-386, May 1994.

[20] R. Gupta, Co-synthesis of Hardware and Software for Digital Embedded Systems. Ph.D. Thesis, Stanford University, December 1993.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-329-8, 1996.

[22] Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of Loads/Stores in Local Register Allocation. In *IEEE Transactions on Software Engineering*. Vol. 15, No. 10, pp 1252-1260, Oct., 1989.

[23] K. Kissel. MIPS16: *High-density MIPS for the Embedded Market*, Silicon Graphics MIPS Group, 1997.

[24] Mika Kuulusa, Jari Nurmi, Janne Takala, Pasi Ojala, and Henrik Herranen. A Flexible DSP Core for Embedded Systems. In *IEEE Design & Test of Computers*. October-December, pp. 60-68, 1997.

[25] Jooho Lee, Dae Hwan Kim, Hyuk Jae Lee and Chinhyuan Kim. Advanced Compiler Optimizations for the ARM-7 Processor. In preparation for publication.

[26] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving Code Density using Compression Techniques. In *Proceedings of the 30th International symposium on Microarchitecture*, pp. 234-251, December, 1997.

[27] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjaing, and Albert Wang. Storage Assignment to Decrease Code Size. In *ACM Transactions on Programming Languages and Systems*. Vol. 18, No. 3, pp. 235-253, May 1996.

[28] Stan Y. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD Thesis, MIT Department of EECS, January 22, 1996.

[29] Minjoong Lim. Variable Length-RISC Architecture for the Design of Embedded Systems, technical documents, *Samsung Electronics*, 1997.

[30] Xiaorong Ma. *Nonhomogeneous Register Allocation for Embedded Processors*. M.S. Thesis. Louisiana Tech University, 1998.

[31] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining Register Allocation and Instruction Scheduling. *Technical Report*, Courant Institute, TR 698, July 1996.

[32] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[33] Cindy Norris and Lori L. Pollock. Register allocation over the program dependence graph. In *Proc. of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 106-117, June 1994.

[34] David A. Patternson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, 1994.

[35] P.G. Paulin et al. *Trends in Embedded Systems Technology in Hardware/Software Co-Design.* Kluwer Academic, Norwell, Mass., pp. 311-337, 1996.

[36] S. S. Pinter, Register Allocation with Instruction Scheduling: A New Approach, *Proc. SIGPLAN '93 Conf. On Programming Language Design and Implementation,* Albuquerque, NM, June 23-25, pp. 248-257, 1993.

[37] Todd A. Proebsting and Charles N. Fischer. Demand-Driven Register Allocator. In *ACM Transactions on Programming Languages and Systems.* Vol. 18, No. 6, pp. 683-710, Nov., 1996.

[38] Simon Segars. ARM7TDMI Power Consumption. In *IEEE MICRO.* Vol. 17, No. 4, pp.12-19, July/August 1997.

[39] Robin Saxby. ARM moves forward with multimedia. *Multimedia Silicon.* Volume 1, Issue 7. June 11, 1997.

[40] K. L. Short, *Embedded Microprocessor Systems Design*, Prentice-Hall, 1998.

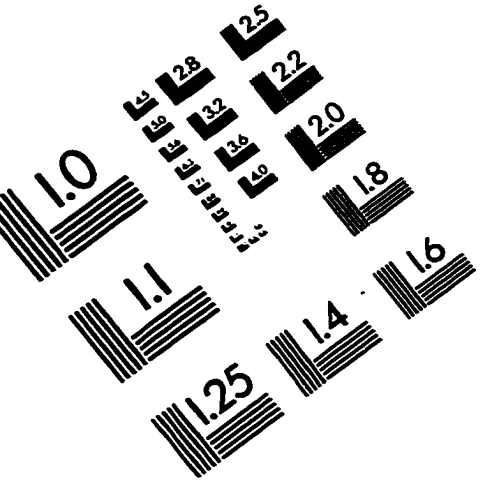
[41] Alex van Someren and Carol Atack. *The ARM RISC Chip: A Programmer's Guide*. The University Press, Cambridge, UK, 1993

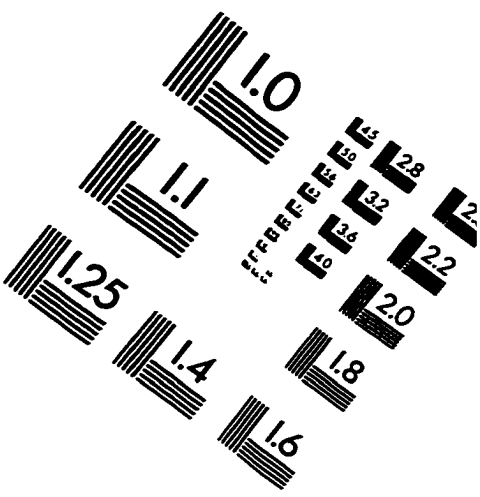
[42] Ashok Sudarsanam, *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* Ph.D. Thesis, Princeton University, Nov. 1998.

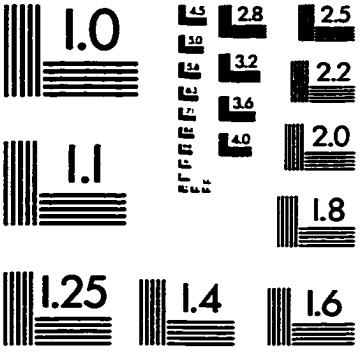
[43] A. Wolfe and A. Chanin, Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December, 1992.

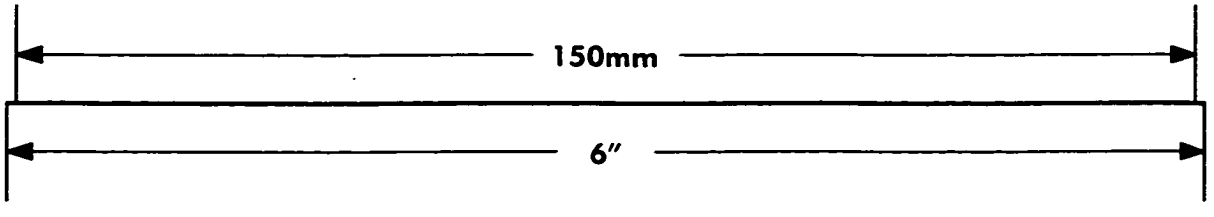
[44] Jorg Wilberg, *Codesign for Real-Time Video Applications.* Kluwer Academic Publishers, Dordrecht, Netherlands, 1997.

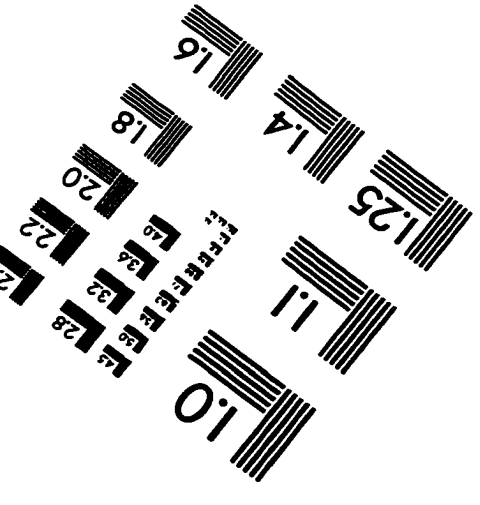
IMAGE EVALUATION
TEST TARGET (QA-3)











APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989