論 文
48A-12-12

레지스터 재활용 사슬의 체계적 생성

A Systematic Generation of Register-Reuse Chains

李 赫 宰*
(Hyuk-Jae Lee)

Abstract - In order to improve the efficiency of optimizing compilers, integration of register allocation and instruction scheduling has been extensively studied. One of the promising integration techniques is register allocation based on register-reuse chains. However, the generation of register-reuse chains in the previous approach was not completely systematic and consequently it creates unnecessarily dependencies that restrict instruction scheduling. This paper 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. Chain merging always generates additional dependencies and consequently enforces the execution order of instructions. A heuristic is developed for the second phase in order to reduce additional dependencies created by merging chains. For matrix multiplication program, the number of registers resulting from the first phase is small enough to fit into available registers for most basic blocks. In addition, it is shown that the restriction to instruction scheduling is reduced by the proposed merging heuristic of the second phase.

Key Words : Optimizing compiler, register allocation, register-reuse chains, dependence analysis

1. Introduction

As the complexity of a processor architecture and organization increases, the impact of compiler optimization on the processor speed becomes more important. Therefore, compiler optimization techniques, especially instruction scheduling and register allocation, have been extensively investigated [1-10]. Instruction scheduling determines the execution order of instructions while register allocation determines which value is to be stored into a processor register and which value is not. If a processor has multiple registers, register allocation also chooses the exact register number.

In most research efforts, instruction scheduling and register allocation are studied separately. However, these two optimizations often make a significant influence on each other. For example, an instruction scheduling decides the live range of a variable and consequently gives significant constraints to register allocation. Therefore, even an efficient instruction scheduling can degrade the overall optimization if the scheduler leads to too many constraints to register allocation resulting in poor register allocation. On the other hand, register allocation affects instruction scheduling because it often creates additional dependencies which add constraints to the scheduler. In order to achieve the best optimization, a compiler needs to adopt a technique which is efficient for both instruction scheduling and register allocation.

In order to develop an optimizing compiler efficient for both a scheduler and a register allocator, recent research has been focused on the integration of these two techniques [7-10]. Berson, Gupta, and Soffa make a promising contribution by proposing register allocation based on register-reuse chains [7,8]. Register-reuse chain is defined as an ordered set of instructions that use the same destination registers. Thus, register allocation in [7,8] is a procedure to decompose a dependence graph into register-reuse chains. Each reuse chain maps to 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 which leads to the reduction of the number of register-reuse chains (see details in Section 2). Since the addition of

* 正 會 員 : Intel Senior Engineer · 工博
接受日字 : 1999年 10月 19日
最終完了 : 1999年 11月 11日

dependencies generates additional restrictions to an instruction scheduler, efficient heuristics is proposed in [7,8] to reduce unnecessary restrictions.

Although the main idea of the register-reuse chain approach is promising, the method proposed by [7,8] 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 possible because the efficiency of the previous heuristic can be degraded 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 proposed register allocation is optimized for general cases that have no restriction on the execution time of a statement.

The register allocation technique proposed in this paper follows the framework in [7,8], and improves the efficiency of the technique. The first step is to find 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 additional dependencies. The proposed register allocation technique is implemented in Local C Compiler (LCC) [3] and the efficiency of the technique is evaluated.

This paper is organized as follows. Section 2 provides a brief explanation of the previous approach for register allocation based on register-reuse chains followed by discussion on possible improvements. Section 3 studies the creation of dependencies due to register allocation, and proposes a register allocation algorithm that avoids the creation of any additional dependencies. Section 4 proposes a heuristic that reduces the number of required registers while attempting to minimize additional dependencies. Section 5 briefly describes the implementation of the proposed register allocation technique, and evaluates the efficiency of the technique. Section 6 concludes the paper.

2. Register Allocation Based on Register-Reuse Chains

This section explains the previous register allocation approach based on register-reuse chains. The main idea is briefly explained with an example, and then possible improvements over the previous approach are discussed.

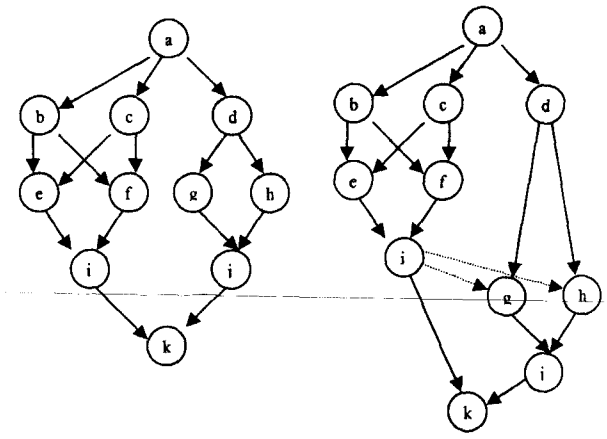
2.1 Previous Approach

In [7], the source code shown in Fig. 2.1 (a) is used to explain the register allocation based on register-reuse chains. Fig. 2.1 (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. Fig. 2.1 (c) shows the register reuse graph derived from the dependence graph. In this graph, 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', i.e., '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 the 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.

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. Fig. 2.1 (d) shows such a graph transformed from Fig. 2.1 (c). By removing edges (d,f), (e,g), (j,k) and (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 Fig., '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].

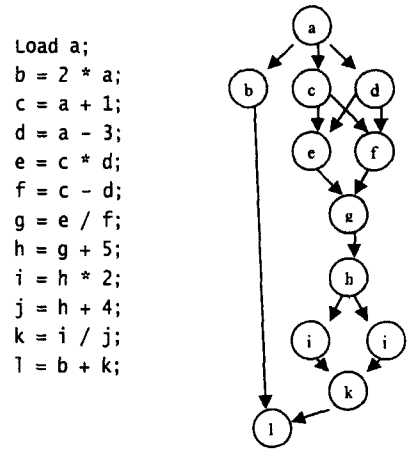
Since each chain is mapped to one register, the number of chains corresponds to the number of necessary registers. For example, the graph in Fig. 2.1 (d) requires four registers. If the number of chains is greater than the number of registers, it is necessary to reduce the number of chains. In order to reduce the number of chains, [8] proposes to add dependencies in the dependence graph. For example, consider a dependence graph as shown in Fig. 2.2 (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 Fig. 2.2 (b). With the new dependencies, [8] can derive new reuse chains of which number is four. More details on the addition of dependencies are explained 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 Fig. 2.2 (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, for the generation of register spill/reload instructions, and optimization across basic blocks. Since the additional optimizations are not the interest of this paper, detailed explanation is omitted.

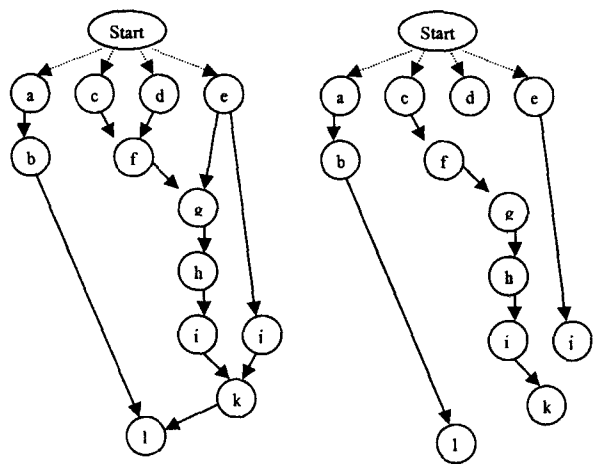


(a) Dependence graph (b) Additional dependencies shown in dotted line

Fig. 2.2. Creation of dependencies for register allocation



(a) Source code (b) Dependence graph



(c) register-reuse graph (d) register-reuse chains

Fig. 2.1 Register allocation example given in [8]

2.2 Possible Improvements

In this paper, 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 a killing node means the node that can reuse the register assigned to the predecessor. The previous research does not have a systematic approach to select a killing node, and consequently can degrade the efficiency of a compiler. For example, consider the dependence graph shown in Fig. 2.3.

Suppose that 'b' is selected to reuse the register for 'a'. This enforces 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 enforce 'c' and 'd' to be executed immediately after the execution of 'a'. Then, node 'b', 'e',

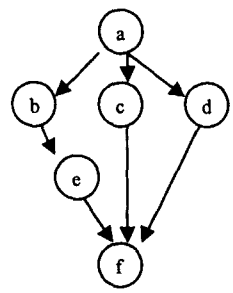


Fig. 2.3 Example of dependence graph

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' needs 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 exist more than one choices 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 different number of operations that can further differentiate the execution time of a statement. This paper proposes a new criterion for the reduction of register-reuse chains. Based on the new criterion, a heuristic is proposed to reduce register-reuse chains. The new heuristic is designed to be used efficiently for the general case when different statements can have different execution times.

3. Register-Reuse Chain and Dependence Analysis

3.1 Longest Possible Live Range Analysis

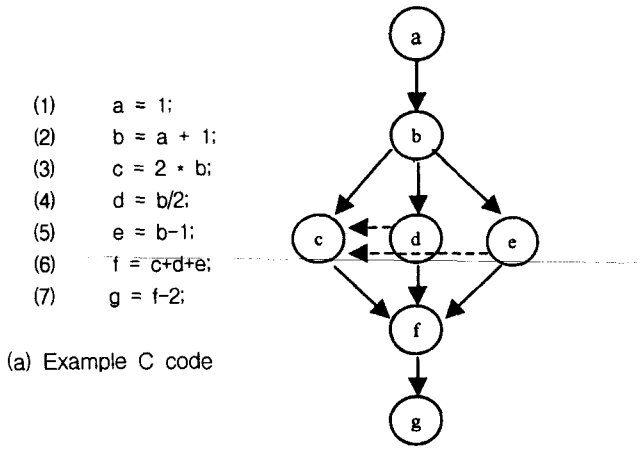
Given a dependence graph, it is impossible to derive a precise live range of a variable before instruction scheduling. This is because definitions and/or uses of variables are not ordered before scheduling. However, it is possible to derive the longest possible live range, that is, the range out of which the variable is guaranteed to be dead no matter what order is employed later by an instruction scheduler. This section investigates how to derive the longest possible live range of a given variable, and then proposes an algorithm for the generation of register-reuse chains based on the longest possible live range analysis.

Note that the live range of a given variable spans to the node that is the last use of the variable. Thus, the problem of finding the live range of a given variable is equivalent to the problem of finding the last use of the variable. Let 'a' be the name of a given variable. If only a single node reuses variable 'a', then this node is the end of the live range. If multiple nodes reuse variable 'a', it is necessary to find the node that is executed last. Since this analysis is performed before instruction scheduling, it is not clear which node is executed last. The only exception is the case when dependencies enforce strict execution order of these nodes. In this case, the node of the last use is the end of the live range. In other cases, it is necessary to derive a node that is dependent on all the nodes that reuse variable 'a'. At this node, a variable is guaranteed to be dead because all the nodes that use the variable is already executed before this node. Thus, this node is called *an ultimately killing node* of variable 'a'. If there exists more than one ultimately killing nodes, the one that is a predecessor of all others is called the *earliest ultimately killing node (EUK)* of a. The EUK of a given variable is the first node that guarantees the end of its live range.

Example 3.1: Consider the example shown in Fig. 3-1. Fig. 3-1 (a) shows a segment of C code. The corresponding data dependence graph is shown in Fig. 3-1 (b). In this graph, variable 'a' has a single dependent node 'b'. Thus, the live range of 'a' ends at node 'b'. Node 'b' has three dependent nodes, 'c', 'd', and 'e'. Node 'f' depends on all three nodes. Thus, 'f' is an ultimately killing node of 'b'. Similarly, 'g' is also dependent on 'c', 'd', and 'e'. Thus, 'g' is also an ultimately killing node of 'b'. Since 'f' is a predecessor of 'g', node 'f' is the *EUK* of 'b', that is, 'f' is the first node that guarantees the end of the live range of 'b'. In fact, 'b' is dead earlier than 'f' because 'b' is dead when 'c', 'd', and 'e' are executed. However, it is not decided which one is executed last at this stage (note that instruction schedule is not fixed yet). Thus, no node among 'c', 'd', and 'e' guarantees the death of 'b'. Only 'f' guarantees that 'b' is dead because 'c', 'd', and 'e' are already executed when 'f' is executed.

Based on the longest possible live range analysis, register allocation can be performed. Note that two variables can share the same register as long as their live ranges do not overlap. For a given variable, its EUK node guarantees the end of its longest possible live range. Thus, the register assigned to a variable can be reused by the EUK node of the variable. The basic idea of the register allocation algorithm proposed in this section is to search the EUK node and assign the same register.

Fig. 3-2 shows the register allocation algorithm based on the longest possible live range. The input of this function is a dependence graph and the output is a set of register-reuse chains. Each chain contains the nodes that can be assigned to the same register. This algorithm consists of two major functions, regAllocLongestRange() and searchEUK(). Function regAllocLongestRange() visits each node in BFS order. If the visited node does not belong to any register-reuse chain, a new chain is created and the node becomes the head of the new chain. Then, find its EUK node by calling searchEUK(). Function searchEUK() takes a node as an input argument and finds



(a) Example C code

(b) Data dependence graph

(1) R3 = 1	(R3=1)
(2) R3 = R3 + 1	(R3=2)
(3) R3 = 2 * R3	(R3=4)
(4) R4 = R3/2	(R4=2)
(5) R5 = R3-1	(R5=3)
(6) R3=R3+R4+R5	(R3=9)
(7) R3 = R3-2	(R3=7)

(c) Incorrect instruction order

(1) R3 = 1	(R3=1)
(2) R3 = R3 + 1	(R3=2)
(4) R4 = R3/2	(R4=1)
(5) R5 = R3-1	(R5=1)
(3) R3 = R3*2	(R3=4)
(6) R3=R3+R4+R5	(R3=6)
(7) R3 = R3-2	(R3=4)

(d) Correct instruction order

Fig. 3.1 Dependencies generated by register allocation

the EUK node of the input node. Then, attach the EUK node to the same chain as the input node. If the EUK node is already attached to a register-reuse chain, find any dependent node of the EUK node that is not attached to any chain. Then, attach it to the same chain as the input. Once a node is attached, then searchEUK() is recursively called by passing the EUK node as the input argument. Recursive call ends when a node does not have a dependent node.

```
RegAllocLongestRange ( DAG )
{
    Node p  = visit DAG in Breadth First Search (BFS) order;
    if (p not visited) {
        createNewChain (p);
        searchEUK (p);
    }
}
searchEUK (Node p)
{
    if ( Node q = findEUK (p) ) {
        if(q not visited) {
            attachNodeToChain (q);
            searchEUK (q);
        } else if (q = firstUnvisitedDescedentNode(q)) {
            attachNodeToChain (q);
            searchEUK (q);
        }
    }
}
```

Fig. 3.2 Register-reuse chain generation algorithm.

Example 3.2: Consider the data dependence graph shown in Fig. 2.1 (b). Function regAllocLongestRange() starts with root node 'a' and creates a new chain that includes only node 'a'. Then, function searchEUK() is called to search the EUK node of 'a'. Note that 'a' has three edges incident into nodes, 'b', 'c', and 'd', respectively. These three paths merge at node 'l', that is, 'l' is the EUK of 'a'. So, node 'l' is assigned to the same register-reuse chain as 'a'. Then, searchEUK() is called again to search the EUK node of 'l'. However, node 'l' does not have any dependent node, and therefore the recursive search of EUK node stops. Function regAllocLongestRange() visits the next node in BFS order. Therefore, node 'b' is visited next. Function regAllocLongestRange() creates the second chain to contain 'b' and then calls searchEUK(). Node 'l' is found as its EUK because the node 'l' is the only dependent node of 'b'. However, node 'l' is already assigned to the first register-reuse chain, so it cannot be assigned to the same chain as 'b'. Since node 'l' does not have any dependent node, recursive call to serachEUK() stops. Thus 'b' is the only node that is assigned to the second register-reuse chain. Now another new search for the third register-reuse chain is initiated starting with node 'c'. The EUK searching finds 'g' as its EUK node and it is not visited. Thus, 'g' is assigned to the third register-reuse chain. Then, by recursively calling searchEUK(), 'h' and 'k' are found and assigned to the third register-reuse chain. Finally, node 'l' is found as the EUK of 'k', but it is already assigned to the first register-reuse chain. So, searching the EUK stops. By visiting all the nodes that are not assigned and recursively searching its EUK nodes, register-reuse chains are derived as shown in Fig. 3-3. This result shows that six registers are necessary to prevent any restrictions on instruction scheduling.

Chain [0] = {a, l}
Chain [1] = {b}
Chain [2] = {c, g, h, k}
Chain [3] = {d, i}
Chain [4] = {e, j}
Chain [5] = {f}

Fig. 3.3 Register-reuse chains generated by EUK search

3.2 Dependencies Created by Register Reuse

In the previous subsection, register-reuse chains are generated based on the longest-possible live range analysis, that is, two variables are attached to the same chain only if their longest possible live ranges do not overlap. However, overlapped longest possible live ranges do not always prevent the two variables from sharing the same register. Instead, the overlapping means that there exist some instruction schedules that prevent the two variables from sharing the same register, while the other instruction schedules can allow them to share the same register. Therefore, even though two variables overlap their longest possible live ranges, they can still be assigned to the same register as long as an instruction scheduler avoids the schedules that prevent the register share. This section analyzes what schedules must be avoided for a given register allocation and how to lead an instruction scheduler to avoid such schedules.

Consider the case when variable 'b' reuses the register assigned to variable 'a' while the two variables have overlapped longest possible live ranges. This enforces instruction scheduling to schedule nodes in such a way that the live range of variable 'a' must end at the instruction that assigns variable 'b'. In other words, all the nodes that use variable 'a' must be scheduled earlier than 'b'. If this is not true, the generated assembly instruction is wrong. Suppose that variable 'c' also uses variable 'a' while 'c' is scheduled later than 'b'. Then, at the time 'c' attempts to use 'a', it is no longer available because the register for 'a' is already occupied by 'b'.

Example 3.3: Consider Fig. 3.1 again. Assume that variables 'a', 'b', 'c', 'f', and 'g' are assigned to the same register, R3 while 'd' and 'e' are assigned to R4, and R5, respectively. Note that 'b' and 'c' are assigned to the same register although their live ranges overlap. Fig. 3.1 shows two different instruction schedules whose resulting assembly codes are given in Fig. 3-1 (c) and Fig. 3-1 (d), respectively. These two figures are different in the execution orders of nodes 'c', 'd' and 'e'. In Fig. 3-1 (c), node 'c' is scheduled before 'd' and 'e' while, in Fig. 3-1 (d), node 'c' is scheduled after 'd' and 'e'. The rightmost columns of these two figures show the results of corresponding instructions. Note that the final value of R3 is wrong in Fig. 3-1 (c). This is because instructions (4) and (5) use a wrong value of 'b'. At instruction (4), R3 does not store the value of 'a' because it is already replaced by 'b' at instruction (3). On the other hand, R3 stores the right value of 'a' until instruction (4) in Fig. 3-1 (d).

The above example shows that a certain schedule results in an invalid assembly code if variables are assigned to the same register although their longest live ranges overlap. Therefore, if a register allocator decides to assign the same register to variables whose longest possible live ranges overlap, it needs to pass information to an instruction scheduler so that the scheduler can avoid an invalid order. Such information can be stored as a dependence. For example, consider again the case when variable 'b' reuses the register assigned to variable 'a' while the two variables have overlapped longest possible live ranges. Then, any instruction depending on 'a' (i.e., using 'a') must be scheduled earlier than 'b'. If dependencies are created from the dependent nodes of 'a' to 'b', these dependencies can lead an instruction scheduler to schedule the dependent nodes earlier than 'b'.

Example 3.3 (continued): In order to avoid the invalid instruction order as shown in Fig. 3-1 (d), it is necessary to create additional dependencies. Since variable 'c' reuses the register assigned to 'b', it is necessary to generate dependencies from all the nodes that depend on 'b' to 'c'. Thus, two dependencies are generated from 'd' to 'c' and 'e' to 'c', respectively. These new dependencies are shown as dashed arrows in the dependence graph (see Fig. 3-1 (b)). The additional dependencies enforce node 'b' to be scheduled after 'c' and 'd'.

The dependence created by register reuse 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. Consider the dependence graph shown in Fig. 3.1 again. Suppose that three functional units are available. Then, 'c', 'd', and 'e' can be executed simultaneously. When 'c' reuses the register assigned to 'b', variables 'd' and 'e' already accessed the value of 'a'. Therefore, the computation for 'd' and 'e' is correct. However, if there is only one functional unit, the dependence behaves exactly the same as a normal dependence. Therefore, the instruction (3) has to be moved behind instruction and (4) and (5) as shown in Fig. 3.1 (d).

It is not always possible to find a valid schedule (i.e., the schedule that generates correct code) for a given register allocation if two variables share the same register

even though their longest possible live ranges overlap. This case can be detected and prevented at register allocation stage. The detection is possible by observing dependence conflict between existing dependencies and the new dependence created by register sharing by two variables whose longest possible live ranges overlap. If the new dependence violates an existing dependence, then the register sharing is impossible, i.e., there is no valid schedule that allows the register sharing.

4. Merge of Register-Reuse Chains

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 reduction of the number of chains creates new dependences 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.

4.1 Criterion of Chain Merging

One way of reducing 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 Fig. 4.1 (a). Dependencies between nodes enforce only partial 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'-->b'-->c'-->d'-->e'-->f'-->'g'-->h'-->k'
a'-->c'-->b'-->d'-->e'-->f'-->g'-->h'-->k'
a'-->d'-->c'-->b'-->e'-->f'-->g'-->h'-->k'
```

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

```
b'-->a'-->c'-->d'-->e'-->f'-->g'-->h'-->k'
```

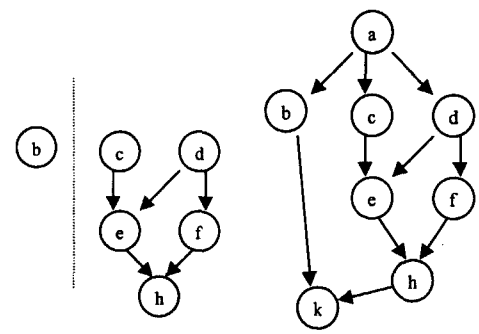
In this dependence graph, there are 30 different possible orderings of the nodes.

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 is proposed to be used as the criterion to decide the efficiency of a register allocator.

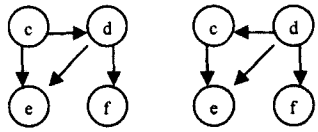
The algorithm shown in Fig. 4.1 computes the number of schedules for a given dependence graph.

```
int Num_Schedules (DAG)
{
    if (DAG is totally ordered) Return 1;
    Remaining_DAG = Remove_Fixed_Nodes (DAG);
    if (Remaining_DAG can be divided into two disjoint
subgraphs)
    {
        /* Divide the Remaining_DAG into LEFT and RIGHT
*/
        S = C(|LEFT|+|RIGHT|,|LEFT|) * Num_Schedules(LEFT)
*               Num_Schedules(RIGHT);
    }
    else {
        S = 0;
        for ( i = 0; i < num_starting_nodes; i ++)  {
            Temp_DAG = Rm_Start_Node(Remaining_DAG, i);
            S += Num_Schedules (Temp_DAG);
        }
    }
    Return S;
}
```

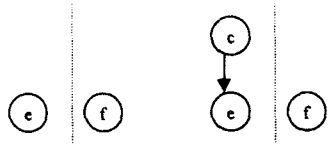
Fig. 4.1 Algorithm for computing the number of schedules



(a) Data dependence graph. (b) Remaining dependence graph after removing nodes with fixed schedule



(c) Selection and removal of the first-scheduled node for RIGHT; 'c' scheduled first vs. 'd' scheduled first.



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

Fig. 4.2 Computation of the number of schedules

The algorithm is explained with the example shown in Fig. 4.2 (a). Function Num_Schedules() checks if the input graph is totally ordered. If it is true, it returns 1 because only one schedule is possible for a totally ordered graph. Next, Remove_Fixed_Nodes() removes the nodes whose order is fixed. Note that the removal of nodes with a fixed schedule does not change the number of schedules. For the example, the orders of 'a' and 'k' in Fig. 4.2 (a) are fixed because node 'a' must be executed before all other nodes and node 'k' must be executed after all other nodes. Therefore, function Remove_Fixed_Nodes() removes these two nodes from the graph. The remaining graph is shown in Fig. 4.2 (b). Then, if-clause checks whether the remaining graph can be divided into two disjoint (i.e., no edge between these two) subgraphs. If it is true, the body of the if-statement is executed. Let the two subgraphs denoted by *LEFT* and *RIGHT*. In the example, *LEFT*={b} and *RIGHT*={c, d, e, f, h}. Then, the total number of schedules can be formulated as:

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

where $S(\ )$ denotes the number of possible schedules for a given graph, $|LEFT|$ is the number of nodes in subgraph LEFT, $|RIGHT|$ is the number of nodes in subgraph *RIGHT*, and $C(|LEFT|+|RIGHT|, |LEFT|)$ is the number of possibilities in choosing $|LEFT|$ elements out of $|LEFT|+|RIGHT|$ elements.

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 the nodes in *LEFT* occupies the entire available time slots of $|LEFT| + |RIGHT|$. For example in Fig. 4.2 (b), there are six nodes in the remaining graph, one node in the *LEFT* subgraph and five nodes in the RIGHT graph. Ordering of these nodes can be considered as allocating these nodes in six time slots. When scheduling these six nodes, node 'b' can be allocated to any time slot from 1 to 6 while 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

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

Now, it is necessary to compute *S(LEFT)* and *S(RIGHT)* which 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 Num_Schedules() function.

Consider *S(LEFT)*, i.e., the number of schedules of the left graph. Since there is only one element, only one schedule is possible (i.e., totally ordered). Thus, Num_Schedules() returns 1. Consider the subgraph *RIGHT* = {c, d, e, f, h}. Node 'h' should be always scheduled last so that Remove_Nodes can remove this node. The resulting graph is shown in Fig. 4.2 (c). There are two starting nodes and therefore their schedules are not fixed. Thus, these starting nodes cannot be removed, and the remaining graph cannot be decomposed into two disjoint subgraphs. So, the test of if-statement fails and the body of 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, Num_Schedules() function is called again. Once every subproblem is solved, the total number of the schedules is simply the summation of the number 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 Fig. 4.2 (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 Fig. 4.2 (d). Now, the graph can be decomposed into two disjoint subgraphs and therefore, 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

Fig. 4.2 (c)). Then, this node can be removed from the graph and the resulting graph can be decomposed into two disjoint subgraphs (see Fig. 4.2 (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:

$$\begin{aligned} S(RIGHT) &= S(\{c, d, e, f, h\}) \\ &= S(\{c, d, e, f\} \mid c \text{ is scheduled first}) \\ &\quad + S(\{c, d, e, f\} \mid d \text{ is scheduled first}) \\ &= S(\{e, f\}) + S(\{c, e, f\}) = 5 \end{aligned}$$

Thus, the total number of schedules is

$$S(GRAPH) = C(|LEFT|+|RIGHT|, |LEFT|) * S(LEFT) * S(RIGHT) = C(6, 1)*1*5 = 30$$

In summary, the algorithm of computing the number of schedules follows 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.

4.2 Heuristics for Chain Merge

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 combinatorially many possibilities for selecting the chains to be merged. Thus, this subsection 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 Fig. 3.3, 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].

Fig. 4.3 describes the algorithm for merging register-reuse chains. Visiting nodes in the dependence graph in BFS order can effectively perform the search for the candidate pairs. When a node is visited, check if its dependent node is the first of a separate chain. If it is true, merge the two chains that include the two nodes. If there are more than one such node, select the one that results in the greatest number of schedules.

```
Merge_Chains(chains, DAG)
{
    while ((number_of_chains > number_of_registers)
            or (all the nodes in DAG are visited) ) {
            node = visit_in_BFS(DAG);
            number_of_chains = merge (chains, node );
    }
}
int merge (chains, node)
{
    if (node has multiple dependent nodes ) {
        chosen_node = select_best_choice ( node );
        if (chosen_node exists)
            merge_two_chains(p,chosen_node,
                             merged_chains);
    }
}
```

Fig. 4.3 Regster-reuse chain merging algorithm

The complexity of the above heuristic is combinatiroal in the worst case. However, it is computationally affordable because a single basic block generally does not have many statements.

5. Evaluation

Matrix multiplication program is used as the test program. Table 5.1 summarizes the result of register allocation. The program consists of twelve basic blocks of which number is given in the first column of the table. The second column shows the number of independent register-reuse chains generated by the optimal algorithm in Section 3. Except two basic blocks, the number of chains is less than or equal to eight that is, in general, less than the number of registers in RISC processors including a relatively small embedded processor. Recall that the optimal register-reuse chains do not create any additional restriction to an instruction scheduler. Thus, for most of basic blocks, the proposed register allocation does not restrict instruction scheduling at all. The third column represents the number of register-reuse chains resulting from chain merging. Note that the number of register-reuse chains is reduced in basic blocks 3 and 5. The fourth and fifth column show the number of schedules before merging and after merging, respectively. In basic block 3 and 5, it is shown that the number of schedules is reduced by chain merging. The last column also represents the number of schedules after merging. In this merging, an arbitrary chain is selected for merging instead of the

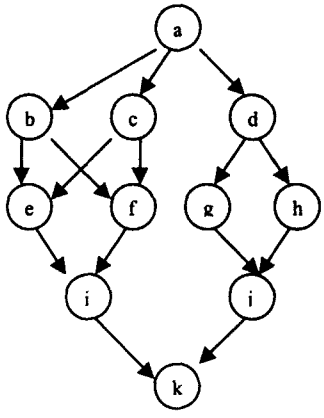
selection scheme used in the proposed heuristic. Both basic blocks 3 and 5, the arbitrary selection results in less number of schedules.

Table 5.1 Evaluation Results

Basic Block	Number of independent reuse chains	Number of reuse chains after merging	Number of schedules before merging	Number of scheduler after merging	Number of schedules after arbitrary merging
1	4	4	24	24	24
2	5	5	120	120	120
3	8	7	6048	4032	2016
4	6	6	720	720	720
5	12	10	2217600	739200	369600
6	9	9	2661120	2661120	2661120
7	6	6	720	720	720
8	7	7	2520	2520	2520
9	5	5	120	120	120
10	6	6	360	360	360
11	4	4	24	24	24
12	5	5	60	60	60

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

(a) Register reuse chains



(b) Additional dependencies created by chain merging

Fig. 5.1 Register allocation and generated dependencies enforcing parallel execution of instructions

Consider again the example dependence graph shown in Fig. 2.2. Recall that the previous approach added two additional dependencies (as shown in Fig. 2.2 (b)) in order to reduce the number chains to four. However, Fig. 5.1 (a) shows the register-reuse chains generated by the proposed heuristic, and Fig. 5.2 (b) shows the corresponding dependence graph with the additional dependencies shown with dotted line. Note that Fig. 5.1 (b) is better than the previous approach in the sense that it does not increase the length of the critical path at all. However, the two dependencies from 'e' to 'f' and from 'f' to 'e' enforce these two instructions to be executed simultaneously. This is an example that shows register allocation enforces simultaneous execution of instructions.

6. Conclusions

In many basic blocks, register-reuse chains generated by the optimal algorithm fit in available registers. Therefore, with the optimal algorithm, register allocation does not create any restriction to instruction scheduling. When chains are merged, additional dependencies cannot be avoided. The proposed merging heuristic reduces additional restriction to instruction scheduling. The number of schedules is used as the criterion to compare register allocators. In the future, they need to be compared by measuring the efficiency of an instruction scheduler for benchmarks. In addition, the merging heuristic can be improved by merging chains that do not have adjacent nodes.

참 고 문 헌

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

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

[3] Christopher Fraser and David Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/ Cummings Publishing Co., 1995.

[4] Steve Beaty. *Instruction scheduling using genetic algorithms*. Ph.D. Thesis. Colorado State University, 1991.

[5] Jinpyo Park, and Soo-Mook Moon, Optimal Register Coalescing. In *Proc. Int. Conf. Parallel Architecture and Compilation Techniques*, Oct. 1998.

[6] Stan Y. Liao. *Code generation and optimization for embedded digital signal processors*, Ph.D. Thesis, MIT Department of EECS, January 22, 1996.

[7] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource Spackling: A framework for integrating register allocation in local and global schedulers. In *Proc. of IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pp 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*, pp 243-254, 1993.

[9] 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*, June 1994.

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

[11] Yukong Zhang, *Instruction Scheduling and Register Allocation for Embedded Processor*, Ph.D. thesis, Louisiana Tech University, 1999.

저 자 소 개

이 혁 재 (李 赫 宰)

1965년 2월 8일생. 1987년 서울대 전자공학과 졸업. 1996년 Purdue대학교 전기 및 컴퓨터 공학과 졸업 (공박). 1996년-1998년 Louisiana 공과대학 컴퓨터학과 조교수. 1998-현재 Intel Senior Engineer.

E-mail : jasonl@ichips.intel.com