Software Pipelining with Reduced Register Requirement *

Jian Wang† Andreas Krall M. Anton Ertl

Institut für Computersprachen
Technische Universität Wien
Argentinierstr. 8
A-1040 Vienna, Austria

Abstract

Although it is well-known that a strong interaction exists between software pipelining and register allocation, simultaneous software pipelining and register allocation is still less understood and remains an open problem. In this paper, we first analyze the influence of the re-use of registers on the loop data dependence graph and model the problem of software pipelining to reduce register requirement based on the concepts of row-number and column-number of decomposed software pipelining. Given the row-numbers of all operations in the loop, generating the column-numbers to obtain the minimum register requirement can be modelled as the integer programming problem which can be solved in polynomial time. Then, a solution is presented to the special case without resource constraints and an approximate solution is presented to the general case with resource constraints. Register spilling and register coloring are considered to further reduce the number of actually needed registers. The preliminary experimental results are presented to indicate the efficiency of our new approach.

Keywords: Instruction-level Parallelism, Loop Scheduling, Software Pipelining, Register Allocation, Register Spilling, Register Coloring

1 Introduction

Exploiting Instruction-Level Parallelism (ILP) within loops has become a key compilation issue for the instruction-level parallel processors like Very Long Instruction Word (VLIW) and superscalar machines [1, 2, 3]. Software pipelining is an efficient compilation technique to exploit ILP for loops, which initiates successive iterations before previous iterations complete [4, 5, 6, 7, 8, 9, 10, 11, 12].

Register Allocation is another key compilation issue [13, 14, 15, 16, 17]. It has been well-known that a strong interaction exists between software pipelining and register allocation. On one hand, performing register allocation before software pipelining may introduce unacceptable anti-dependences due to the reuse of registers, which may limit software pipelining [17, 3]. On the other hand, if software pipelining is done before register allocation, more registers than

*This work was supported by the Lise Meitner Stipendium funded by the Austrian Science Foundation (FWF) and the Austrian Science and Research Ministry.

†Email: jian@mips.complang.tuwien.ac.at; Tel: 43-1-588014474; Fax: 43-1-5057838.

necessary may be needed, which may cause unnecessary register spillings and severely degrade the performance of the pipelined loop [3]. However, simultaneous register allocation and software pipelining is still less understood and remains open.

The interaction between register allocation and loop-free code scheduling has been studied since the mid 1980s [10, 18, 13, 19, 20, 21], and register allocation for software pipelined loop has been studied by many researchers and some efficient techniques have been proposed [22, 12, 17, 15]. However, the interaction between register allocation and software pipelining was lately considered in few studies. Mangione-Smith, et al. developed a lower bound on the number of registers needed for a given modulo scheduled loop [23]. A technique called lifetime-sensitive modulo scheduling has been presented by Huff [24], in which he uses the idea of bidirectional slack-scheduling to perform the modulo scheduling with a try for shortening the lifetime of a variable. Under the assumption of unlimited resources, Ning and Gao have modelled the optimal loop scheduling and buffer allocation problem as an integer programming problem and developed a polynomial time algorithm [25, 26]. Govindarajan et al have integrated the consideration of resource constraints into Ning and Gao's model [27], but their approach may suffer from exponential computation complexity in the worst case. In [28], software pipelining is treated as a three step procedure: MRT-scheduling, iteration-scheduling and register allocation. After getting a MRT-schedule, they consider the register requirement problem in the step of iteration-scheduling which is similar to the phase of finding column-number in our decomposed software pipelining framework. However, for the loops with dependence cycles, it is not clear in their approach how to find a MRT-schedule such that the iterative-scheduling is successful.

In this paper, we study the interaction between software pipelining and register allocation from a new perspective. Based on the concepts of row-number and column-number of decomposed software pipelining [29, 30, 31], we first analyze the influence of the re-use of registers on the loop data dependence graph and model the problem of software pipelining to reduce register requirement. In our model, data dependences, resource constraints and register requirement can be considered altogether. Our idea is that the row-number is used to satisfy the resource constraints whereas the column-number is used to satisfy the data dependences and control the register requirement. Then we find that, given the row-numbers of all operations in the loop, generating the column-numbers to obtain the minimum register requirement can be modelled as the integer programming problem which can be solved in polynomial time. The work of Ning and Gao is actually a special case of our model when no resource constraint exists. Finally we develop a heuristic approach for software pipelining to reduce register requirement. Register spilling and register coloring both are considered to further reduce the number of actually needed registers. Our approach is supported by preliminary experimental results.

This paper is organized as follows: The next section gives a brief introduction to decomposed software pipelining. Section 3 first analyzes the influence of the re-use of registers on the loop data dependence graph and then models the problem of software pipelining to reduce the register requirement. In Section 4, we present a solution to the special case of our model when no resource constraint exists. In Section 5, an approaximate solution is developed and both the register spilling and the register coloring are considered to further reduce the number of actually needed registers. The preliminary experimental results are also presented. Conclusion is given in Section 6.

2 Decomposed Software Pipelining(DESP)

The data dependences of a loop can be represented by a Loop Data Dependence Graph (LDDG), (O, E, λ, δ), where O is the operation set and E the dependence edge set; the **dependence**

distance λ and the **delay** δ are two non-negative integers associated with each edge. For example, $e = (op, op')$ and $(\lambda(e), \delta(e))$ denote that op' can only be issued $\delta(e)$ cycles after the start of the operation op of the $\lambda(e)$th previous iteration [2, 9].

DESP is a novel modulo scheduling approach, and its idea can be illustrated by Figure 2.1 as an example. First, we modify the LDDG by removing some edges so that the graph becomes acyclic; secondly, we apply the list scheduling technique [1, 2] on the modified graph to generate the software pipelined loop body under the resource constraints, and use the row-number to denote the cycle-number of each operation in the loop body; thirdly, we determine the iteration-number (denoted as column-number in the context of DESP) of each operation such that all data dependences in LDDG are satisfied.

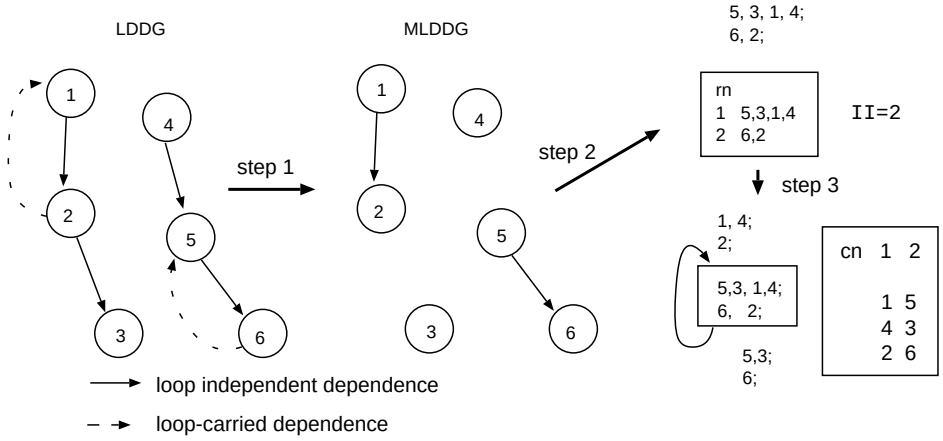

Figure 2.1 Decomposed Software Pipelining

Formally, DESP theoretically decomposes the loop schedule σ into two functions, row-number and column-number.

Definition 2.1 Let $G = (O, E, \lambda, \delta)$ be the LDDG of a loop, and σ a valid loop schedule for G with initiation interval II[1]. We define the row-number rn and the column-number cn, two mappings from O to N (non-negative integer set), such that

$$\sigma(op, 1) = rn(op) + II * (cn(op) - 1) \quad and \quad \sigma(op, i) = \sigma(op, 1) + II * (i - 1).$$

□

Thus, software pipelining can be described below with the concepts of row-number and column-number.

Definition 2.2 (Decomposed Software Pipelining) Let $G = (O, E, \lambda, \delta)$ be the LDDG of a loop, we say that the row-number, rn, and the column-number, cn, are valid for the loop, if and only if the following constraints are satisfied:

1. resource constraints: $\forall op_i, op_j \in O$, if $rn(op_i) = rn(op_j)$, then op_i and op_j must not conflict with respect to the resources[2];

2. dependence constraints:

$$\exists II \in N, \quad rn(op') - rn(op) + II * (\lambda(e) + cn(op') - cn(op)) \geq \delta(e), \quad \forall e = (op, op') \in E.$$

[1] That is, a new iteration of the loop can be issued every II cycles

[2] Here, we only consider the pipelined operations and the single-cycle operations, but the definition is easily extended to the case of multi-cycle non-pipelined operations.

II is called the initiation interval or the length of the software pipelined loop body. The goal of decomposed software pipelining is to find valid row-number and column-number with minimum II. □

In this paper, we assume that $min(rn(op)) = 1$ and $min(cn(op)) = 0$. In previous papers [29, 30, 31], we have proven the following theoretical results.

Theorem 2.1 For a given LDDG, suppose we have constructed row-number rn which satisfies the resource constraints. We can construct column-number cn such that the data dependence constraints are also satisfied, if and only if, for each cycle C of the LDDG,

$$\sum_{\forall e \in C} \tau(e) \leq 0$$

where $\tau(e) = -\lambda(e) + \lceil(\delta(e) + rn(op) - rn(op'))/II\rceil$, $e = (op, op')$. □

Theorem 2.1 implicitly points out that, if we have constructed row-numbers taking into account the resource constraints for a LDDG without cycle, then we can always construct column-numbers such that the data dependence constraints are also satisfied.

3 Modelling the Problem

In this section, we will model the problem of software pipelining to reduce register requirement. That is, under the constraints of resources and data dependences, our goal is minimizing both the initiation interval and the register requirement.

3.1 Influence of Register Requirement on LDDG

In the case of register allocation for a software pipelined loop, more than one register could be allocated to one variable. We assume that the registers are well-distributed to different iterations, as shown in Figure 3.1.

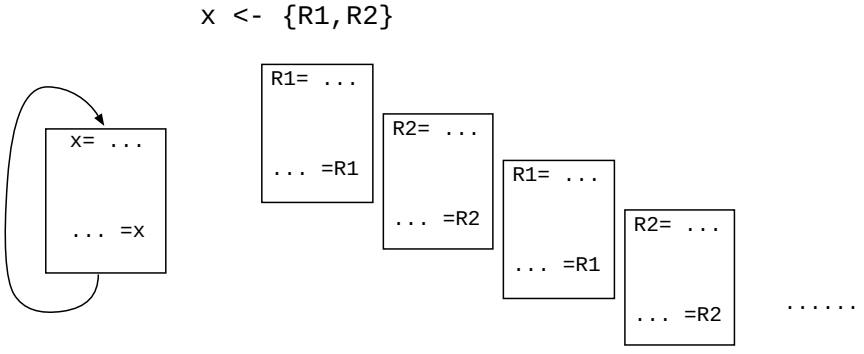

Figure 3.1 Register Allocation for Software Pipelined Loops

Thus, the anti-dependence edges caused by the re-use of registers are introduced to LDDG in such a way that,

(1) If the variable u is first defined by op_i and then used by op_j in the original loop body – we call (op_i, op_j) a loop-independent dependence(denoted as lid), and u is allocated with K_u registers, then one anti-dependence edge with the iteration-distance of K_u (i.e. $\lambda((op_j, op_i)) = K_u$) is introduced to LDDG from op_j to op_i (e.g. the variable x in Figure 3.2);

(2) If the variable u is first used by op_j and then defined by op_i in the original loop body – we call (op_i, op_j) a loop-carried dependence(denoted as lcd), and u is allocated with K_u registers, then one anti-dependence edge with the iteration-distance (i.e. $\lambda((op_j, op_i)) = K_u - 1$) is introduced into LDDG from op_j to op_i (e.g. the variable y in Figure 3.2).

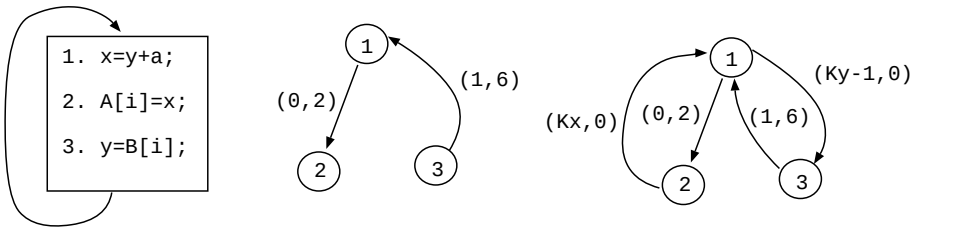

Figure 3.2 Anti-Dependence Edges Caused by Re-Use of Registers

3.2 Definitions

The above analysis can be concluded by a new dependence graph $LDDG^+$ which will be defined as follows. In addition, we will also define the valid software pipelined loop and the valid software pipelined loop body.

Definition 3.1 If a software pipelined loop body is given, then the initiation interval (the length of the loop body) II and the row-number of each operation are determined, denoted as $\{II, rn\}$; If a software pipelined loop is given, then the initiation interval II, the row-number and the column-number of each operation are determined, denoted as $\{II, rn, cn\}$. □

In fact, if rn is determined, then II can be determined in terms of the LDDG. How to determine II, however, is beyond the scope of this paper.

Definition 3.2 Given the LDDG (O, E, λ, δ) of a loop, a software pipelined loop $\{II, rn, cn\}$ is valid if and only if, (1) the resource constraints are satisfied; and (2) $\forall e = (op_i, op_j) \in E, rn(op_j) - rn(op_i) + II * (\lambda(e) + cn(op_j) - cn(op_i)) \geq \delta(e)$. □

Definition 3.3 Given the LDDG (O, E, λ, δ) of a loop, a software pipelined loop body $\{II, rn\}$ is valid if and only if, for each cycle C of the LDDG, $\sum_{\forall e \in C} \tau(e) \leq 0$, where $\tau(e) = -\lambda(e) + \lceil(\delta(e) + rn(op_i) - rn(op_j))/II\rceil, e = (op_i, op_j)$. □

Obviously, Definition 3.2 and 3.3 are directly derived from Definition 2.2 and Theorem 2.1, respectively.

Definition 3.4 Given the LDDG (O, E, λ, δ) of a loop, we define $LDDG^+ = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$, where

(1) $E_{lid} = \{(op_i, op_j) | op_i, op_j \in O$; op_j first defines a variable and then op_i uses the variable in the loop body. $\}$; $E_{lcd} = \{(op_i, op_j) | op_i, op_j \in O$; op_i first uses a variable and then op_j defines the variable in the loop body. $\}$;

(2) $\forall e \in E_{lid} \cup E_{lcd}, \delta(e) = 0$;

(3) $\forall e \in E_{lid}, \lambda(e) = K_u$; $\forall e \in E_{lcd}, \lambda(e) = K_u - 1$. Where K_u is the number of registers allocated to u and u is the corresponding variable. □

Figure 3.2 is an example, where (b) is LDDG and (c) is $LDDG^+$. Actually, $LDDG^+$ is the graph extended by all anti-dependence edges caused by the re-use of registers to LDDG.

Definition 3.5 Given a loop, in the context of our model, its register requirement RR is

defined as the sum of the number of registers allocated to each loop-variant variable in the loop, that is, $RR = \sum_{\forall u \in V} K_u$. Where V is the set of all loop-variant variables(that is, they are defined in the loop body) and K_u the number of registers allocated to u. □

3.3 The Problem Description

We are ready to model the problem of software pipelining to reduce register requirement. From Theorem 2.1, Definition 3.2 and 3.3, we directly have the following result.

Theorem 3.1 Given a loop and a valid software pipelined loop body $\{II, rn\}$, we can always find the column-number, cn, such that $\{II, rn, cn\}$ is a valid software pipelined loop. □

Now, given the LDDG (O, E, λ, δ) of a loop and a valid software pipelined loop body $\{II, rn\}$, any column-number, cn, which we find to make $\{II, rn, cn\}$ valid, must satisfy the data dependence constraints below:

$$rn(op_j) - rn(op_i) + II * (\lambda(e) + cn(op_j) - cn(op_i)) \geq \delta(e), \ \forall e = (op_i, op_j) \in E.$$

Also, let $LDDG^+ = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$, for any anti-dependence edges caused by the re-use of registers, the following constraints must be satisfied:

$$\begin{aligned} rn(op_j) - rn(op_i) + II * (K_u + cn(op_j) - cn(op_i)) \geq 0, \forall (op_i, op_j) \in E_{lid}; \\ rn(op_j) - rn(op_i) + II * (K_u - 1 + cn(op_j) - cn(op_i)) \geq 0, \forall (op_i, op_j) \in E_{lcd}; \end{aligned}$$

Thus, given a valid $\{II, rn\}$, the minimum register requirement can be determined by the above dependence constraints.

Definition 3.6 Given the LDDG (O, E, λ, δ) of a loop and a valid software pipelined loop body $\{II, rn\}$, let $LDDG^+ = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$, the problem of finding the minimum register requirement can be modelled as an integer programming problem as follows:

$$\min \sum_{\forall u \in V} K_u$$

Subject to

$$\begin{aligned} cn(op_j) - cn(op_i) \geq -\lambda(e) + \lceil (\delta(e) - rn(op_j) + rn(op_i))/II \rceil, \forall e = (op_i, op_j) \in E; \\ K_u + cn(op_j) - cn(op_i) \geq \lceil (rn(op_i) - rn(op_j))/II \rceil, \forall (op_i, op_j) \in E_{lid}; \\ K_u + cn(op_j) - cn(op_i) \geq 1 + \lceil (rn(op_i) - rn(op_j))/II \rceil, \forall (op_i, op_j) \in E_{lcd}; \\ K_u, cn(op) \ integers, \ \forall op \in O, u \in V. \end{aligned}$$

Where V is the set of all loop-variant variables. The minimum register requirement corresponding to $\{II, rn\}$ is denoted as $RR_{min}(II, rn)$. □.

The column-number can be automatically determined while the integer programming problem is solved. Thus, the problem of software pipelining to reduce register requirement can be expressed in a very simple way.

Definition 3.7 (The problem of software pipelining to reduce register requirement) Given the LDDG of a loop, find a valid software pipelined loop body $\{II, rn\}$ such that both the II and the $RR_{min}(II, rn)$ are minimum. □

Theorem 3.2 Given the LDDG (O, E, λ, δ) of a loop and a valid software pipelined loop body $\{II, rn\}$, the solution to the integer programming problem of Definition 3.6 exists. □

Proof: Given $\{II, rn\}$, the right hand side of each inequality of the data dependence constraints is a constant, so we only need to prove that, for any valid $\{II, rn\}$, we can always find cn such that

$$cn(op_j) - cn(op_i) \geq -\lambda(e) + \lceil(\delta(e) - rn(op_j) + rn(op_i))/II\rceil, \ \forall e = (op_i, op_j) \in E.$$

This is actually the result of Theorem 3.1. □

Theorem 3.3 The constraint matrix in the integer programming problem of Definition 3.6 is totally unimodular. □

Theorem 3.3, which is directly derived from the work of Ning and Gao [25, 26], points out that the integer programming problem of Definition 3.6 can be solved as a linear programming problem and the optimal solution is guaranteed to be integral. Therefore, in order to solve our integer programming problem, we can use general linear programming algorithms such as simplex, ellipsoid or interior point methods [32, 33, 34]. Also, Ning and Gao presented a more efficient algorithm whose computation complexity is $O(n^3 \log n)$, where n is the number of nodes in LDDG.

4 A Solution to the Special Case without Resource Constraints

In Definition 3.6 and 3.7, we use the row-number to treat the resource constraints and use the column-number to satisfy the data dependences and to determine the minimum register requirement. In this section, we will point out that, when no resource constraint exists, the model of software pipelining to reduce register requirement can be greatly simplified.

As the first step, we combine the row-number and the column-number to a new function, π.

Definition 4.1 Given a valid software pipelined loop $\{II, rn, cn\}$, we define $\pi(op) = rn(op) + II * cn(op)$ for each operation op. □

π actually represents a pipelinable loop with the initiation interval of II, where $\pi(op)$ denotes the cycle-number of op in the pipelinable loop body. Figure 4.1 gives an example.

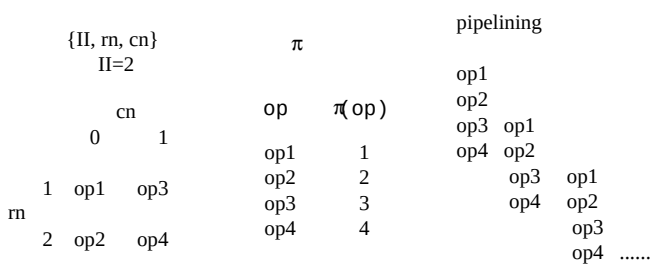

Figure 4.1 {II, rn, cn} and π

With the concept of π, Definition 3.6 and 3.7 can be combined below:

Definition 4.2 (The problem of software pipelining to reduce register requirement without resource constraints) Given $LDDG = (O, E, \lambda, \delta)$ of a loop, let $LDDG^+ =$

$(O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$, the problem of software pipelining to reduce register requirement can be modelled as an integer programming problem as follows:

$$\min \sum_{\forall u \in V} K_u$$

Subject to

$$\begin{aligned} \pi(op_j) - \pi(op_i) \geq -\lambda(e) * MII + \delta(e), \forall e = (op_i, op_j) \in E; \\ MII * K_u + \pi(op_j) - \pi(op_i) \geq 0, \forall (op_i, op_j) \in E_{lid}; \\ MII * K_u + \pi(op_j) - \pi(op_i) \geq 1, \forall (op_i, op_j) \in E_{lcd}; \\ K_u, \pi(op) \ integers, \ \forall op \in O, u \in V. \end{aligned}$$

Where V is the set of all loop-variant variables, MII is the minimum initiation interval which can be computed as

$$MII = \max_{\forall C \in LDDG}((\sum_{\forall e \in C} \delta(e))/(\sum_{\forall e \in C} \lambda(e)))$$

□.

Obviously, the above integer programming problem is of the same properties as those of Definition 3.6, so it can be also efficiently solved.

5 An Approximate Solution to the General Case

As to the general case with resource constraints, we can only look for heuristic approaches. The key is first to find a valid software pipelined loop body which is sensitive to the register requirement.

Our software pipelining framework is based on the DESP as shown in Figure 2.1. In the first step, we use the following method to modify the LDDG [29, 30, 31]:

(1) find out all strongly connected components (SCCs) in the LDDG, remove all edges which are not included in the SCCs;

(2) under the unlimited resource constraints, generate a software pipelined loop for the SCCs, denoted as (rn_0, cn_0);

(3) for each edge $e = (op_i, op_j)$ of SCCs, if $rn_0(op_j) - rn_0(op_i) < \delta(e)$, then remove e from the SCCs.

The remaining graph is acyclic, denoted as MLDDG. We have proven that any software pipelined loop body satisfying the data dependences of the MLDDG is valid.

In the second step, we use the heuristic scheduling method presented in this section. In the third step, we solve the integer programming problem to determine the column-number. Two other measures – register spilling and register coloring – are also presented to further reduce the register requirement.

5.1 The Heuristic Scheduling

The register requirement of a variable is determined by its lifetime; and its lifetime is mainly determined by the column-numbers of the corresponding operations. For example, u is written by op_i and read by op_j, then u's lifetime, $lt(u)$, satisfies the inequality: $II*(cn(op_j)-cn(op_i))-II+1 \leq lt(u) \leq II*(cn(op_j)-cn(op_i))+II-1$. If $rn(op_i)$ and $rn(op_j)$ are known, then $lt(u)$ can be precisely computed by $lt(u) = II*(cn(op_j)-cn(op_i))+rn(op_i)-rn(op_j)$. Note that $|rn(op_i)-rn(op_j)| < II$, the objective of this subsection is only to reduce $cn(op_j)-cn(op_i)$.

Let us consider two operations, op and op', with a dependence edge $e=(op,op')$ in the LDDG, we have the data dependence constraint: $II*(\lambda(e)+cn(op')-cn(op))+rn(op')-rn(op) \geq \delta(e)$. If $rn(op')-rn(op) \geq \delta(e)$, then $cn(op')-cn(op)$ can take the minimum value, $-\lambda(e)$. In general, the greater is $rn(op')-rn(op)$, the less is $cn(op')-cn(op)$. These facts help us to develop scheduling heuristics to the register requirement.

First, add some edges to the MLDDG such that $cn(op')-cn(op)$ of these edges can take the minimum values. As most dependence edges originally in the LDDG have been removed in the MLDDG, there may be a lot of schedulable operations at each cycle. Without increase of the estimated II[3], it is possible that some operations can be delayed to schedule such that some dependence edges are satisfied.

We suggest that an edge $e=(op,op')$ can be added to the MLDDG only if

(1) The operation op' does not use the critical resources. res is one of the critical resources if $t-1+\lceil N/n \rceil \leq$ the estimated II, where t is the current cycle, N is the number of operations using res and n is the number of res in the machine; and

(2) The lengths of the resulting dependence paths are not greater than the estimated II. That is, $t+\delta(e)+height(op')-1 \leq$ the estimated II, where t is the current cycle and $height(op')$ is the height of op' in the MLDDG. In this case, we say op' can be delayed at cycle t.

Then we present some scheduling heuristics to determine the scheduling priorities of operations.

In order to obtain the optimal time efficiency, we consider the height of operation(that means the length of the longest path from the operation to the end of the graph) in the MLDDG as the first heuristic. Besides, two heuristics sensitive to register requirement are considered as follows:

(1) If an operation can be delayed and uses the critical resources, then it has a lower scheduling priority;

(2) If an operation has no predecessor in the LDDG, then it has a higher scheduling priority.

We take an example shown in Figure 5.1 and 5.2 to demonstrate the above scheduling process. Figure 5.1(1) is the loop and (2) the machine model. The LDDG is shown in Figure 5.2(1). After the first step of DESP software pipelining framework, we get the MLDDG as shown in Figure 5.2(2). The machine has only one multiplier but the loop includes two multiplications, so the estimated II is 2. First, $op2$ does not use the critical resources and can be delayed, so the edge $(op1, op2)$ is added to the MLDDG. At the first cycle, $op4$ uses the critical resources and can be delayed, so it has a lower scheduling priority; thus, $op1$, $op3$, $op5$ and $op6$ are put in the first cycle. $op2$ and $op4$ are put in the second cycle.

[3]The estimated II can be derived from the critical cycle of the LDDG and the number of operations using the critical resources.

```
The Original Loop:

for i=1 to n do
s=s+a[i]
a[i]=s*s*a[i]
enddo
```

(1) The Loop

```
The Code of
the Loop Body:

1. t0=t0+1;
2. t1=a[t0];
3. s=s+t1;
4. t2=s*s;
5. t3=t1*t2;
6. a[t0]=t3
```

Pipeline	Number	Operation	Latency
Memory port	2	Load	13
		Store	1
Address ALU	2	Add/Sub	1
Adder	1	FAdd/FSub	1
		IAdd/ISub	1
Multiplier	1	FMUL	2
		IMUL	2

(2) The Machine Model

Figure 5.1 An Example

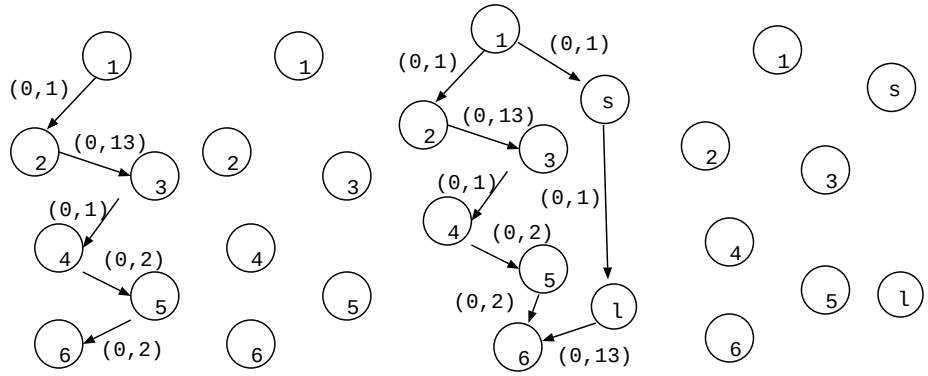

Figure 5.2 LDDGs, MLDDGs

5.2 Register Spilling

Spilling decision are conventionally made only when a register conflict occurs, that is, the number of simultaneously live variables is greater than the number of available machine registers. The effect of spilling is keeping the result of a computation in memory rather than in a register such that the register can be re-used to keep the result of a new computation at the cost of increasing the number of load/store operations.

In this subsection we discuss register spilling problem for software pipelining. Our starting-points are that, (1) spilling decision should be made during software pipelining; and (2) spilling can be used to reduce the register requirement without degradation of the optimal software pipelining performance.

In our heuristic scheduling and the solving of the integer programming problem of Definition 3.6, we try to minimize the lifetime of each variable. Given a loop, however, each variable has a lower bound on its lifetime; this lower bound is determined by the length of the longest path in the LDDG from the variable's definition to its use. For example, in Figure 5.2(1), the lower bound on $t0$'s lifetime is the length of the path from $op1$ to $op6$, that is 19.

The effects of a spilling can re-construct the LDDG and may decrease the lower bound on the lifetimes for some variables. For the example as shown in Figure 5.2, after we spill the use, $use(op6, t0)$, – that is, spill $t0$ only for $op6$ – the LDDG is modified to the one as shown in Figure 5.2(3). It is not difficult to compute that the lower bound on $t0$'s lifetime becomes 1. The register requirement is reduced even if we consider the registers needed for the newly introduced variable.

Spilling introduces new load/store operations. We suggest that the load/store operations caused by spilling does not increase the estimated II.

The $spilling - benefit$ of a use is defined as the decrease of lifetime per inserted load/store operation. More precisely, given a use, $use(op, u)$, where op is the operation using variable u, let $lt(u)$ be the lower bound on u's lifetime before spill $use(op, u)$ and $lt'(u)$ the one after spill $use(op, u)$. Thus, the spilling-benefit of $use(op, u)$ is $\lceil (lt(u) - lt'(u))/2 \rceil$ as a store and a load are inserted to the LDDG. Obviously, a use with greater value of spilling-benefit is the one with higher spilling priority.

We take the loop shown in Figure 5.1 as an example to illustrate the above ideas. We discuss two cases: (1) scheduling without spilling; (2) scheduling with spilling.

For the first case, the estimated II is 2 since the machine has one multiplier but the loop body contains two multiplications. The valid software pipelined loop body can be found under the constraint of the MLDDG (shown in Figure 5.2(2)). After adding an edge $(1, 2)$ to the MLDDG, we obtain $rn(1) = rn(3) = rn(5) = rn(6) = 1$ and $rn(2) = rn(4) = 2$. It is easy to compute the number of required registers for the loop, which is 23.

For the second case, the estimated II is also 2. After computing the spilling-benefits of all uses, we find that $up(op6, t0)$ has the greatest value of spilling-benefit which is $\lceil (19 - 15)/2 \rceil = 2$, so $up(op6, t0)$ has the highest spilling priority. After spilling $up(op6, t0)$, the modified LDDG and MLDDG are shown in Figure 5.2(3) and (4), respectively. After adding an edge $(1, 2)$ to the MLDDG, we obtain $rn(1) = rn(3) = rn(5) = rn(6) = rn(s) = 1$ and $rn(2) = rn(4) = rn(l) = 2$. It is easy to compute the number of required registers for the loop, which is 21.

5.3 Register Coloring

After we obtain the software pipelined loop with the minimum register requirement, we can further reduce the number of actually needed registers by register coloring since we defined the register requirement of a loop as the sum of the register requirements of all variables when we built our model.

First, we construct the weighted interference graph (WIG), where the weight on each node represents the register requirement of the corresponding variable. Thus, the number of actually needed registers can be counted by the conventional method [14].

Secondly, we define the degree of a node with consideration of the weight of each node. For example, let K_u denote the weight of u, $NS(u)$ be its adjacent node set, then the degree of u, $deg(u) = K_u + \sum_{v \in NS(u)} K_v$.

Finally, the number of registers can be counted as follows:

(1) For each node u in the WIG, compute its degree $deg(u)$;

(2) Let $RR = \min_u(deg(u))$;

(3) Find a node v in the WIG with the minimum degree; if $deg(v) > RR$ then $RR = deg(v)$;

(4) Update the WIG by removing v and all its edges;

(5) If the WIG is empty then return (RR); else re-compute the degrees of all remaining nodes in the WIG and goto (3).

5.4 The Algorithm

The algorithm is described as follows:

Algorithm Scheduling;

INPUT: The loop to be software pipelined and its LDDG;

OUTPUT: The software pipelined loop;

BEGIN

1. Construct the MLDDG, determine the estimated II;

2. Compute the height of each operation in the MLDDG;

3. Check and add some edges to the MLDDG;

4. Call the spill-checking;

5. Re-compute the heights of some operations;

6. Find out all schedulable operations at the current cycle and put them in the DRS (Data Ready Set);

7. Determine the scheduling priorities of all operations in the DRS;

8. Under the constraint of resources, select the operation with the highest scheduling priority from the DRS and place it in the current cycle, update the DSR. This step repeats until no operation can be placed in the current cycle;

9. If all operations of the loop have been scheduled then goto step 10; else update the DRS, the MLDDG and the estimated II and goto step 3;

10. For each operation, let its row-number be its cycle-number. Determine the II; solve the integer programming problem of Definition 3.6 to determine the column-number and the minimum register requirement

11. Generate the software pipelined loop in terms of the row-numbers and the column-numbers;

12. Call the register coloring and determine the number of actually needed registers;

END;

The **spill-checking algorithm** is described as follows:

Algorithm Spill-Checking;

BEGIN

1. If the memory access unit is one of the critical resources, then return;

2. Compute the spilling-benefit of each use;

3. Under the constraint of not increasing the estimated II, select a (group of) use(s) for spilling. In this step, if no use can be selected then return;

4. Update the MLDDG and the DRS; return;

END;

5.5 The Preliminary Experimental Results

The effort to implement the above algorithm is underway. Before getting extensive experimental tests, we select four examples to verify our algorithm. Except for example 1, the other three examples are selected from the Livermore benchmarks. As our preliminary experiments are mainly conducted by a manual simulation, we try to select some simple loops in a random

way. The machine model we use in the experiments is shown in Figure 5.1(2). For comparison, example 1 and the machine model are directly cited from [17].

Table 1. Register Requirement for Two Scheduling Approaches

Example	L	MII	DESP	Our new approach
1	20	2	27	21
2	22	3	39	30
3	18	3	21	18
4	17	2	27	25

note: L = the length of the longest dependence path in the loop body;
MII = the minimum initiation interval.

Table 1 gives the register requirements for the optimal software pipelining performance by two scheduling approaches – DESP and our new approach presented in this paper. We see that our new approach can obtain an improvement over DESP up to 23.1% in register use without degradation of the optimal performance. Note that, for example 1, [17] gets the result with the requirement of 28 registers.

6 Conclusion

This paper studies the interaction between software pipelining and register allocation. Based on the concepts of row-number and column-number of decomposed software pipelining, we analyze the influence of the re-use of registers on the loop data dependence graph and develop a model of software pipelining to reduce register requirement in which data dependences, resource constraints and register requirement can be considered together. We use the row-number to satisfy the resource constraints and use the column-number to satisfy the data dependences and control the register requirement. Given the row-numbers of all operations in the loop, generating the column-numbers to obtain the minimum register requirement can be modelled as an integer programming problem which can be solved in polynomial time. A solution to the special case without resource constraints and an approximate solution to the general case with resource constraints are presented. Register spilling and register coloring are considered to further reduce the number of actually needed registers. The preliminary experimental results show the efficiency of our new approach. Our future work is to implement our new model and approaches using our compiler testbed.

References

[1] J.A. Fisher, D. Landskov, and B.D. Shriver. Microcode compaction: Looking backward and looking forward. In *proceedings of 1981 National Computer Conference*, 95-102 1981.

[2] F. Gasperoni. Compilation techniques for vliw architectures. Technical Report TR435, New York University, March 1989.

[3] B. R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1), January 1993.

[4] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *proceedings of the 14th In-*

ternational Symposium on Microprogramming and Microarchitectures (MICRO-14), pages 183–198, October 1981.

[5] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *proceedings of European Symposium on Programming, Lecture notes in Computer Science, No.300*, pages 221 –235. Spring-Verlag, June 1988.

[6] P. Y. T. Hsu. *Highly Concurrent Scalar Processing.* PhD thesis, University of Illinois, Urbana-Champaign, 1986.

[7] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In *proceedings of the 20th International Symposium on Microprogramming and Microarchitectures (MICRO-20)*, pages 69–79, 1987.

[8] B. Su, S. Ding, and J. Xia. Urpr - an extension of urcr for software pipelining. In *proceedings of the 19th International Symposium on Microprogramming and Microarchitectures (MICRO-19)*, pages 104 – 108, 1986.

[9] Bogong Su and Jian Wang. Loop-carried dependence and the general URPR software pipelining approach. In *proceedings of the 24th Annual Hawaii International Conference on System Sciences*, pages 366–372. IEEE and ACM, January 1991.

[10] R.F. Touzeau. A fortran compiler for the fps-164 scientific compute. In *proceedings of ACM SIGPLAN Symposium on Compiler Construction*, 1984.

[11] A.E. Charlesworth. An approach to scientific array processing: The architecture design of the ap-120b/fps-164 family. *Computer*, pages 18–27, September 1981.

[12] M.S. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, CMU, 1987. CMU-CS-87-187.

[13] D.G. Bradlee, S. J. Eggers, and R.R. Henry. Integrated register allocation and instruction scheduling for riscs. In *proceedings of the 4th International Conference on ASPLOS*, 1991.

[14] G. J. Chaitin. Register allocation and spilling via graph coloring. In *proceedings of ACM SIGPLAN Symp. on Compiler Construction*, 1982.

[15] L.J. Hendren, G.R. Gao, E. R. Altman, and C. Mukerji. Register allocation using cyclic interval graph: A new approach to an old problem. Technical Report ACAPS Technical Memo 33, McGill University, 1992.

[16] S. S. Pinter. Register allocation with instruction scheduling: A new approach. In *proceedings of ACM SIGPLAN PLDI*, 1993.

[17] B. R. Rau, M. Lee, P.P. Tirumalai, and M.S. Schlansker. Register allocation for software pipelined loops. In *proceedings of PLDI*, 1992.

[18] J. R. Goodman and W. Hsu. Code scheduling and register allocation in large basic blocks. In *proceedings of International Conference on Supercomputing*, 1988.

[19] S.A. Mahlke, W.Y. Chen, P.P. Chang, and W.W. Hwu. Scalar program performance on multiple-instruction-issue processors with a limited number of registers. In *proceedings of the 25th HAWAII International Conference on System Sciences*, January 1992.

[20] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-conscious global register allocation. In *proceedings of PLSA*, April 1994.

[21] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. In M. Cosnard, G. R. Gao, and G. M. Silberman, editors, *proceedings of International Conference on Parallel Architectures and Compilation Techniques*. IFIP, North-Holland, August 1994.

[22] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Compile-time optimization of memory and register usage on the cray-2. In *proceedings of the second Workshop on Languages and Compilers*, 1989.

[23] William Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register requirements of pipelined processors. In *proceedings of 1992 ACM International Conference on Supercomputing*, 1992.

[24] R. Huff. Lifetime-sensitive modulo scheduling. In *proceedings of ACM SIGPLAN PLDI*, pages 258–267, June 1993.

[25] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. Technical Report ACAPS Technical Memo 42, McGill University, 1993.

[26] Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. In *proceedings of POPL*, January 1993.

[27] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrainted rate-optimal software pipelining. In *proceedings of the 27th International Symposium on Microprogramming and Microarchitectures (MICRO-27)*, December 1994.

[28] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *proceedings of the 27th International Symposium on Microprogramming and Microarchitectures (MICRO-27)*, December 1994.

[29] Jian Wang and Christine Eisenbeis. Decomposed Software Pipelining: A new approach to exploit instruction level parallelism for loop programs. In Michel Cosnard, Kemal Ebcioglu, and Jean-Luc Gaudiot, editors, *proceedings of IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 3–15. IFIP, North-Holland, January 1993.

[30] Jian Wang and Christine Eisenbeis. Decomposed Software Pipelining. Reseach Repport RR-1838, INRIA-Rocquencourt, France, 1993.

[31] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. Decomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):357–379, 1994.

[32] V. Chvatal. *Linear Programming*. W.H. Freeman and Company, 1983.

[33] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, (4), 1984.

[34] L.G.. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Doklady*, (20), 1979.