Software Pipelining with Register Allocation and Spilling *

Jian Wang† Andreas Krall
M. Anton Ertl Christine Eisenbeis‡

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

Abstract

Simultaneous register allocation and software pipelining is still less understood and remains an open problem. In this paper, we first present the Register Requirement Graph (RRG) which can dynamically reflect the register requirement during software pipelining. Then, using the RRG as a basis, we develop a Register-Pressure-Sensitive (RPS) scheduling technique and study the problem of register spilling for software pipelining. We also present three algorithms – RPS without spilling, RPS with spilling and the software pipelining with a limited number of registers. The preliminary experimental results show that the first two algorithms can efficiently reduce the register requirement without degradation of the optimal performance and the third can effectively exploit instruction-level parallelism within loops even for those machines with a small register file.

Keywords: Instruction-level Parallelism, Loop Scheduling, Software Pipelining, Register Allocation, Spilling, Data Dependence Graph

1 Introduction

It has been well known that 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 has been proposed for exploiting ILP within loops, which can effectively overlap the execution of operations from different iterations [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 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 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.

*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.

‡Dr. Eisenbeis is with INRIA-Rocquencourt, Domaine de Voluceau, BP 105-78153, Le Chesnay Cedex, France.

The interaction between register allocation and loop-free code scheduling has been studied since the mid 1980s [10, 18, 13, 16, 19], and register allocation for software pipelined loop has been studied by many researchers and some efficient techniques have been proposed [20, 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 [21]. Ning and Gao have presented a framework of register allocation for software pipelining by which they deduce the minimal number of registers needed for finding some optimal software pipelined loop [22], but they do not consider the resource constraints. A called lifetime-sensitive modulo scheduling technique has been presented by Huff [23], 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, but he does not consider the register spilling problem.

Our approaches presented in this paper are different from all of the above. In order to understand the interaction between register allocation and software pipelining, we present a novel framework, called Register Requirement Graph (RRG), which can dynamically reflect the register requirement during software pipelining. While software pipelining, the RRG is used to control the register pressure caused by software pipelining itself. On one hand, the RRG gives the register related information to guide the scheduling process such that no more register than necessary is needed. On the other hand, from the RRG we can dynamically estimate the register requirement such that the spilling decision and the tradeoff between the initiation interval and register pressure are efficiently made.

The next section gives a background to make this paper self-contained. The work reported in this paper can be concluded as follows: (1) Present the RRG to estimate the register requirement during software pipelining (Section 3); (2) Use the RRG to develop a Register- Pressure-Sensitive (RPS) scheduling technique (Section 4); (3) Study the problem of register spilling to reduce the register pressure without degradation of the optimal performance (Section 5); (4) Present three software pipelining algorithms – RPS without spilling, RPS with spilling and the software pipelining with a limited number of registers (Section 6); (5) Give the preliminary experimental results to indicate the efficiency of the three algorithms (Section 7).

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[1]. First, we modify the LDDG by removing some edges so that the graph becomes acyclic; secondly, we apply the list scheduling technique 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.

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

[1] For all examples in this paper, the loop-independent dependence edges are solid edges whereas loop-carried ones are dotted if we do not attach (λ, δ) to each edge.

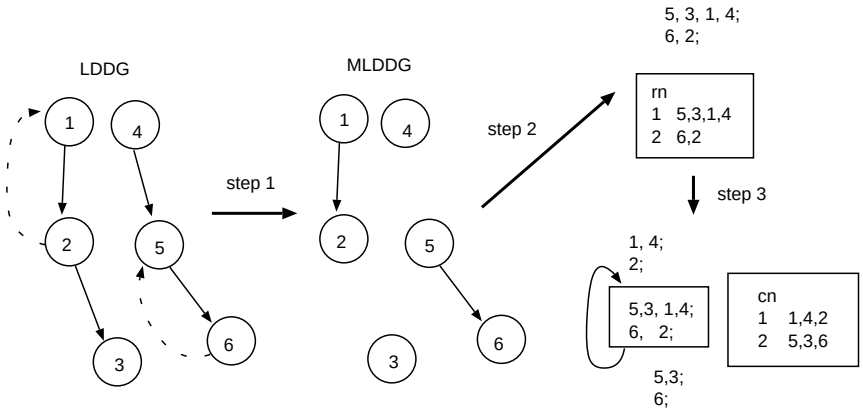


Figure 2.1 Decomposed Software Pipelining

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. 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 can not be resource-conflict[2];

2. dependence constraints:

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

II is called as 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 our previous papers [24, 25, 26], 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 $\tau(e) = -\lambda(e) + \lceil(\delta(e) + rn(op) - rn(op'))/II\rceil$, $e = (op, op')$. □

[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.

The following corallary is direct from Theorem 2.1.

Corallary 2.1 For a LDDG without cycle, if we have constructed row-number taking into account the resource constraints, then we can always construct column-number such that the data dependence constraints are also satisfied. □

3 Register Requirement Graph

In decomposed software pipelining, the column-number is an important parameter to control the register requirement of each variable. In fact, the register requirement is mainly determined by the difference between the column-numbers of two operations which have a data dependence (denoted as dcn_{ij}). For example, suppose variable u is written by op_i and read by op_j, then dcn_{ij} gives the estimate of the lifetime of u. Thus, we first present the Register Requirement Graph (RRG) which can dynamically estimate dcn_{ij}. The RRG gives the heuristics to guide the scheduling process (determining the row-number).

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 [24, 25, 26]:

(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 row-numbers satisfying the data dependences of the MLDDG must satisfy the condition of Theorem 2.1.

Given the LDDG (O, E, λ, δ) of a loop, after the first step of decomposed software pipelining, we obtain an acyclic dependence graph $MLDDG = (O, E_m, \delta)$. A new graph, called **register requirement graph**, is defined as $RRG = (O, E, \omega)$, where ω is a weight on each edge which represents the estimated difference between the column-numbers of two operations in the worst case.

Let MII be the estimated minimum initiation interval, before scheduling the software pipelined loop body, we initially define ω as follows:

(1) $\omega(e) = -\lambda(e), \forall e \in E_m$;

(2) $\omega(e) = -\lambda(e) + \lceil(\delta(e) + MII - 1)/MII\rceil, \forall e \in E - E_m$.

While scheduling the software pipelined loop body, we recompute $\omega(e)$ for each $e = (op_i, op_j) \in E - E_m$ as follows:

(1) $\omega(e) = -\lambda(e) + \lceil(\delta(e) - (rn(op_j) - rn(op_i)))/MII\rceil$, if $rn(op_i)$ and $rn(op_j)$ both are determined;

(2) $\omega(e) = -\lambda(e) + \lceil(\delta(e) - 1 + rn(op_i))/MII\rceil$, if $rn(op_i)$ is determined but $rn(op_j)$ is not;

(3) $\omega(e) = -\lambda(e) + 1 + \lceil(\delta(e) - rn(op_j))/MII\rceil$, if $rn(op_j)$ is determined but $rn(op_i)$ is not;

An example of RRG is given in Figure 3.1 and 3.2, Figure 3.1(1)is the loop and (2) the machine model. Its LDDG and MLDDG are shown in Figure 3.2(1) and (2), respectively. Figure 3.2(3) is the initial RRG. Figure 3.2(4) is the RRG when $rn(op1) = rn(op3) = rn(op5) = rn(op6) = 1$ and $rn(op2) = rn(op4) = 2$.

```
The Original Loop:

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

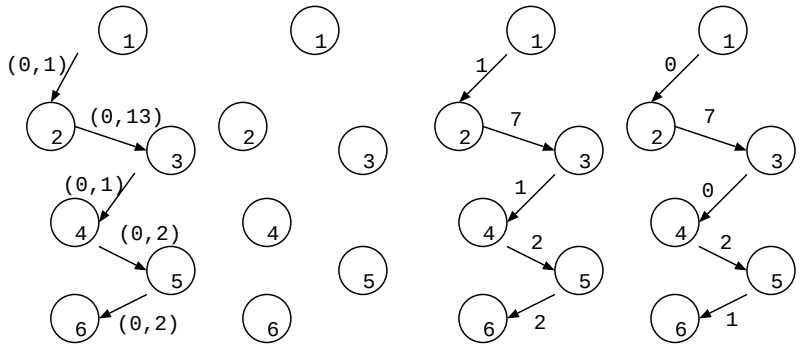
```
The Code of
the Loop Body:

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

(1) The Loop (2) The Machine Model

Figure 3.1 An Example



(1) LDDG (2) MLDDG (3) RRG (4) RRG

Figure 3.2 LDDG, MLDDG and RRGs

A **definition-use path** is defined as a path from the operation writing a variable to any operation reading the variable in the LDDG. The critical definition-use path of variable u, $cdup_u$, is defined as

$$\sum_{\forall e \in cdup_u} \omega(e) = \max_{any\ dup\ of\ u} (\sum_{\forall e \in dup_u} \omega(e)).$$

Let $RRG = (O, E, \omega)$, for each edge $e \in E$, $\alpha(e)$ is defined as the number of variables whose critical definition-use path include e.

RRG has the following two properties:

(1) Let $RRG = (O, E, \omega)$, $cdup_u$ be the critical definition-use path of u, then $\sum_{\forall e \in cdup_u} \omega(e)$ gives the estimate of the register requirement of u.

(2) Let $RRG = (O, E, \omega)$, during scheduling the software pipelined loop body, for any edge e which is in the LDDG but not included in the MLDDG, if e is satisfied (that is, $rn(op_j) - rn(op_i) \geq \delta(e), e = (op_i, op_j)$), then the register requirement may be decreased by up to $(\omega(e) + \lambda(e)) * \alpha(e)$ registers compared to the case when e is not satisfied.

4 RPS Scheduling

We present the following two heuristics to direct the scheduling process:

(1) Delay some operations to be scheduled such that some dependence edges can be satisfied in the software pipelined loop body;

(2) Develop register-pressure-sensitive heuristics to determine the scheduling priorities for operations.

In the second step of our software pipelining framework, we use list scheduling on the

MLDDG (obtained in the first step) to determine the row-numbers for all operations. First, we find out all schedulable operations at the current cycle and put them into the Data Ready Set (DRS), then we select the operations with the highest scheduling priority to schedule.

As most dependence edges originally in the LDDG have been removed in the MLDDG, there may be a lot of schedulable operations in the DRS at each cycle. Without increase of the estimated II[3], it is greatly possible that some operations can be delayed to schedule such that some dependence edges avoid being unnecessary broken.

We suggest that an operation can be delayed and removed from the current DRS only if

(1) The operation 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, e is the dependence edge which we are willing to hold and $height(op)$ is the height of op in the MLDDG.

For the example of Figure 2.2(2), at the first cycle, all operations are schedulable and can be put into the DRS, but only operation 2 can be delayed and removed from the DRS.

When there are more than one operation which can be delayed, we first consider the operation with the greatest value of $(\omega(e) + \lambda(e)) * \alpha(e)$, where e is the dependence edge which we are willing to hold.

Next we discuss how to determine the scheduling priorities for the operations of the DRS.

In order to obtain the optimal time efficiency, we consider the height of operation in the MLDDG as the first heuristic. The second heuristic is sensitive to the register pressure and is derived from the RRG.

At the current cycle t, suppose op_i and op_j are the operations with the greatest value of height in the MLDDG. If op_i and op_j are not resource-conflict, then they should be scheduled at t. If op_i and op_j are resource-conflict, then we use the second heuristic to determine their scheduling priorities as follows:

(1) If an operation is scheduled at t, then another should be scheduled after the tth cycle;

(2) Suppose op_i is scheduled at t, let $DES(op_i)$ be the dependence edge set which included all edges adjacent to op_i. Let $rn(op_i) = t$, we re-compute the new value of ω of each edge in $DES(op_i)$, denoted as ω_{new}. Thus, we can compute the $register - benefit$ of op_i,

$$\eta(op_i, t) = \sum_{\forall e \in DES(op_i)} (\omega(e) - \omega_{new}(e)) * \alpha(e);$$

(3) By the same method as step (2), we compute $\eta(op_j, t)$;

(4) The operation with greater value of η (the register-benefit) is the one with higher scheduling priority.

4.1 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

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

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 and probably degrading the code performance. Software pipelining overlaps the execution of the operations from different iterations, increasing register pressure and generating excessive spill code in the case of small machine register files.

This section discusses register spilling problem for software pipelining. Our starting-point is that spilling decision should be made during software pipelining such that the interactions between register allocation and loop scheduling can be seen. The RRG can dynamically reflect the change on the register requirement during software pipelining and make our starting-point feasible.

Two problems to be discussed are as follows: (1) When is a spilling decision made during software pipelining? (2) How to do a spilling ?

In the loop body, we suppose that, a variable only has a definition (the operation defining the variable) but may have more than one use (the operation using the variable). We first want to make a remark: The meaning of spilling in the context of this paper is something different from the conventional spilling problem [14]. We say spilling a (a group of) use(s) but do not say spilling a variable (that is, spilling all its uses). By spilling a use, we mean that a store operation after the definition and a load operation before the spilled use are inserted, and other uses still reference the value of the variable in a register.

From the RRG, we can dynamically estimate the register requirement at each cycle. Spilling is needed only if the number of required registers is greater than the number of available machine registers. In fact, other measures like delaying some operations to schedule and introducing some dependence edges into the MLDDG can also decrease the register requirement.

Another necessary condition for spilling is that the load/store operations caused by spilling does not increase the estimated II. In the case of that there are not enough available machine registers to reach the estimated II, we first increase the estimated II and then consider spilling or other measures to decrease the register requirement (see next section).

The spilling process consists of two steps: (1) Select a (a group of) use(s) for spilling; (2) Modify the MLDDG and the RRG by adding the necessary load/store operations and re-computing the value of corresponding ω and α.

The $spilling - benefit$ of a use is defined as the number of saved registers per inserted load/store operation. More precisely, given a use, $use(op, u)$, where op is the operation using variable u, under the assumption of that $use(op, u)$ has been spilled, we re-compute the minimal register requirement of variable u and the new introduced variable, denoted as K'_u. Thus, the spilling-benefit of $use(op, u)$ is $\lceil (K_u - K'_u)/2 \rceil$ as a store and a load are inserted to the MLDDG and the RRG for spilling a use.

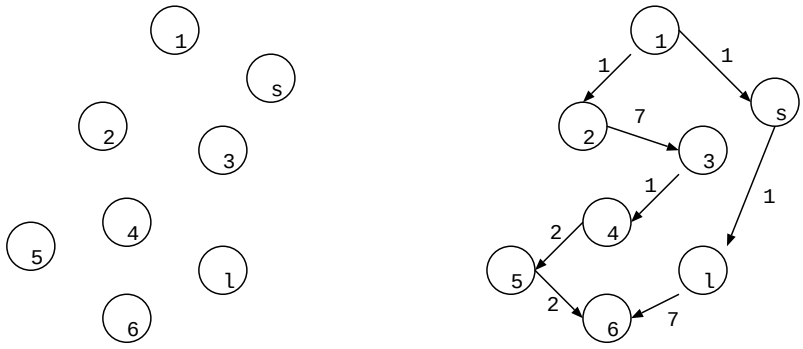
Obviously, a use with greater value of spilling-benefit is the one with higher spilling priority.

We take the loop shown in Figure 3.1 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 software pipelined loop body can be found under the constraints of the MLDDG (shown in Figure 3.2(2)) and the initial RRG (shown in Figure 3.2(3)). By delaying operation 2, 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 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 $\lceil (13-7-2)/2 \rceil = 2$, so $up(op6, t0)$ has the highest spilling priority. After spilling $up(op6, t0)$, the modified MLDDG and the modified initial RRG are shown in Figure 5.1. By delaying operation 2, 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 which is 21.



(1) The modified MLDDG (2) The modified initial RRG

Figure 5.1 Scheduling with Register Spilling

An important observation is that, spilling can decrease the register requirement without degradation of the optimal software pipelining performance if the spilling decision can be efficiently controlled.

5 Algorithms

On the basis of the last three sections, we present three software pipelining algorithms. The first two are software pipelining to minimize the register requirement and the third is software pipelining with a limited number of registers.

5.1 RPS Scheduling without Spilling

The algorithm is described as follows:

Algorithm RPS-without-Spilling;

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. Find out all definition-use paths of each variable, construct the RRG;
4. Find out all schedulable operations and put them in the DRS;
5. Find out those operations which can be delayed one by one, remove them from the DRS;
6. Determine the scheduling priorities of all operations in the DRS;
7. 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;

8. If all operations of the loop have been scheduled then goto step 9; else update the DRS and the RRG and goto step 5;

9. For each operation, let its row-number be its cycle-number. From Theorem 2.3, the column-number of each operation is computed in terms of the row-numbers and the II;

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

END;

5.2 RPS Scheduling with Spilling

This algorithm is different from the RPS-without-Spilling algorithm in the way that a new spill-checking step is inserted between step 5 and step 6. The new step calls a spill-checking algorithm which 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, we actually only consider those uses which are on the critical definition-use paths;

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

4. Update the MLDDG, the RRG and the DRS; return;

END;

5.3 Software Pipelining with a Limited Number of Registers

The above two algorithms try to obtain the optimal software pipelined loop with the minimal register requirement. In this section we present an approach for software pipelining with a limited number of registers.

Our idea is that we first estimated the register requirement, if the number of required registers is greater than the given number of available machine registers then we increase the estimated II such that the register requirement is reduced.

However, it is difficult and complicated to precisely estimate the register requirement. RRG only estimates the register requirement of each variable. The problem of which variables can share the same registers remains open during software pipelining.

We present the following heuristics: Let K_0 be the given number of available machine registers; K_{est} be the estimated number of required registers from RRG. A non-negative integer N_0 is introduced. If $K_{est} - N_0 \leq K_0$ then we call the algorithm of RPS scheduling with spilling; else we first increase the estimated II (maybe also increase N_0 in some cases) to satisfy $K_{est} - N_0 \leq K_0$. After getting the software pipelined loop body, we can precisely compute the number of required registers. If the number is greater than K_0, then we increase the estimated II and call the al-

gorithm of RPS with spilling again. The process repeats until a software pipelined loop body is obtained whose register requirement is not greater than K_0.

We have not yet any theoretical analysis about N_0, but we believe that N_0 can be estimated empirically.

6 Preliminary Experimental Results

The effort to implement the algorithms presented in this paper is underway. Before getting extensive experimental tests, we select six examples to verify our algorithms. Except for example 1, the other five examples are selected from the Livermore benchmarks, shown in Table 1. 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 3.1(2).

Table 1. Experimental Examples

Example	L	MII	with lcd ?	Remarks
1	20	2	no	Figure 2.1(1)
2	22	3	no	Kernel 1
3	17	1	yes	Kernel 3
4	18	3	yes	Kernel 5
5	16	1	yes	Kernel 11
6	17	2	no	Kernel 12

note 1: L = the length of the longest dependence path in the loop body.
note 2: MII = the Minimal II.
note 3: lcd = loop-carried dependence.

Table 2 gives the register requirements for the optimal software pipelining performance by three scheduling approaches – DESP, RPS without spilling and RPS with spilling. Although DESP itself adopts the measures to reduce the register requirement when it determines the column-numbers, the algorithm of RPS without spilling can still obtain an improvement over DESP from 7.4% to 17.9% in register use without degradation of the optimal performance except for example 3 and 5. For example 3 and 5, no improvement in register use can be obtained since the initiation intervals (II) of the software pipelined loops are 1. For example 1 and 2, the algorithm of RPS with spilling can further obtain an improvement over DESP in register use of 22.2% and 23.1%, respectively, without degradation of the optimal performance.

Table 2. Register Requirement for Three Scheduling Approaches

Example	II	DESP	RPS without Spilling	RPS with Spilling
1	2	27	23	21
2	3	39	32	30
3	1	30	30	30
4	3	21	18	18
5	1	29	29	29
6	2	27	25	25

note: II = the initiation interval of the software pipelined loop.

The results of our algorithm for software pipelining with a limited number of registers are presented in Table 3 and Figure 7.1. Table 3 gives the initiation intervals (II) obtained by our

algorithm for the six examples when the number of available machine registers (K_0) is 8, 16 and 32, respectively. The relations between K_0 and the speedup are shown in Figure 7.1. The speedup is defined as L/II, where L is the length of the longest dependence path in the loop body (shown in Table 1), representing the optimal performance when we only exploit the ILP within the loop body. The results show that our algorithm can obtain the optimal speedup when $K_0 = 32$ (the minimal size of register file in the current ILP processors) and an average speedup of 2.34 when $K_0 = 8$, indicating that our algorithm can still efficiently exploit the ILP across iterations for loops even for a small register file ($K_0 = 8$).

Table 3. Software Pipelining with a Limited Number of Registers

(The Initiation Interval of the Software Pipelined Loop)

Example	The number of available machine registers:		
	8	16	32
1	7	3	2
2	16	8	3
3	7	3	1
4	8	4	3
5	5	2	1
6	9	4	2

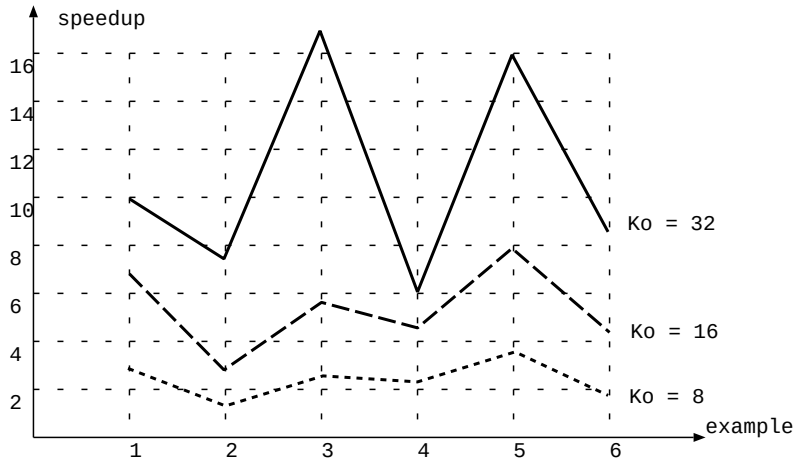


Figure 7.1 Software Pipelining with a Limited Number of Registers

7 Conclusion

This paper presents the Register Requirement Graph (RRG) which can dynamically reflect the register requirement during software pipelining. On the basis of the RRG, A Register-Pressure-Sensitive (RPS) scheduling technique is developed and the problem of register spilling for software pipelining is studied. We also present three algorithms – RPS without spilling, RPS with spilling and the software pipelining with a limited number of registers. The preliminary experimental results indicate that the first two algorithms can efficiently improve the register use without degradation of the optimal performance and the third can effectively exploit the ILP across iterations for loops even for those machines with a small register file.

The three algorithms are being implemented on our compiler testbed. We expect extensive experimental tests.

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 International 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] 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.

[21] 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.

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

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

[24] 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.

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

[26] 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.