計算機アーキテクチャ 130－12
（1998. 8. 5）

Register Spilling for Software Pipelining

Huan Liu† Dingchao Li ‡ Naohiro Ishii †

†Department of Intelligence and Computer Science, Nagoya Institute of Technology
‡Educational Center for information processing, Nagoya Institute of Technology
†E-mail: {hliu,ishii}@egg.ics.nitech.ac.jp
‡E-mail: liding@center.nitech.ac.jp

Software Pipelining is technique for exploiting instruction level parallelism in a variety of loops. In this paper, we improve the existing register-constrained software pipelining algorithm for efficiently schdeuling a loop with recurrences. Our strategy is to avoid spilling lifetimes in recurrence paths during the scheduling process. This enables software pipelining to reduce the register requirement without increasing the initiation interval of the loop's successive iterations.

Keywords: Fine grain parallel architectures, comipler optimization, software pipelining, modulo scheduling, register spilling.

1 Introduction

Loops are the major source of parallelism in scientific programs. To fully exploit instruction-level parallelism, software pipelining has been proposed as an efficient method for loop scheduling for high-performance VLIW and super-scalar architectures. Software pipelining derives a periodic pattern (or, a schedule) that overlaps instructions from different iterations of a loop body. The performance of such a schedule is measured by the initiation interval (II) of successive iterations. A smaller II corresponds to a shorter execution time.

Today there are variety of algorithms for software pipelining. An excellent survey of these algorithms can be found in [2]. However, many of them implicitly assume unlimited number of registers. If the number of real registers is not sufficient, there could be a problem since any valid schedule must fit in the available number of registers of a target machine. In this paper, we are interested in finding software-pipelined schedules in the face of a bounded number of registers. Our aim is to develop an efficient approach that inserts spill code during the software pipelining process to reduce register pressure when required.

Due to some characteristics of software pipelining, traditional spilling techniques are difficult to apply. Recently, the work by Llosa et al. [1] discussed how to add spill code for software pipelining. Their scheme selects a register to spill, based on the consideration of the largest lifetime, or the largest lifetime divided by the number of additional memory operations required. However, for the loops with recurrences, such a spilling decision may lengthen the initiation interval (II) of successive iterations. To solve this problem, the strategy described in this paper attempts to avoid spilling lifetimes in recurrence paths so that the II is not increased.

The rest of this paper is organized as follows. In Section 2, we establish machine and program models. In section 4, we review the existing register-constrained software pipelining scheme. In section 5, we describe an improvement for the efficient placement of spill code. In section 5, we report its effects on register pressure.

2 Machine model and program model

The target machine under consideration in this paper is a heterogeneous non-pipelined processor with different types of functional units, such as Integer, Floating Point (FP) Add, Load/Store, FP Multiply, and so on. To simplify the discussion in this paper, we consider that this machine has a single set of identical general-purpose registers. That is, it does not have different classes of registers: integer registers, address registers, and floating-point registers.

We also assume that a DAG is used to represent a loop iteration. In a DAG, vertece represent the operations in a program. An arc from one operation to another indicates that there is a data dependence between them. We consider here two types of data dependencies: loop-independent and loop-carried. A loop-independent dependency is between operations in the same iteration of a loop. A loop-carried dependency exists between operations in successive loop iterations.

Figure 1 shows a loop iteration and its DAG, in which loop-carried dependences are indicated by arcs with positive numbers beside them indicating their dependence.

loop body:

$$X[i] = Z[i] * Y[i-5] + Y[i] + Z[i-3]$$

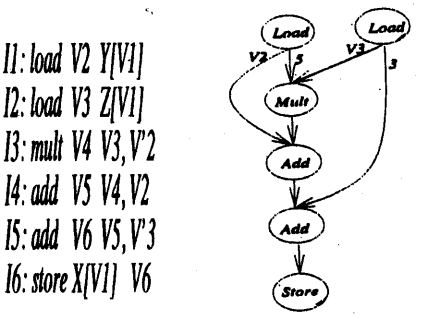

Figure 1: A sample program and its DAG

3 Overview of Modulo Scheduling

In a software pipelined loop, the schedule for an iteration is divided into stages so that the execution

of consecutive iterations that are in distinct stages is overlapped. The number of cycles per stage is the initiation interval(II).

The initiation interval II between two successive iterations is bounded either by loop-carried dependences in the DAG graph or by resource constraints of the architecture. The lower bound on the II is termed the Minimum Initiation Interval($MII = \max RecMII, ResMII$)[2].

4 Existing Register-Constrained Scheme

This section reviews the exsiting register spilling algorithm proposed in [1]. The algorithm of [1] employs the well-known modulo scheduling for implementing software pipelining (see Figure 2).

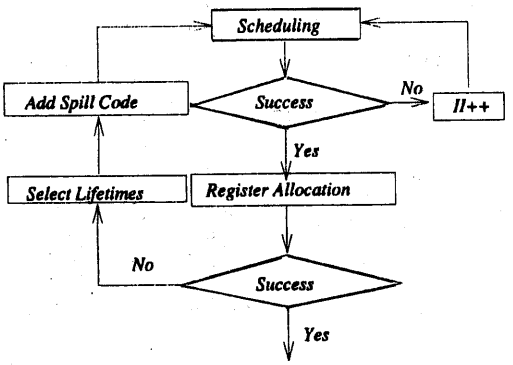

Figure 2: Flow diagram of alogrithm of [1]

When the number of real registers is not sufficient during modulo scheduling, the algorithm of [1] selects the lifetimes to spill using the following two strategies:

- Spill the longest lifetime regardless of the cost. It's called $Max(LT)$.
- Spill the largest Lifetime/Cost. It's called $Max(LT/Traf)$.

5 The algorithm

Let us now consider the examle in Figure 1 to illustrate the problem of the algorithm in [1]. For purpose of discussion in this section, we consider an architecture with 2 Load/Store units, 2 ALU units, and 1 Mult unit. Further let the execution time of an Integer add/sub be 1 cycle, while that of an FP multiply be 1 cycle. Assume also that load/store takes 1 cycle and only 14 registers could be available to execute the example program (Figure 1).

Now we do modulo scheduling and use the alogrithm of [1] to reduce register pressure.

First, we calcuate the II, RII is

$$RII = \max_{r \in R} \frac{C_r}{n_r}$$

Example program (Figure 1)'s RII is

$$RII = \max(\lceil \frac{3}{2} \rceil, \lceil \frac{2}{2} \rceil, \lceil \frac{1}{1} \rceil) = 2$$

In the example program (Figure 1), $II = RII = 2$.

Then we schedule example program (Figure 1) and allocate all operations to modulo resource table. The result is shown in the table 1.

Kernel table after scheduling has been shown in table 2

The register reqirements could be calculated by the following equation,

$$\sum RR = \sum_{i=1}^{n} RR$$

$$RR = \frac{LTDist_u}{II} + \frac{LTSch_u}{II}$$

$$\sum RR = 3(V1) + 5(V2) + 4(V3) + 1(V4) + 1(V5) + 1(V6) = 15$$

RR is represented register requirement.

For the example program (Figure 1), 15 registers would be needed by the regular scheduling. As we assumed the target machine could apply only 14 registers to execute example program(Figure 1), Lifetimes should be spilled out to reduce register pressure. By the alogrithm of [1], the largest lifetime V2 should be spilled out.

As the figure 3's show, value V2 has been spilled out. Operation Load and Mult should be scheduled simultanously as "complex operation"(see [1]). However, there isn't empty memory unit in table 2's first cycle. II should be increased if add spill code to value V2. The kernel table after adding spill code is shown in table 3.

In spite that the sum of register requirement have been decreased from 15 to 10, II has been

Cycle	Mem1	Mem2	ALU1	ALU2	MULT
1	$I2$				
2	$I1$				$I3 : n+2$
3			$I4$		
4			$I5$		
5		$I6$			
6					

Table 1: Modulo resource Table for sample program

Cycle	Mem1	Mem2	ALU1	ALU2	MULT
1	$I2 : n+2$	$I6 : n$	$I4 : n+1$		
2	$I1 : n+2$		$I5 : n+1$		$I3 : n+2$

Table 2: Kernal Table for sample program

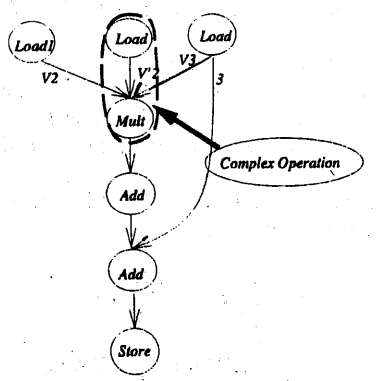

Figure 3: DAG of the example program after adding spill code by the alogrithm of [1]

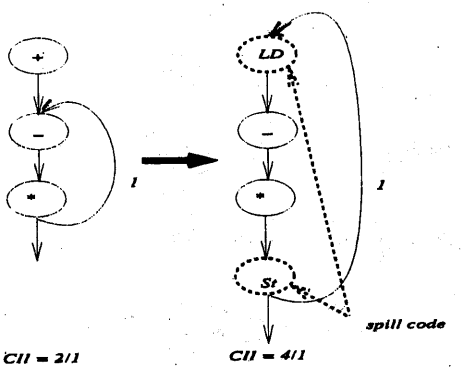

Figure 4: Adding a spill code to a value could cause CII increasing.

increased by one. However, the ideal result is that schedule the program without II increasing. The alogrithm of [1] could not give us ideal result in this example program.

Besides that, if there is a dependence distance in a program, adding spill code to the longest lifetime or the largest lifetime/cost could cause CII increasing, see Figure 4.

The above example and Figure 4 show that:

- If there isn't empty Load/Store unit in appropriate timing while the longest lifetime or the largest lifetime/cost is being spilled, the initiation interval should be increased.
- If the longest lifetime or the largest lifetime/cost is in recurrence path, inserting spill code may cause II increasing.

To solve this problem, we select all lifetimes that could be spilled as candidates and select a appropriate lifetime to be spilled out in order to avoid increasing initiation interval. First, we calculate a set of values could be spilled out. The equation is shown below.

$$\forall Lifetime \in LT, RR(Lifetime) \geq 2$$

LT is represented the set of all lifetimes, and RR is represented register requirement.

Then, we make the modulo scheduling and allocate register in the meantime.

At the last, we select a appropriate lifetime from the lifetimes set to be spilled out.

Figure 5 shows the flow diagram of the scheduling process by improved spill code technique, after register allocating, a appropriate lifetime will be selected to be spilled out to reduce register pressure when required.

Figure 6 shows how to add a spill code to mod-

Cycle	Mem1	Mem2	ALU1	ALU2	MULT
1	$I2:n+2$	$I6:n$	$I4:n+1$		
2	$I'1:n+2$		$I5:n+1$		
3	$I1:n+2$				$I3:n+2$

Table 3: The Kernel table after adding spill code to value V2

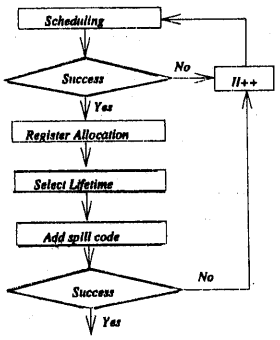

Figure 5: A flow diagram of scheduling process by improved spill code technique

ulo resource table by the improved spill code technique. In figure 6, RA is represented register available. We will select a lifetime to be spilled out until lifetimes set becomes empty or loop requires no more registers than available. If lifetimes set becomes empty and register pressure still exists, II should be increased by one and schedule the program again.

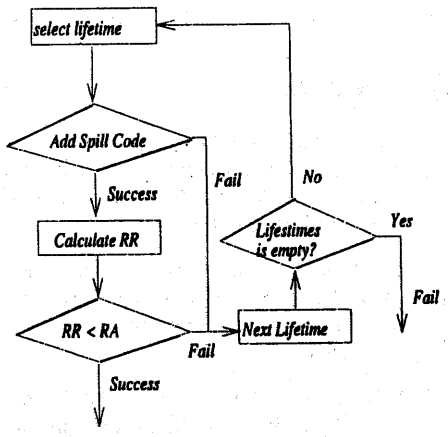

Figure 6: Flow diagram of adding spill code process by the Improved alogrithm

In example program (Figure 1), value V1, V2 and V3 could be spilled out, the sum of register requirement could not be decreased if spill out V1,

I1	:	load V2 Y[V1]
I2	:	load V3 Z[V1]
I3	:	mult V4 V3, V'2
I4	:	add V5 V4, V2
I'2	:	load V'3 Z[V1-3]
I5	:	add V6 V5, V'3
I6	:	store X[V1], V6

Table 4: Re-write sample program after spilling out value V3

spilling out V2 could cause increasing of II. As the result we try to spill out V3. As the table 4 shows, operation I'2 has been inserted to load value v3 before its consumer operation Add. the load operation I'2 in table 4 could be inserted into table 2's second cycle memory unit2. The DAG of example program (Figure 1) after spilling out V3 is shown in Figure 7. The modulo resource table after adding spill code is shown in table 5.

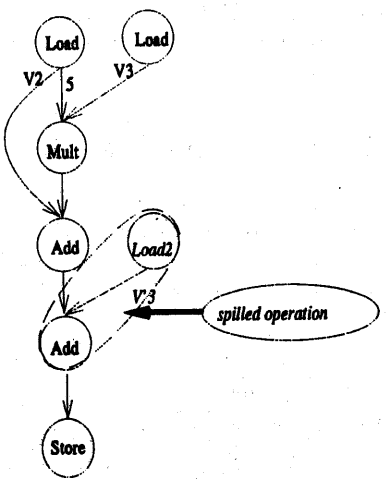

Figure 7: DAG of example program after spilling out V3

As the result, the sum of register requirements is 13 after spilling out lifetime V3.

$$\sum RR = 3(V1) + 5(V2) + 1(V3) + 1(V13) + 1(V4) + 1(V5) + 1(V6) = 13$$

Cycle	Mem1	Mem2	ALU1	ALU2	MULT
1	$I2:n+2$	$I6:n$	$I4:n+1$		
2	$I1:n+2$	$I'2:n+2$	$I5:n+1$		$I3:n+2$

Table 5: The Kernel table after adding spill code by Improved spill code technique

Avaliable Machine Register Number = 8			
Sample No.	II++	max LT or max $LT/Traf$	Improved alogrithm
1	$II=7$	$II=7$	$II=6$
2	$II=3$	$II=3$	$II=3$
3	$II=5$	$II=5$	$II=4$
4	$II=15$	$II=11$	$II=9$
5	$II=5$	$II=5$	$II=4$
6	$II=16$	$II=15$	$II=14$
7	$II=1$	$II=1$	$II=1$
8	$II=13$	$II=12$	$II=12$

Table 6: Comparison of $II++$, max LT or max $LT/Traf$ and Improved spill code technique

II of example program (Figure 1) is 2, and register requirements is 13. Comparing with the alogrithm of [1], improved alogrithm could give a program a flexible scheduling result without II increasing.

If spilling out a lifetime which is in a recurrence path could cause CII increasing, the lifetime should be removed from the candidate lifetimes set.

6 Preliminary Results

In order to verify our algorithm, we selected eight examples from the Livermore benchmarks (kernel 5,3,2,7,11,10,12 and 19) to do a preliminary experiment. The machine model we used in the experiment is shown in Table 7.

Unit	Operation	Number	Latency
Memory	Load/Store	2	1
ALU	Add/Sub	2	1
Mult	Mult	1	1

Table 7: Machine Model

Table 6 gives the initiation intervals obtained by the previous algorithm and our new algorithm. It is easy to see from this table that using the improved alogrithm could generate smaller II than the alogrithm of [1]. Specially, for the loops with recurrences (kernel5, 2, 7, 11, 10), spilling out appropriate lifetime (not limited to the longest lifetime or the largest lifetime/cost) correponds to the shortest execution time.

7 Conclusions

In this paper we have presented a simple spilling method for software pipelining in the face of a bounded number of registers. This method finds the appropriate lifetimes to spill for a loop with recurrences so that the initiation interval (II) is not increased. The preliminary experimental results indicate that the algorithm can efficently improve the performance of the existing spilling scheme.

References

[1] J.Llosa, M.Valero and E.Ayguade, "Heuristics for Register-constrained Software Pipelining", Proc. of the 1996 Int'l Conf. on Microarchitecture, pp. 250-261, 1996.

[2] B.R.Rau and J.A.Fisher "Instruction-level parallelism: History, overview and perspective", j. of Supercomputing, vol. 7, pp. 9-50, May 1993.

[3] V.H.Allan, R.Jones, R.Lee, S.J.Allan "Software Pipelining", ACM Computing Surveys, Sep. 1995.