Code Scheduling and Register Allocation in Large Basic Blocks

James R. Goodman

Computer Sciences Department
The University of Wisconsin-Madison
Madison, WI 53706

Wei-Chung Hsu[1]

Development Building
Cray Research Inc.
Chippewa Falls, WI 54729

Abstract

We discuss the issues about the interdependency between code scheduling and register allocation. We present two methods as solutions: (1) an integrated code scheduling technique; and (2) a DAG-driven register allocator. The integrated code scheduling method combines two scheduling techniques— one to reduce pipeline delays and the other to minimize register usage— into a single phase. By keeping track of the number of available registers, the scheduler can choose the appropriate scheduling technique to schedule a better code sequence. The DAG-driven register allocator uses a dependency graph to assist in assigning registers; it introduces much less extra dependency than does an ordinary register allocator. For large basic blocks, both approaches were shown to generate more efficient code sequences than conventional techniques in the simulations.

1. Introduction

Pipelining is a common technique used in high-performance computers [Kogg81]. It increases system performance by overlapping instruction execution. Ideally, more pipelined stages (*i.e.* a more finely segmented pipeline) means higher throughput. However, the presence of branch instructions and inter-instruction data dependencies often restricts the effectiveness of a long pipeline. Hardware techniques like out-of-order instruction execution [Toma67, Thor70] have been used occationally to alleviate the data dependency problem. However, they are not used widely because (1) they are expensive; (2) their complexity may slow down the clock rate. On the other hand, code scheduling, a software technique that rearranges the code sequence at compile time to reduce possible run-time delays, has been shown to be effective for improving the performance of pipelined processors [Henn83, Youn85, Gibb86, Weis87].

Code scheduling normally exploits existing parallelism in basic blocks. The code scheduler interleaves several independent instructions so that the latency (both memory latency and function unit latency) can be largely hidden. However, the observed parallelism in typical basic blocks is often limited to a factor of two to three [Tjad70]. This limits the effectiveness of a long pipeline. However, compiler techniques like loop unrolling [Dong79, Weis87] and trace scheduling [Fish81, Elli85] can be used to generate large basic blocks. Large basic blocks are common in scientific applications written for supercomputers, and they provide a better opportunity to exploit parallelism in basic blocks.

While code scheduling is effective in reducing pipeline interlocks and hiding memory latency, it creates a problem for register allocation. Code scheduling increases the time between a write to a register and the reads after the write. Having longer register lifetimes increases the number of simultaneously live registers, interfering with register allocation. Code scheduling may cause the register allocator to spill some registers. This is a major reason why researchers choose postpass code scheduling — code scheduling after the register allocation is done [Henn83, Gibb86] — rather than prepass scheduling. However, since the register allocator may inadvertently introduce dependencies by allocating the same register for unrelated instructions (so called storage-related dependency), postpass code scheduling is more restricted than prepass code scheduling. When basic blocks are small, this restriction makes little difference. But for large basic blocks and long pipelines, it may make a significant difference in performance.

We introduce two approaches to attack the interdependency problem between code scheduling and register allocation. Section 3 describes a code scheduling method which combines two code motion techniques — one to reduce pipeline delays and the other to minimize register usage— into a single phase. By keeping track of the number of available registers, the scheduler can choose a more appropriate scheduling technique. This method also considers spilling as a trade-off with runtime interlocks because for some heavily pipelined processors, spilling may not be more expensive than long interlocks. The other approach is based on a DAG-driven register allocator which uses a Dependency DAG to assist in assigning registers; it introduces much less extra dependency than does an ordinary register allocator. It will be described in section 4.

2. Background

2.1. Code Scheduling Constraints: The Dependency DAG

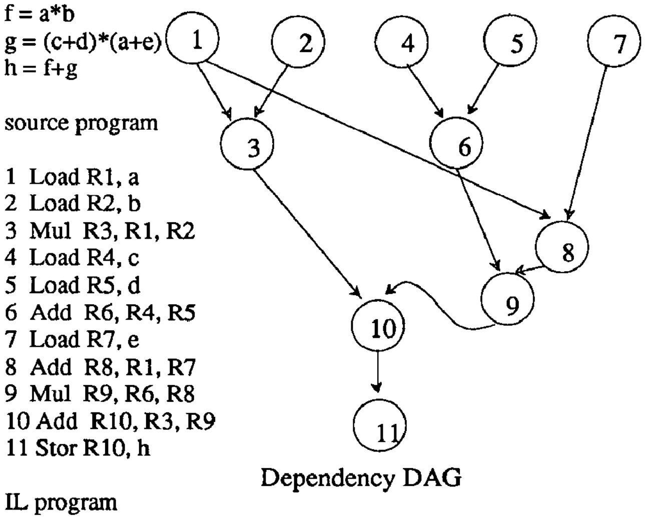
Code scheduling algorithms generally reorder instructions to improve program execution time [Henn83, Gibb86]. The ordering must preserve the original partial order imposed by operational precedence constraints. DAGs (Directed Acyclic Graphs) are normally used to represent program precedence constraints [Aho86]. A DAG defines legal evaluation orders within a basic block; nodes represent instructions, and edges represent serialization dependencies between instructions. An edge leading from instruction *A* to instruction *B* indicates that *A* must be executed before *B* in the scheduled code sequence. An example code sequence and its dependency DAG are shown in Figure 2.1. This example program will be used repeatedly in this paper.

2.2. The Use of Low Level Intermediate Languages

The use of intermediate languages (IL) simplifies code generation and optimizations. The PL.8 compiler [Ausl82] uses a low

[1] The work described here was done while this author was at University of Wisconsin--Madison.

level IL with an unlimited number of symbolic registers. In the register allocation phase, the symbolic registers are mapped into a limited number of physical registers. The IL used in this paper is similar to the assembly language of a load/store, register-register, three-address format machine. We will use it to illustrate examples.



```
f = a*b
g = (c+d)*(a+e)
h = f+g
```

source program

```
1 Load R1, a
2 Load R2, b
3 Mul R3, R1, R2
4 Load R4, c
5 Load R5, d
6 Add R6, R4, R5
7 Load R7, e
8 Add R8, R1, R7
9 Mul R9, R6, R8
10 Add R10, R3, R9
11 Stor R10, h
```

IL program

Figure 2.1 Example Program and its Dependency DAG

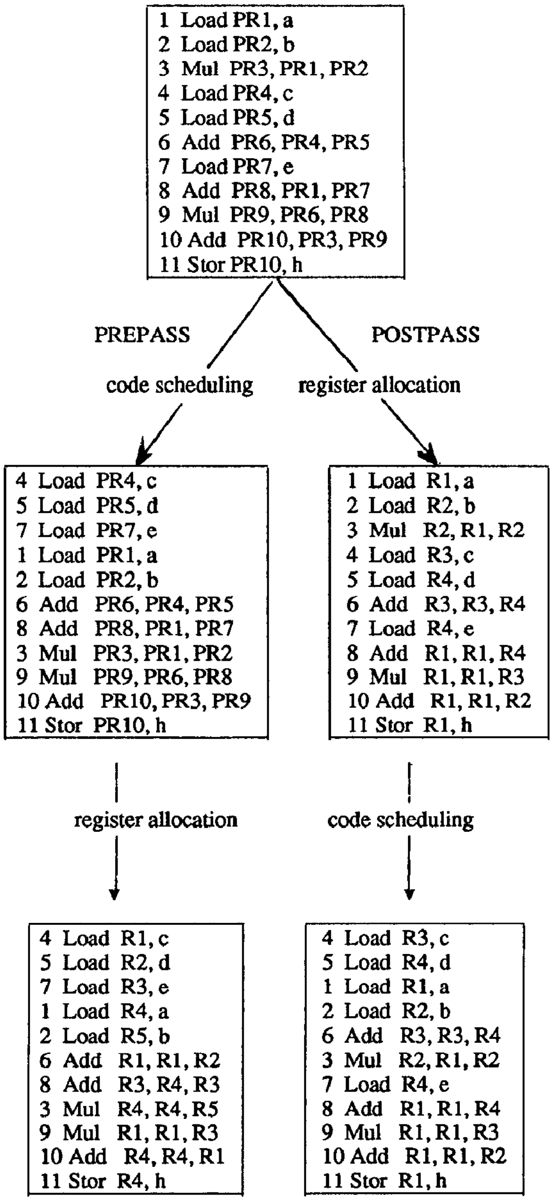
2.3. Prepass or Postpass?

Code scheduling can be applied to a program in IL either before the register allocation pass (prepass) or after register allocation (postpass). The advantage of prepass scheduling is that the full parallelism of the program could be exploited. Its drawback is the possibility of overusing registers which causes excessive register spilling. The increased instructions for register spilling will slow down the computation. Postpass scheduling does not increase spilled code, since register allocation has already been done. However, the register allocator is likely to assign the same register for unrelated instructions. The reuse of registers introduces new dependency constraints, making code scheduling more restricted. An example in Figure 2.2 illustrates the pros and cons of the above two scheduling policies. In Figure 2.2, the same program as in Figure 2.1 is used, PR means pseudo-registers.

In the above example, we assume a stack is used to manage the register pool: dead registers are returned to the top of the stack and new registers are allocated from the top of the stack[1]. The DAG in Figure 2.1 is used for prepass code scheduling. The original DAG, which is based on the dependencies of pseudo-registers, preserves maximal parallelism. After the register allocation, the reuse of registers forces new dependencies. For example, the reuse of register 4 in instruction 7 adds a write-after-read (WAR) dependency (or antidependency [Padu86]) from instruction 6 to 7 (Figure 2.3). This newly introduced dependency prevents instruction 7 from overlapping instruction 4 or 5, in postpass scheduling, introducing artificial pipeline delays. However, the code sequence of prepass code scheduling consumes five registers while the code sequence of postpass scheduling requires only four. If only four registers are available, the prepass code needs load and store instructions to spill registers.

2.4. Two Conflict Scheduling Techniques

Two code rearranging techniques could be applied during the optimization phases. They are: (1) code scheduling [Henn83, Gibb86, Youn85] to avoid delays in pipelined machines as we have discussed before — we call this technique CSP (Code Scheduling for Pipelined processors) for short, and (2) code reorganization [Davi86] to minimize the number of registers required — we call it



```
1 Load PR1, a
2 Load PR2, b
3 Mul PR3, PR1, PR2
4 Load PR4, c
5 Load PR5, d
6 Add PR6, PR4, PR5
7 Load PR7, e
8 Add PR8, PR1, PR7
9 Mul PR9, PR6, PR8
10 Add PR10, PR3, PR9
11 Stor PR10, h
```

```
4 Load PR4, c
5 Load PR5, d
7 Load PR7, e
1 Load PR1, a
2 Load PR2, b
6 Add PR6, PR4, PR5
8 Add PR8, PR1, PR7
3 Mul PR3, PR1, PR2
9 Mul PR9, PR6, PR8
10 Add PR10, PR3, PR9
11 Stor PR10, h
```

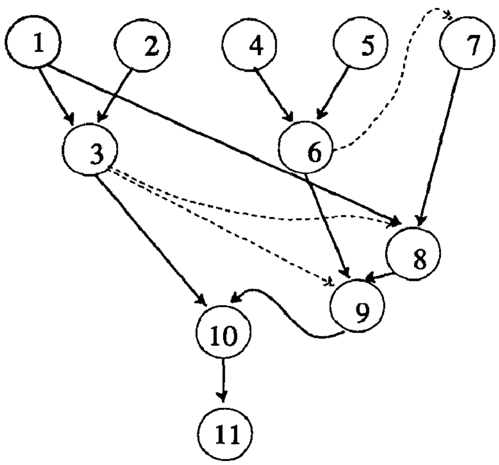
```
1 Load R1, a
2 Load R2, b
3 Mul R2, R1, R2
4 Load R3, c
5 Load R4, d
6 Add R3, R3, R4
7 Load R4, e
8 Add R1, R1, R4
9 Mul R1, R1, R3
10 Add R1, R1, R2
11 Stor R1, h
```

```
4 Load R1, c
5 Load R2, d
7 Load R3, e
1 Load R4, a
2 Load R5, b
6 Add R1, R1, R2
8 Add R3, R4, R3
3 Mul R4, R4, R5
9 Mul R1, R1, R3
10 Add R4, R4, R1
11 Stor R4, h
```

```
4 Load R3, c
5 Load R4, d
1 Load R1, a
2 Load R2, b
6 Add R3, R3, R4
3 Mul R2, R1, R2
7 Load R4, e
8 Add R1, R1, R4
9 Mul R1, R1, R3
10 Add R1, R1, R2
11 Stor R1, h
```

Figure 2.2 Prepass and Postpass Scheduling

[1] Alternate allocation policies are discussed in section 3.



solid lines -- original dependencies.

dashed lines -- dependencies added by register allocation

Figure 2.3 New Dependency Edges Added by Register Allocation

CSR (Code Scheduling to minimize Registers usage) for short. CSP could be applied before and/or after the register allocation phase while it only makes sense to apply CSR before register allocation. The use of CSP has been discussed; we now introduce the use of CSR in the following example.

As in Figure 2.2, a typical register allocation for the IL program requires four registers. The rearranged code sequence using CSR (as in Figure 2.4) needs only three registers.

The rearranging technique that generates the above code sequence has been used in the code generation phase to determine better evaluation orders of expression trees or DAGs. The well-known Sethi-Ullman algorithm [Seth70] generates the optimal evaluation order (using minimal number of registers) of expression trees. Heuristic algorithms are also available for DAGs (with common subexpressions) [Aho77]. Recently, Davidson [Davi86] has separated this optimization technique from the code generation phase and implemented it as an independent code reorganization technique. The key idea of this optimization is to prevent a register from holding a temporary too long. Hence the number of simultaneously live registers could be reduced.

CSP and CSR conflict with each other. CSP tends to increase the lifetime of each pseudo-register while CSR wants to shorten it.

```
4  Load PR4, c                                Load R1, c
5  Load PR5, d                                Load R2, d
6  Add  PR6, PR4, PR5                         Add  R1, R1, R2
1  Load PR1, a                after           Load R2, a
7  Load PR7, e            == register ==>     Load R3, e
8  Add  PR8, PR1, PR7        allocation       Add  R3, R2, R3
9  Mul  PR9, PR8, PR6                         Mul  R1, R3, R1
2  Load PR2, b                                Load R3, b
3  Mul  PR3, PR1, PR2                         Mul  R3, R2, R3
10 Add  PR10, PR3, PR9                        Add  R1, R3, R1
11 Stor PR10, h                               Stor R1, h
```

Figure 2.4 CSR Minimizing the Use of Registers

We attempt to integrate CSP and CSR into a single phase so that they will cooperate in generating a better code sequence.

3. Integrated Prepass Scheduling

The major disadvantage of prepass scheduling is that it may overuse registers causing register spilling. We propose to integrate CSP and CSR in prepass code scheduling to control register spilling. The basic idea is to keep track of the number of available registers during code scheduling. Since each issued instruction may create a new live register and terminate the lifetime of some registers, we can keep track of the number of available registers. When there are enough registers, the scheduler uses CSP to reduce pipeline delays. When the number of available register is getting low, the scheduler switches to CSR to control the use of registers. The following example explains this approach.

Example

Suppose the input program is the same as in Figure 2.1 and there are four[2] registers available for this program. Our new code scheduler will schedule the program in a sequence like:

```
4 Load PR4, c
5 Load PR5, d
7 Load PR7, e
1 Load PR1, a
```

The scheduler must now choose between issuing instruction 2, which activates one register, and instruction 6, which frees one net register. Since the available registers have been used up, CSR takes charge of scheduling, and issues instruction 6. We then return to CSP and instruction 2 is issued after instruction 6. The complete reorganized code sequence is as in Figure 3.1.

Notice that this code sequence uses four registers, the same number as the postpass code sequence used. Compared to the postpass code sequence as in Figure 2.2, however, this code sequence has fewer runtime interlocks.

3.1. Implementation Notes

3.1.1. CSP, CSR and AVLREG

There are two major parts in our approach: CSP and CSR. The use of CSP in our work is based on the work by Young [Youn85]. We also take ideas from others [Gibb86, Henn83] to improve the CSP algorithm. Young [Youn85] assumes the target machine has multiple functional units whose pipelines vary in

```
4  Load PR4, c                                Load R1, c
5  Load PR5, d                                Load R2, d
7  Load PR7, e                                Load R3, e
1  Load PR1, a                                Load R4, a
6  Add  PR6, PR4, PR5          after          Add  R1, R1, R2
2  Load PR2, b             == register ==>    Load R2, b
8  Add  PR8, PR1, PR7         allocation      Add  R3, R4, R3
3  Mul  PR3, PR1, PR2                         Mul  R4, R4, R2
9  Mul  PR9, PR6, PR8                         Mul  R1, R1, R3
10 Add  PR10, PR3, PR9                        Add  R4, R4, R1
11 Stor PR10, h                               Stor R4, h
```

Figure 3.1 Code Sequence Using Integrated Scheduling

[2] A machine typically has eight or 16 general purpose registers. However, we assume that other registers have already been preallocated to frequently used variables or constants and thus only four are left for this basic block.

length. Instructions complete whenever they leave their particular functional unit pipelines. The estimated execution time of each instruction is used to compute the cumulative cost of each node in the DAG. This cumulative cost identifies which node is on the critical path during instruction scheduling. Instructions are scheduled in a topological sort order of the DAG. Nodes on the current critical path have higher issue priority. In contrast to [Henn83], hardware interlocks are assumed rather than using software to enforce interlocks.

The CSR used in our approach is different from earlier work [Seth70, Aho77], which determines the complete evaluation order of an expression tree or a DAG. In our approach, when CSR is called, the evaluation order of the DAG has been partially determined (some nodes have been issued). The goal of CSR at this point is to find the next instruction which will not increase the number of live registers, or if possible, decrease that number. Our CSR does not decide the total evaluation order. The basic approach of our CSR is to find an instruction that frees more registers than the number of live registers it creates. When no such instructions exist, the scheduler looks for instructions on partially evaluated paths, since once the partially evaluated path is fully evaluated, registers may be freed.

Switching between CSP and CSR is driven by the number of available register, AVLREG. CSP is responsible for code scheduling most of the time. When AVLREG drops below a threshold (say, one) CSR is invoked. After AVLREG is restored to an acceptable value, CSP resumes scheduling. AVLREG is initially determined by the total number of registers minus the number of registers live-on-entry. Global data flow analysis [Aho86] can supply the information of registers live-on-entry. Reference counting is used to determine when pseudo-registers are dead and can be freed. We increase AVLREG when there are freed registers, and decrease AVLREG when instructions create live registers.

Renaming techniques [Padu86] have been used to enforce single assignments — every pseudo-register is written only once statically in a basic block — in order to maintain maximal scheduling flexibility.

3.1.2. Interlock Checking at Scheduling Time

In load/store, register-register architectures, it is easy to resolve all interlocks at the instruction issue stage [Kogg81, Cray82]. The code scheduler can use instruction execution time information to estimate possible runtime interlocks, and schedule instructions to avoid as many interlocks as possible.

3.1.3. Leader Set and Ready Set

A leader of a DAG is a vertex with no predecessors. An instruction can not be issued until it becomes a leader. As instructions are issued, their nodes are removed from the DAG and some successor nodes become new leaders. All leaders are maintained in a *leader set*. Instructions in the leader set lacking interlocks with previously issued instructions are promoted from the leader set to a *ready set*. All the instructions in the ready set are ready to be issued.

3.1.4. Integrated Scheduling Algorithm

(1) Rename pseudo-registers to enforce single assignment.

(2) Input a basic block, create the DAG and calculate the reference count of each pseudo-register.

(3) Compute the cumulative cost of each node in reverse topological sort order.

(4) Issue instructions in topological sort order.

Details of step 4:

```
  4.0 Calculate the leader set.

while (leader set or ready set is not empty) do

  4.1 Move nodes without interlocks from leader set to ready set.
  4.2 if (AVLREG > threshold value) then
        if (ready set is not empty) then
          select one node from ready set with maximum
          cumulative cost.
        else
          select one node from leader set with maximum
          cumulative cost.
        endif
      else {invoke CSR}
        if there are nodes in ready set that can free registers
        then
          select one node which frees the most registers.
          if there are more than one such node then
            select one with maximum cumulative cost.
          endif
        else
          if there are nodes in leader set that can free registers
            then
            select one which frees the most registers.
            if there are more than one such node then
              select one that has the fewest interlocks.
            endif
          else
            find a partially evaluated path, (for example,
                one of its RAW dependency has been lifted)
            select one node from the leaders of this path.
            if there are no such partially evaluated paths then
              select any one node from the ready set
              or from the leader set if the ready set is empty.
            endif
          endif
        endif
      endif

  4.3 Issue the selected instruction
      if the issued instruction creates one live register then
        decrement AVLREG by 1.
      for each pseudo-register referenced in this instruction do
        decrement its reference count by 1
        if the reference count drops to 0 then
          increment AVLREG by 1.
        endif
      end for
      Remove this instruction from the DAG
      Remove all dependencies caused by this instruction
      Reserve the destination register in a reservation table.

  4.4 Insert new leaders into the leader set

end while
```

The input program used in this example is the same one as used in section 2. In addition, we assume the following timing for the relevant functions.

Function	Timing(clock periods)
Load	4
Store	1
Add	2
Multiply	3

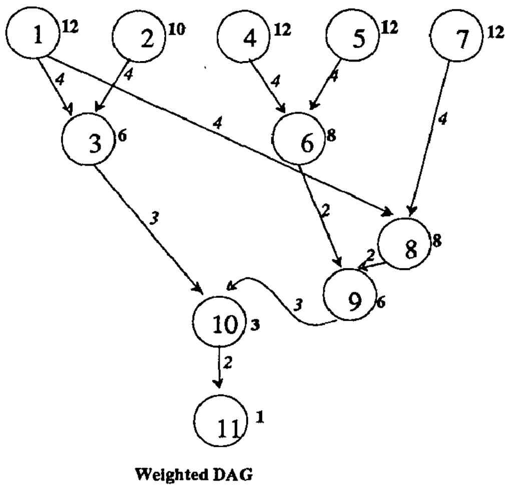
We assume that the initial value of AVLREG is 4. The weighted DAG is shown in the Figure 3.2 The code sequences generated by prepass code scheduling, postpass code scheduling, and our algorithm are shown in Figure 3.3. Notice that, since the number of available registers is 4, prepass code scheduling incurs spilling costs.

3.1.5. A Variation on Profitable Register Spilling

In the above algorithm, we have assumed that register spilling is more costly than run-time interlocks. This assumption may not be true for highly pipelined machines.

(1) In highly pipelined machines, a pipeline interlock could be very long compared to the issuance of a couple of spill instructions. Also, for machines that have a back up register file like the B/T registers in Cray-1 and Cray-XMP, spilling is not expensive.

(2) Spill code can often be scheduled to run in otherwise wasted cycles.



boldface numbers associated with nodes are cumulative costs
italic numbers associated with edges are execution time estimates

Figure 3.2 Weighted DAG Identifying Critical Paths

(3) Registers containing loop invariants can be spilled at a lower cost.

We have developed a variation of the algorithm to consider profitable register spilling. The variation operates as follows: when available registers are running out and the next selected instruction has a long interlock with a previously issued instruction, the scheduler checks if there is a live pseudo-register which could be spilled at a low cost. If there is at least one such register, the scheduler will revert to CSP scheduling to favor instructions having no interlocks and/or are on the critical evaluation path. Because of the uncertainties involved in predicting how inserted load instructions will interfere with subsequent register usage, we have only attempted to guess at a good threshold value for determining when spilling is worthwhile.

3.2. Simulation Studies and Discussion

3.2.1. Simulations

In this section, we show some experimental results concerning the effectiveness of our new approach. An interpreter and a performance simulator have been built to evaluate how fast instructions can be issued for a hypothetical machine. The hypothetical machine architecture has a load/store, register-oriented, three-address instruction format. It has a single general purpose register file. Its pipelined implementation is shown in the Figure 3.4. The number of general purpose registers and the degree of pipelining of the machine can be varied by changing the parameters in a profile. We assume the hypothetical machine has hardware hazard detection and an interlock mechanism.

We use the first twelve Livermore loops [McMa72] for benchmark programs. Loop unrolling [Dong79, Weis87] have been used to obtain large basic blocks. Since unrolled loops may overflow an instruction buffer, decreasing performance, we unroll loops until their program size is a little less than some predefined limit. In our simulation, this limit is 32 instructions. Although 32 is relatively small compared to the instruction buffer in modern supercomputers (CRAY-1 has a buffer size of 256 instruction parcels, CRAY-XMP has a size of 512), it is large enough to study the interdependency between code scheduling and register allocation. All of the loops are translated into the IL of the hypothetical machine and optimized based on standard optimization techniques [Aho86, Ausl82]. In

Prepass	Postpass	Integrated
1 Load R1, a	4 Load R3, c	4 Load R2, c
4 Load R2, c	5 Load R4, d	5 Load R3, d
5 Load R3, d	1 Load R1, a	7 Load R4, e
7 Load R4, e	2 Load R2, b	1 Load R1, a
s Stor R4, templ	6 Add R3, R3, R4	6 Add R2, R2, R3
2 Load R4, b	3 Mul R2, R1, R2	2 Load R3, b
6 Add R2, R2, R3	7 Load R4, e	8 Add R4, R1, R4
s Load R3, templ	8 Add R1, R1, R4	3 Mul R1, R1, R3
8 Add R3, R1, R3	9 Mul R1, R1, R3	9 Mul R2, R2, R4
3 Mul R1, R1, R4	10 Add R1, R1, R2	10 Add R1, R1, R2
9 Mul R2, R2, R3	11 Stor R1, h	11 Stor R1, h
10 Add R1, R1, R2		
11 Stor R1, h		
(22 cycles)	(20 cycles)	(17 cycles)

Figure 3.3 Comparisons of Scheduled Code Sequences

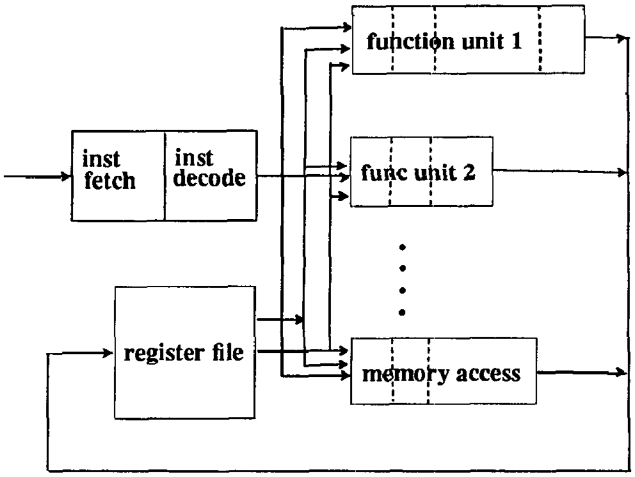


Figure 3.4 Pipelined Implementation of Our Model Architecture

order to simplify the simulation, all floating point operations have been changed to integer operations.

Different approaches for code scheduling that have been tested in the simulation are:

(1) Prepass: IL --> CS --> RA

(2) Twopass: IL --> CS --> RA --> CS

> code scheduling is performed both before and after register allocation. The second pass scheduling is primarily for the inserted load/stores introduced by register spilling.

(3) Postpass: IL --> RA --> CS

(4) Postpass with round-robin register allocation (PostRR for short):

> The regular register allocation uses a stack to manage register reuse. Some papers [Henn83, Youn85] suggest using round-robin allocation, which cycling through registers, to reduce the inadvertent dependencies introduced by register reuses. Hence we implemented PostRR as a variation of general Postpass. The code scheduling algorithm for approaches (1) through (4) is essentially the CSP part as discussed in previous section.

(5) GoodWay0 (GW0): As described in the previous section.

> This algorithm does prepass code scheduling with appropriate control of register usages.

(6) GoodWay1 (GW1): A variation of GW0.

> This version considers spilling a register as an alternative when the next issuing instruction has long interlocks with previously issued instructions.

All the above approaches used the same register allocator, which uses a replacement-based algorithm [Hsu87].

In Figure 3.5, we present the relative performance of six different approaches. The performance measure is the number of clock cycles needed to issue and execute a program. The number of available registers is varied from 4 to 30. The machine is assumed to be heavily pipelined (HP for short), similar to the CRAY-1 [Cray82]. We assume 11 clock periods (CP) for a load, 3 CPs for an integer add, 6 CPs for an integer multiply, and so on. Figure 3.6 is similar to Figure 3.5 except that the machine has a slower clock rate. In this machine, we assume a load takes 6 CPs, an add takes 2 CPs, an multiply takes 3 CPs, and so on.

Figures 3.5 and 3.7 base on the same machine assumptions, as do Figures 3.6 and 3.8. In Figures 3.7 and 3.8, the measure is the number of instructions. Since prepass code scheduling often results in register spilling, the sizes of its resulting programs are usually larger than programs of postpass scheduling and our scheduling scheme.

3.2.2. Discussion

Using Figures 3.5 through 3.8, we make the following observations.

(1) In Figures 3.5 and 3.6, prepass code scheduling usually has better performance than postpass scheduling unless the number of available registers is very low. This is because prepass scheduling has much better flexibility to schedule code, especially when more parallelism exists. However, we should not conclude that prepass scheduling is better than postpass scheduling. In Figures 3.7 and 3.8, the prepass

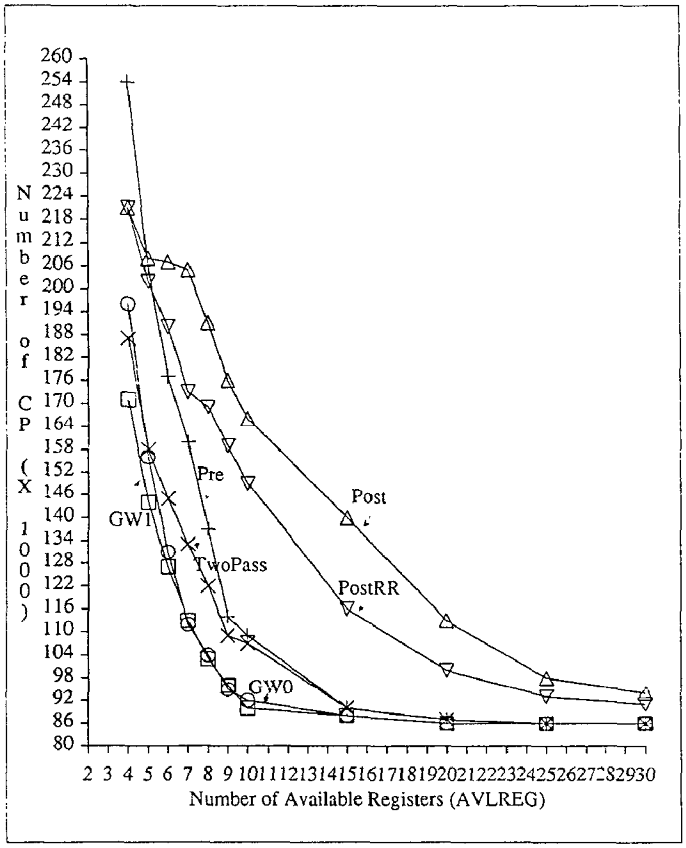


Figure 3.5 Comparisons of Speed (Highly Pipelined Model)

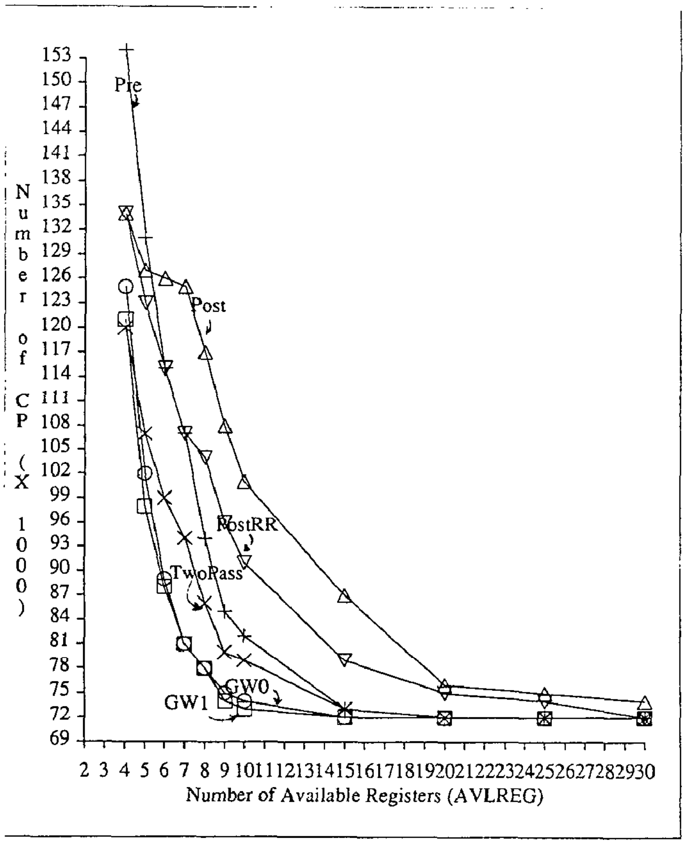


Figure 3.6 Comparisons of Speed (Medium Pipelined Model)

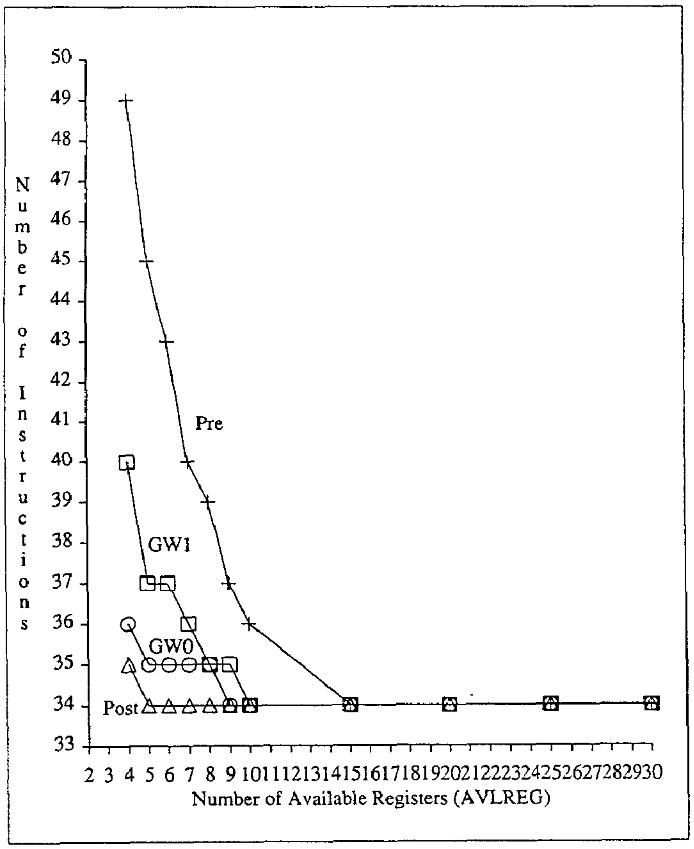


Figure 3.7 Comparisons of Code Size (Highly Pipelined Model)

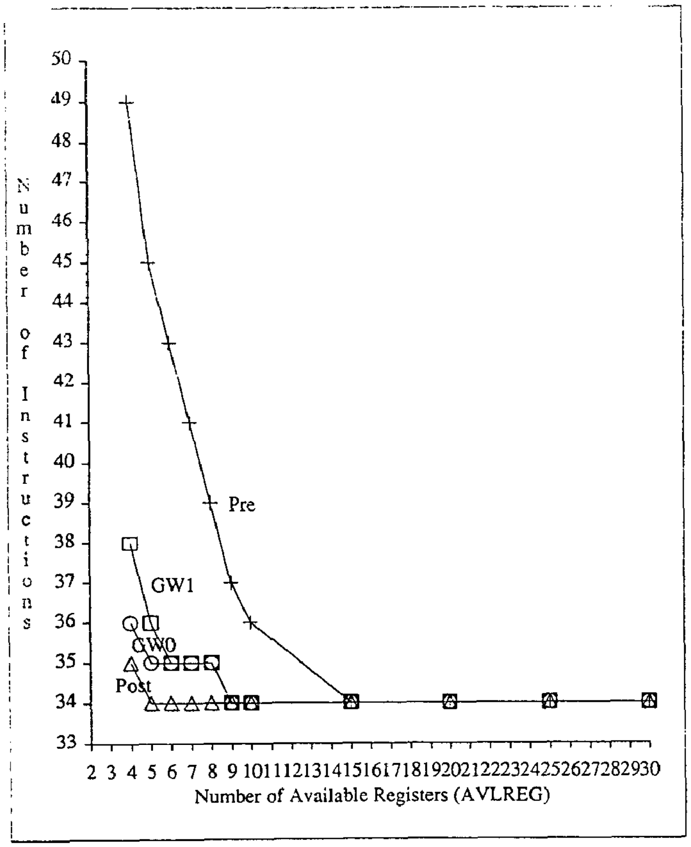


Figure 3.8 Comparisons of Code Size (Medium Pipelined Model)

scheduled programs have significantly larger size than the programs of postpass scheduling. Larger programs may execute more slowly, since larger loops are more likely to overflow the instruction buffer, and inserted load/store instructions increase the number of memory fetches and stores. Additional load/stores will slow the computation seriously if memory bandwidth is the performance bottleneck (this is likely in a multiprocessor system).

Our approach GW0 outperforms both prepass scheduling and postpass scheduling as shown in Figure 3.5 and 3.6 while the additional load/store instructions of GW0 are only slightly (less than 3%) more than postpass scheduling, as shown in Figure 3.7 and 3.8.

(2) In theory, if there are an infinite number of registers, all the different approaches have the same performance. As shown in in Figures 3.5 and 3.6, all the curves level off to the same point when the number of available registers becomes large. However, minimizing the number of registers is critical for designing high-performance computers; for register access to be fast, the size of the register file should be small [Henn84]. Hence, the more important question is how to use limited registers efficiently. As shown in Figures 3.5 and 3.6, the curve of our approach levels off more quickly than others, implying that better scheduling can use a (relatively) small number of registers efficiently.

A register file can be used to hold temporaries and frequently used variables. An efficient algorithm uses fewer registers for temporaries, leaving more registers for frequently used constants and variables. The number of registers is well-defined in an architecture. Yet the architecture can have quite different implementations. A highly pipelined implementation requires more registers for temporaries so that interlocks can be reduced. When a highly pipelined implementation is used, the approach of using better algorithms to make effective use of the register file is more favorable than the approach of redesigning the architecture with more registers.

Advances in silicon technologies may make it relatively easy to have a large number of registers on-chip in the near future. However, for GaAs technology, which is much faster than silicon technology, the amount of on-chip memory allowed is very limited [Milu86]. In a memory hierarchy, the space in the top level is always limited, and efficient algorithms are necessary to make effective use of the scarce space.

(3) Some compilers designed for pipelined processors use round-robin register allocation, which cycles through the registers available for use. Intuitively, this allocation policy seems to avoid the situation of having a long evaluation path due to the intensive reuse of certain registers. In Figure 3.6, postpass scheduling with round-robin allocation does outperform stack allocation most of the time. It is also true in Figure 3.5 except when the number of available register is low. Since every reuse of a register will add a storage-related dependencies to the DAG, may combine two parallel evaluation paths into a sequential one, without having the detailed information of the DAG, no allocation policy will be uniformly superior to others in balancing the length of merged evaluation paths. One alternative to the current approach is to provide DAG information to the register allocator. With the DAG information, the register allocator may be able to reuse registers in a way such that the depth of the new DAG can be minimized. Such an approach will be described in the next section.

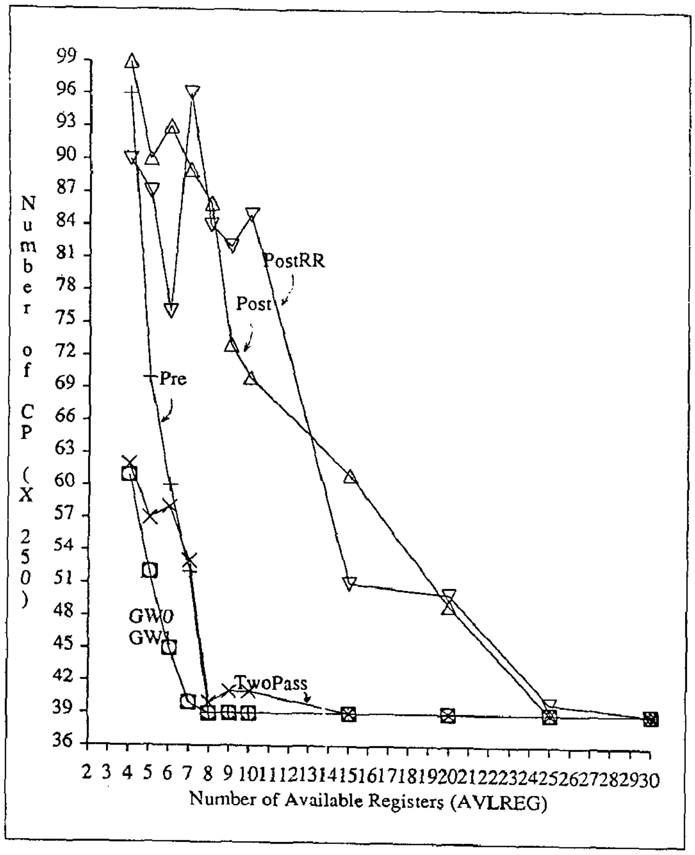


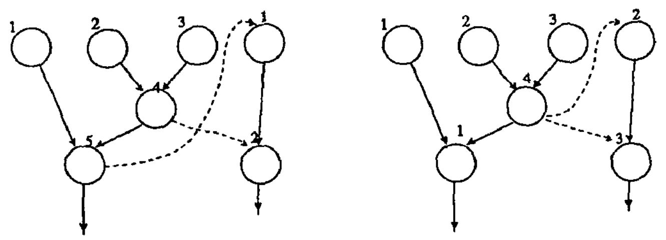
Figure 3.9 Anomalies (from Loop 3)

Some anomalies exist in Twopass and Postpass scheduling experiments. Since they can not be observed from Figure 3.5 and 3.6 which are figures averaged over all loops, Figure 3.9 is used to present the curves of a single loop. The following discussions are based on Figure 3.9.

(1) The Post curve in Figure 3.9 exhibits a deterioration in performance when the number of registers is increased from five to six. The PostRR curve exhibits a similar anomaly for the transition from six to seven and nine to ten registers. These anomalies result from variations in induced dependencies, as illustrated in Figure 3.10. The DAG on the left hand side has five available registers while the DAG on the right hand side has only four. Round-robin register allocation was used. Notice that after register allocation, the left DAG is more restricted than the right DAG. Because the right tree in the left DAG can not be interleaved with the left tree any more, instructions are forced to be executed in a sequential order.

Figure 3.9 also showed that sometimes Post (stack register allocation) outperforms PostRR (round-robin register allocation) and vise versa. This supports our previous assertion that no allocation policy will be superior to others without detailed DAG information.

(2) Two-pass scheduling is used to improve the performance of prepass scheduling. The second scheduling pass is used to schedule the inserted load/store instructions. In Figures 3.9, the anomaly of the Twopass curve occurs when the number of registers is eight, nine and ten. Not only is the performance poorer than the single pass scheduling but the curve also exhibits a deterioration in performance when the number of registers is increased. When there are enough registers, no inserted load/stores are required for spilling. When there are no inserted load and store instructions, a second pass scheduling is useless. The performance deterioration anomaly is created by the effect of register allocation.



solid lines -- original dependencies.

dashed lines -- dependencies added by register allocation

Figure 3.10 One Explanation of the Anomaly in Postpass Scheduling

4. DAG-Driven Register Allocation

The previous section suggests solving the interdependency problem with an integrated prepass scheduling method. This section discusses an approach which is for the postpass scheduling. We use the dependency DAG to direct register allocation so that very little storage-related dependency are introduced. Two advantages to use postpass scheduling: (1) a postpass scheduler can be applied both to code output from a compiler and to hand-written assembly-language code; (2) unlike prepass scheduling, postpass scheduling will never introduce additional spill code. If minimizing load/stores is the major concern, then postpass scheduling should be favored.

4.1. Balancing DAG Reconstruction

In order to introduce the concept of the DAG-driven register allocation, we define two terms: *width* and *height* of a DAG. The width of a DAG is defined as the maximal number of mutually independent nodes which need a destination register (a store instruction, for example, does not need a destination register), and the *height* of a DAG is the length of its longest path. Since we use the single assignment rule in naming temporaries, the dependency DAG will have a greatest width exposing maximum parallelism. If the number of real registers is larger than the width of the DAG, the shape of the DAG can remain unchanged during register allocation. Otherwise, the register allocator will reduce the width of the DAG to be smaller than or equal to the number of real registers by reusing registers. While the width is reduced, the height is increased since each reuse of registers may merge two evaluation paths into one. The greater the height, the longer the critical path. The longer the critical path, the less efficient the code scheduling. Therefore, our DAG-driven register allocator is trying to minimize the height of the reconstructed DAG. Two strategies to control the growth of the height are: exploiting free WAR dependencies and balancing the growth of the DAG.

4.1.1. Free WAR Dependencies

The reuse of a register creates new dependencies, primarily write-after-read (WAR) dependencies. We have explained that the added WAR dependencies reduce available parallelism and result in less effective code scheduling in section 2. We assumed a pipeline structure in which the operand registers are read at the time an instruction is issued (see Figure 3.4). So long as instructions issue in order (at run time), WAR hazards at register level will never occur. Therefore, the WAR dependency edges are essentially used to enforce the logical order of instructions. We assign WAR dependency a cost of 1, the lowest cost of all dependencies.

Figure 4.1 shows the DAG of the following program segment. Assume there are 5 registers. As the register allocator reads the

```
1 Load PR1, a
2 Load PR2, b
3 Add  PR3, PR1, PR2
4 Load PR4, c
5 Sub  PR5, PR4, PR1
6 Addi PR6, PR5, #4
7 Mul  PR7, PR5, PR1
  .
  .
```

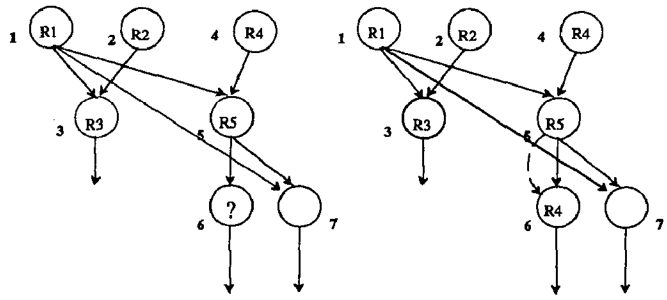


Figure 4.1 Free WAR Dependencies

program, it allocates register R1 through R5 for the destination registers of instruction 1 to instruction 5. At instruction 6, since the available registers have been used up, the register allocator tries to find a dead register to replace. Two registers, register 2 and register 4, are dead before instruction 4. As in Figure 4.1, the reuse of register 4 at instruction 6 introduces a WAR dependency which is represented as a dashed line from instruction node 5 to 4. This dependency is redundant since the logical order of instructions 4, 5 and 6 can be enforced by already existing dependency edges. But if register 2 is allocated rather than register 4 to instruction 6, the added dependency (from node 3 to node 6) is not redundant. Redundant dependencies are *free dependencies*, since they will not increase the height of the DAG.

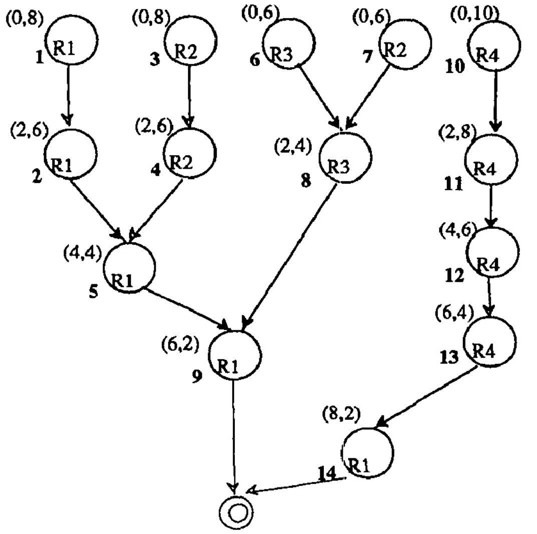
To minimize the increase in height of the DAG, the register allocator will first select a dead register to replace such that only redundant dependencies are introduced. In other words, the uses of the dead register are on the dependent path of the current instruction. For example, in Figure 4.1, when the register allocator allocates register 4 for instruction 6, the last use of register 4 is instruction 5, which is on the dependent path of instruction 4. Therefore, this allocation introduces no additional dependencies.

4.1.2. Balancing the Growth of the DAG

All register replacements which add new dependencies may increase the height of the DAG. When there are no free dependencies, we allocate registers based on *earliest issue time* and *earliest finish time.* The earliest issue time (EIT for short) of a node is the maximal path cost from the beginning of the DAG to the node. The earliest finish time (EFT for short) of a node is the maximal path cost from the node to the end of the DAG. As the names suggest, the EIT of a node indicates the earliest possible issue time of that instruction from the beginning of the execution of the DAG. The EFT of a node indicates the earliest possible finish time from the issue of that instruction to the end of the execution of the DAG. Figure 4.2 shows the EIT and EFT attributes of each node in a DAG. If the register allocator assigns two independent paths to share a register, then a new WAR dependency connects the two paths into one long path. For example, in Figure 4.1, if the register allocator assigns register 2 to instruction 6, then instruction node 6 is connected with instruction node 3 by a WAR dependency edge. The maximal cost of this new path is apparently $(EIT_3 + EFI_6 + 1)$, where 1 is the cost of the WAR dependency edge.

The key idea in minimizing the height of the DAG is to balance the growth of the DAG. The allocator tries not to connect two nodes such that one has a large EIT and the other has a large EFT. If the current instruction has a high EFT, then the allocator would select a dead register such that all the nodes the current instruction will connect to have a small EIT. But how does the allocator know if the EFT of the current instruction is relatively large or small? This suggests that the allocator should look at all the unallocated instructions, especially the leader nodes, to determine where the current instruction stands.

Statically computed EIT cannot be used directly in allocating registers since each register replacement may change the EIT of some nodes. The details of implementation can be found elsewhere [Hsu87].



numbers in parenthesis are (EIT, EFT)
assuming each instruction takes 2 clock periods
the double circle is a pseudo node which
indicates the end of the DAG

Figure 4.2 Computations of EIT and EFT

4.1.3. DAG-Driven Allocation Algorithm

(1) Rename pseudo-registers to enforce single assignment

(2) Use CSR to reduce the number of simultaneously live registers

(3) Read in the basic block, build the DAG and set up a hash table of register reference histories

(4) Compute the EFT of each node, set up the adjacency matrix representing the dependency relation

(5) Register allocation:

```
while there are instructions to be allocated do

    foreach pseudo-register in the instruction, in the order of
    first operand, second operand, destination do
```

```
while there are instructions to be allocated do

  foreach pseudo-register in the instruction, in the order of
  first operand, second operand, destination do

  if (pseudo-register Miss) then

     if (Write Miss) then

        if (there are dead registers on the dependent path)
        then
           select one and replace;
           go to 2;
        endif
     endif
     if (there are dead registers) then
        sort them in ascending order on the EIT field,
        count how many remaining leaders have a higher EFT
           than the current instruction into a variable P,
        if (P+1 is greater than the number of dead registers)
        then
           select the dead register with highest EIT to replace;
        else
           select the P+1st dead register to replace;
        endif
     else
        normal replacements;
     endif
  endif
endfor

  1: update dependent relation incrementally;

  2: update EIT of real registers;

endwhile
```

4.2. The Performance of DAG-driven Register Allocation

Our test results, based on the same test environment described in section 3, show DAG-driven register allocation significantly improves the performance of postpass code schedulings (cf. Figures 4.3 and 4.4). We also compared DAG-driven register allocation to the integrated prepass scheduling. For the highly pipelined model (Figure 4.3), the integrated prepass scheduling approach that considers profitable spilling slightly outperforms the DAG-driven register allocation approach. However, in the medium pipelined model (Figure 4.4), DAG-driven allocation approach outperforms all the others. We explain our results as follows: for the highly pipelined model, where an interlock could be relatively expensive, spilling may be profitable. The integrated prepass scheduling can easily accommodate profitable spilling, this flexibility creates an opportunity for a better performance than the DAG-driven allocation approach. The DAG-driven allocation approach has the advantage that it does not increase the code size at all. Therefore, it is favored by machines which discourage code spilling.

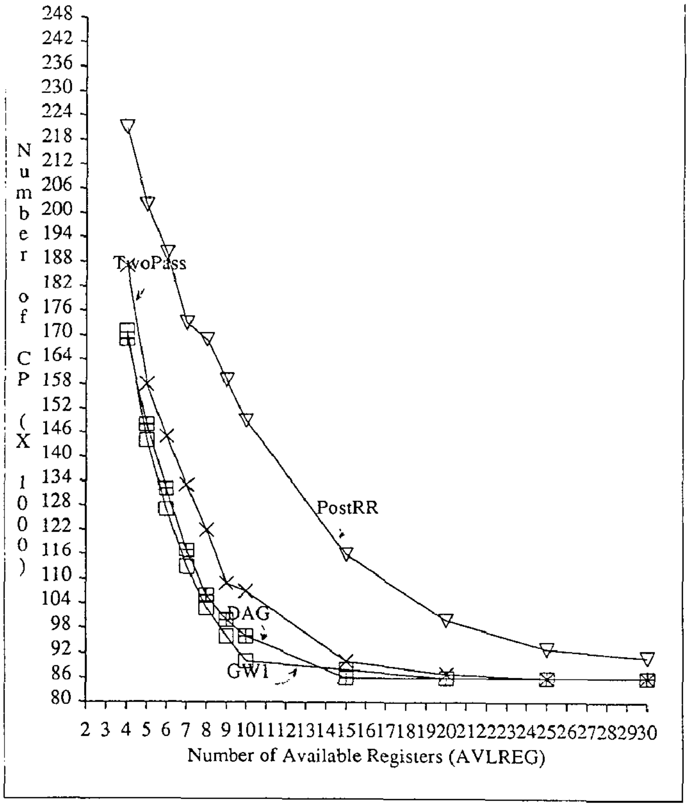


Figure 4.3 Performance of DAG-Driven Allocation (Highly Pipelined)

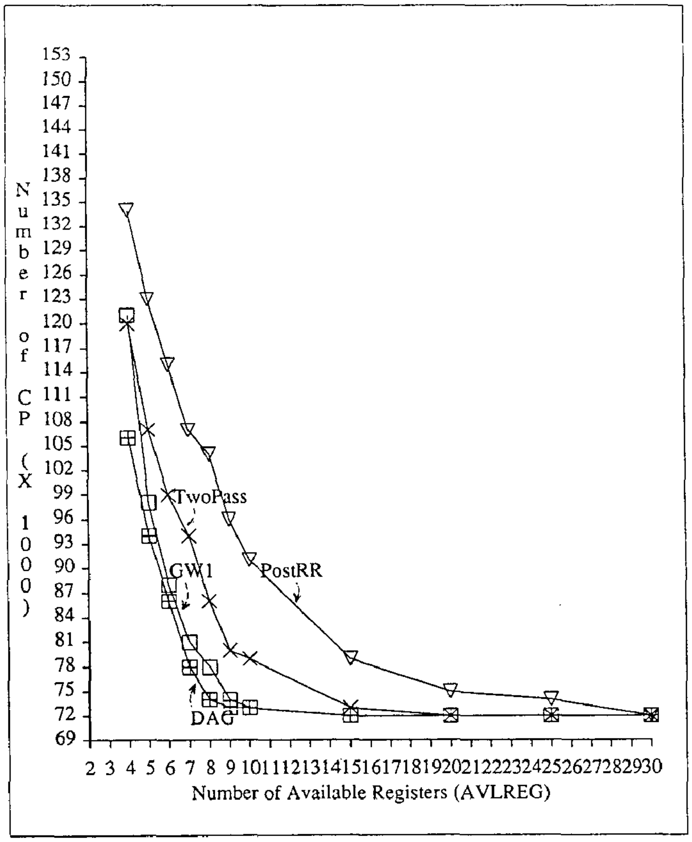


Figure 4.4 Performance of DAG-Driven Allocation (Medium Pipelined)

5. Conclusion

The problems of register allocation and instruction scheduling are often treated independently, although they are closely related. We attempt to look at the two problems at the same time by investigating two approaches: register allocation driven code scheduling and dependence graph driven register allocation. Both the integrated prepass scheduling and the DAG-driven register allocation approaches have been shown to be effective in solving the problem of the interdependency between code scheduling and register allocation.

Conceptually, the DAG-driven register allocation approach is simpler. It attempts to minimize the storage-related dependencies by carefully allocating a limited number of registers available to a basic block. The integrated prepass scheduling is more flexible and more aggresive than the DAG-driven allocation. When long interlocks encounted during code scheduling, our integrated prepass scheduling will force spilling. This is useful in a case where many values in the outer loop are carried through registers, leaving insufficient registers for the inner loop to have a decent scheduling. Our prepass scheduler will force the inner loop to spill (spill code will be moved to the outer loop), creating sufficient registers for code scheduling.

REFERENCES

[Aho77] Aho, A. V., S. C. Johnson, and J. D. Ullman, "Code Generation for expressions with common subexpressions," *JACM*, 24:1, 146-160, 1977

[Aho86] Aho, A. V., J. D. Ullman, and R. Sethi, "Compilers Principles, Techniques, and Tools," Addison-Wesley, Reading, MA, 1986.

[Ausl82] Auslander, M. and M. Hopkins, "An Overview of the PL.8 compiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June, 1982.

[Cray82] Cray Research Inc., *Cray-1 Computer System S* Series Mainframe Reference Manual (HR-0029), 1982.

[Davi86] Davidson, J. W., "A Retargetable Instruction Reorganizer," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction.* June, 1986

[Dong79] Dongarra, J. J. and A. R. Jinds, "Unrolling Loops in Fortran," *Software Practice and Experience* 9, 3, pp. 219-226, Mar., 1979

[Elli85] Ellis, J. R., "Bulldog: A Compiler for VLIW Architectures," *Ph.D. Thesis*, YaleU/DCS/RR-364, Yale University, Feb., 1985

[Fish81] Fisher, J., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981.

[Gibb86] Gibbons P. B., and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Jun., 1986

[Henn83] Hennessy, J. L., and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Transactions on Programming Languages and Systems* 5, 3, pp. 422-448, July 1983

[Henn84] Hennessy, J. L., "VLSI Processor Architecture," *IEEE Transactions on Computers*, Vol. c-33 No. 12, Dec., 1984.

[Hsu87] Hsu, Wei-Chung, "Register Allocation and Code Scheduling for Load/Store Architectures" *UW Computer Science Technique Report #722*, Oct., 1987

[Kogg81] Kogge, P. M., "The Architecture of Pipelined Computers," McGraw-Hill, New York, 1981

[McMa72] McMahon, F. H, "FORTRAN CPU Performance Analysis", Lawrence Livermore Laboratories, 1972

[Milu86] Milutinovic, Veliko, "GaAs Microprocessor Technology" *Computer*, Vol. 19, No. 10, Oct. 1986

[Padu86] Padua, D. and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communication of the ACM*, Dec. 1986

[Seth70] Sethi, R. and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *JACM* 17, 6, 1970, pp. 715-728

[Thor70] Thornton, J. E., "Design of a Computer, The Control Data 6600," Scott, Foresman and Co., Glenview, IL, 1970.

[Tjad70] Tjaden, G. S. and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers* 19(10):889-895, Oct., 1970

[Toma67] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development* V 11, pp. 25-33, Jan., 1967.

[Weis87] Weiss, S. and J. E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers" *Second International Conference on Architectural Support for Programming* Languages and Operating Systems," Oct., 1987.

[Youn85] Young, H., "Evaluation of a Decoupled Computer Architecture and the Design of A Vector Extension," *Computer Sciences Technical Report #603*, July, 1985