CHAPTER 9

Instruction Scheduling for Uniprocessors

9.1 Overview

In this chapter we present methods for scheduling instructions on single-processor machines that can issue several instructions on each cycle. Such machine architectures have been used in supercomputers in the past but they have now become common on high-performance multiprocessors that are used in today's workstations and parallel supercomputers. On these machines, the compiler is responsible for arranging the instructions of the object program in such a way as to take maximum advantage of the parallelism in the machine architecture—parallelism in instruction processing and function execution—without compromising the meaning of the program.

We will begin our treatment of this subject by defining an abstract machine model that will be used as the target for the strategies developed

in this chapter. The machine will have an arbitrary number of functional units and the capability of issuing one or more instructions of each functional type on each machine cycle. This machine model should be reasonably close to the practical machine designs that will be found in new computers for some time. We will then explore two general strategies: list scheduling for straight-line code and software pipelining for loops. Finally, we will discuss the issues that arise in dealing with control flow within a loop.

9.2 Machine Model

We assume a machine with a number of *issue units*, of various types. Each issue unit corresponds to a machine resource such as an integer or floating point arithmetic unit. Each issue unit will be able to issue one operation per cycle on the functional unit it controls, which we assume to be pipelined. Each issue unit will have an associated *type*, which specifies the kind of resource it controls, and a *delay*, which is the number of cycles required before the result of an operation, once issued, are available.

We will index issue units by two integers: the integer index denoting the unit type and the integer index of the particular unit within that type. The notation

$$\mathrm{I}_j^k$$

will be used to denote the jth unit of type k. We let m_k denote the number of type k. Thus the total number M of issue units in the machine is

$$M = \sum_{i=1}^{l} m_i \qquad \text{(EQ 9.1)}$$

where l is the number of issue-unit types available in the machine. Since each issue unit can issue one operation per cycle, the peak issue rate of the machine is M operations per cycle.

For the purpose of exposition, we will assume a wide-word instruction model, in which each wide instruction consists of M subinstructions, one for each issue unit. The job of the compiler is to select, for each cycle, a set of $\leq M$ operations to be included in the wide instruction, such that the number of operations of type k is $\leq m_k$. Assume that, if the number of

instructions of type k is strictly less than m_k, the operation for each unused instruction issue slot is a no-op.

In this model, a scheduling problem will be represented as a graph in which the vertices represent instructions of a given type and the edges represent dependences with a given delay equal to the delay for the type of operation at the source vertex of the edge.

9.3 Straight-Line Graph Scheduling

We begin with the simplest of scheduling problems, namely that of scheduling straight-line code. We assume that a preliminary phase of the compiler constructs a scheduling graph

$$G = (N, E, type, delay) \quad \text{(EQ 9.2)}$$

where N is the set of instructions in the code, where each $n \in N$ has a type given by $type(n)$ and a delay given by $delay(n)$. A correct schedule is a mapping S from vertices in the graph to nonnegative integers, representing cycle number such that:

1. $S(n) \geq 0$ for all $n \in N$,
2. If $(n_1,n_2) \in E$,, $S(n_1) + delay(n_1) \leq S(n_2)$, and
3. For any type t, no more than m_t vertices of type t are mapped to a given integer.

We will define the *length* of the schedule S, denoted $L(S)$, as follows

$$L(S) = \max_{n \in N} (S(n) + delay(n)) \quad \text{(EQ 9.3)}$$

The goal of straight-line scheduling is to find the shortest possible correct schedule, where a straight-line schedule S is said to be *optimal* if

$$L(S) \leq L(S_1)$$

for any other correct schedule S_1 for the same graph.

9.3.1 List Scheduling

The usual way to schedule a straight-line graph is by the use of *list scheduling*, a technique by which instructions that are ready to schedule are taken off of a worklist of ready instructions. A naive list scheduling algorithm is given in Figure 9.1.

FIGURE 9.1 Naive list scheduling algorithm

```
/* W[MaxC] is an array of worklists to hold ready instructions */
/* MaxC is 1 greater than the largest delay of any instruction  */

for each n ∈ N do count(n) := 0; earliest(n) = 0 od;
for each (n1,n2) ∈ E do
      count(n2) := count(n2) + 1;
      successors(n1) := successors(n1) ∪ {n2};
od;
for i := 0 to MaxC – 1 do W[i] := ∅;
Wcount := 0;
for each n ∈ N do
      if count(n) = 0 then
          W[0] := W[0] ∪ {n};
          Wcount := Wcount + 1
      fi
od;
c := 0     /* c is the cycle number */
cW := 0  /* cW will be the number of the worklist for cycle c */
while Wcount > 0 do
      while W[cW] = ∅ do
          instr[c] := ∅;
          c := c + 1; cW := mod(cW+1,MaxC)
      od;
      nextc := mod(nextc+1,MaxC);
      while W[cW] ≠ ∅ do
          select and remove an arbitrary instruction x from W[cW];
          if there are free issue units of type(x) on cycle c then
              instr[c] := instr[c] ∪ {x}; /* schedule the instruction */
              for each y ∈ successors(x) do
                  count(y) := count(y) – 1;
                  earliest(y) := max(earliest(y), c +delay(x));
                  if count(y) = 0 then
                      loc := mod(earliest(y),MaxC);
                      W[loc] := W[loc] ∪ {x}
                  fi
              od
          else
              W[nextc] := W[nextc] ∪ {x}
          fi
od od
```

The basic idea behind this algorithm is to schedule an instruction at the first opportunity after all operations on which it depends have completed. The *count* array is used to determine when the final predecessor has been scheduled and the *earliest* array is used to determine the earliest possible scheduling time for a given instruction. When an instruction's last predecessor has been scheduled, its *earliest* value is just late enough to insure that all inputs to the instruction will be available. This value is then used to determine which of the worklists will hold the instruction. We only need enough worklists to ensure that no instruction will be scheduled ahead of some instruction it depends upon. Choosing the number of worklists to be greater than the largest delay accomplishes this. Thus, if the earliest cycle on which an instruction could be scheduled is c, we will put it on worklist $W[\mathrm{mod}(c,MaxC)]$.

One problem with the algorithm in Figure 9.1 is that it selects randomly from the worklist for a particular cycle. Thus if there are not enough resources of a particular type to schedule all the instructions on the worklist for a particular cycle, the remaining instructions after all these resources are used will be put on the worklist for the next cycle. This might have the effect of delaying instructions on the critical path in favor of less urgent instructions. The problem lies in the undisciplined way in which instructions are selected from the current worklist. We can improve the algorithm by making a preliminary pass through the graph, determining minimum number of cycles that remain after each instruction. This can be done via a backward sweep through the graph as shown in Figure 9.2

FIGURE 9.2

Prepass to determine critical instructions.

```
for each n ∈ N do count(n) := 0; remaining(n) = delay(n) od;
for each (n1,n2) ∈ E do
        count(n1) := count(n1) + 1;
        predecessors(n1) := predecessors(n1) ∪ {n2};
od;
W := ∅;
for each n ∈ N do
        if count(n) = 0 then
            W := W ∪ {n}
        fi
od;
while W ≠ ∅ do
        select and remove an arbitrary instruction x from W;
```

```
        r := remaining(x);
        for each y ∈ predecessors(x) do
            count(y) := count(y) – 1;
            remaining (y) := max(remaining(y), r +delay(y));
            if count(y) = 0 then
                W := W ∪ {y}
            fi
        od
od
```

Once this computation is done, we can use it in the scheduling algorithm, we always select the instruction with the highest *remaining* value when taking from the worklist. This requires that we organize the worklist as a priority queue, which may mean a loss of a logarithmic factor in complexity.

9.3.2 Trace Scheduling

List Scheduling methods can be extended to more general non-looping data structures—control flow trees and directed acyclic graphs—by techniques referred to as "trace scheduling." The basic idea of trace scheduling is that one "trace" of control flow is scheduled at a time, where a *trace* is defined as a specific collection of basic blocks which form a single pass through all or part of a program.

To apply trace scheduling to a directed acyclic graph, we must first subdivide the dag into a collection of traces. This is done by first finding the most probably path through the dag and making that a trace. Next, the second most profitable path is chosen, where this path might branch out of or into the trace already chosen. The process concludes when every path has been accounted for.

Next the scheduling process is performed. First, the most probable path is scheduled. At each subsequent step, the most probable remaining trace is selected and scheduled. If the current trace branches into or out of a previously-scheduled trace, "fix-up code" is inserted at the beginning and end of the trace to finish any operations that remain from before the branch point and to finish any operations that need to be concluded before rejoining the previous trace.

(More details here)

Trace scheduling can be extended to handle loops by completely unrolling those loops into dags and using the methods above.

9.4 Software Pipelining

9.4.1 Kernel Scheduling

The approach used by trace scheduling to handle loops—complete unrolling—can lead to very large code sizes and long compile times. To avoid this, we can attempt to schedule the loop more directly producing three portions:

1. a *kernel*, which includes the code that must be executed on every cycle of the loop, once it has reached steady state;
2. a *prologue*, which includes the code that must be performed before steady state can be reached and
3. an *epilogue*, which contains the code that must be executed to finish the loop once the kernel can no longer be executed.

Note that the kernel will be executed at most the same number of times as the original loop.

Definition 9.1. The *kernel scheduling problem* seeks to find a minimal-length kernel for a given loop.

A question that comes to mind is whether minimizing the kernel is equivalent to optimally scheduling the loop. Of course, this is true for very large iteration counts for the whole loop, but for small numbers of iterations, the length of the prologue and epilogue will have a significant impact on the length of the overall schedule. Nevertheless, we will look for techniques that discover optimal kernel length as the single most important effect on the length of the overall schedule.

We now present a more formal definition.

Definition 9.2. A *kernel scheduling problem* is a graph

$$G = (N,E,\ delay,\ type,\ cross)$$

where $cross(n_1,n_2)$, defined for each edge in E is the number of iterations crossed by the dependence relating n_1 and n_2..

For a loop-independent dependent dependence, this figure will be 0. If a dependence carries into the next iteration, then its *cross* value will be 1, and so on.

Definition 9.3. A solution to the kernel scheduling problem is a pair of functions (S,I), where the *schedule* S maps each instruction n to a cycle within the kernel and the *iteration* I maps each instruction to an iteration offset from zero, such that:

$$S(n_1)+delay(n_1) \leq S(n_2)+(I(n_2)-I(n_2)+cross(n_1,n_2))L_k(S) \quad \text{(EQ 9.4)}$$

for each edge (n_1,n_2) in E, where $L_k(S)$ is the length of the kernel for S.

$$L_k(S) = \max_{n \in N} S(n) \quad \text{(EQ 9.5)}$$

In other words, each instruction is mapped to a cycle within the kernel and an offset from the base cycle. Consider a simple example. If the original source was

```
   ld   r1,0
   ld   r2,400
   fld  fr1, c
l0 fld  fr2,a(r1)
l1 fadd fr2,fr2,fr1
l2 fst  fr2,b(r1)
l3 ai   r1,r1,8
l4 comp r1,r2
l5 ble  l0
```

Assume that there are three units in the machine: a load/store unit, an integer unit and a floating-point unit. All loads have a delay of 2, while stores have a delay of 1. floating point addition has a delay of 3 cycles. The integer unit handles all branch instructions with a delay of 1 cycle on the branch taken side for "`blt`." All other integer instructions have a delay of 1. For the instructions in the loop, the following is a legal schedule

$S(\texttt{l0}) = 0$; $I(\texttt{l0}) = 0$;
$S(\texttt{l1}) = 2$; $I(\texttt{l1}) = 0$;
$S(\texttt{l2}) = 1$; $I(\texttt{l2}) = 1$;
$S(\texttt{l3}) = 0$; $I(\texttt{l2}) = 0$;
$S(\texttt{l4}) = 0$; $I(\texttt{l2}) = 0$;
$S(\texttt{l5}) = 0$; $I(\texttt{l2}) = 0$;

This assumes that we have enough intermediate storage to capture and hold results—the index value to be used in the store in this case—over a period of more than one iteration. As we shall see, this is not always easy. The kernel schedule is depicted graphically in Figure 9.3.

FIGURE 9.3 Example kernel schedule.

Unit	Load/Store	Integer	Floating Pt
Cycle 1	`fld a(r1)`	`ai r1,r1,4`	
Cycle 2		`comp r1,r2`	
Cycle 3	`fst b-8(r1)`	`ble`	`fadd fr2`

Notice that, on each iteration of the kernel, the floating point store in cycle 3 actually corresponds to the floating point load and add from the previous iteration of the kernel. This means that the program as originally stated above will not quite work because we need to save the result of the floating point addition in another register, say `fr3`, so that we may reuse `fr2` for the load in cycle 1 on the next iteration of the kernel. Thus, the program we are executing is the following:

```
   ld   r1,0
   ld   r2,400
   fld  fr1, c
p1 fld  fr2,a(r1); air1,r1,8
p2 comp r1,r2
p3 beq  e1; fadd fr3,fr2,fr1
k1 fld  fr2,a(r1); air1,r1,8
k2 comp r1,r2
k3 fst  fr3,b-16(r1);blek1; fadd fr3,fr2,fr1
e1 nop
e2 nop
e3 fst  fr2,b-8(r1);bltl0; fadd fr2,fr2, fr1
```

Instructions `p1`–`p3` form the prologue, `k1`–`k3` the kernel, and `e1`–`e3` the epilogue.

The kernel in this example is of length 3 and given the requirement for three integer operations on each iteration of the kernel, this is the smallest possible.

Definition 9.4. Let N be the loop upper bound. Then the *schedule length* $L(S)$ is given by

$$L(S) = N\,L_k(S) + \max_{n \in N}(S(n) + delay(n) + (I(n)-1)L_k(S)) \quad \text{(EQ 9.6)}$$

This definition leads us to the observations that for very large N, minimizing the length of the kernel minimizes the length of the schedule. Since N will usually be unknown at compile time, we will assume that it is large and try to construct the shortest possible kernel.

9.4.2 A Kernel Scheduling Algorithm

It is possible to compute a lower bound on the kernel length by dividing the problem into two cases. First, assume that there is no recurrence in the loop.

Theorem 9.1. Let $\#t$ denote the number of instructions in each iteration that must issue in a unit of type t. Then,

$$L_k(S) \geq \max_{\boldsymbol{t}} \left\lceil \frac{\#t}{m_t} \right\rceil \quad \text{(EQ 9.7)}$$

This lower bound is often referred to as the *finite hardware constraint.*

Theorem 9.2. For a loop with no recurrence, there exists a schedule S such that

$$L_k(S) = \max_{\boldsymbol{t}} \left\lceil \frac{\#t}{m_t} \right\rceil. \quad \text{(EQ 9.8)}$$

In other words, we can always find a kernel schedule of optimal length.

Proof. Use a list scheduler and a resources scoreboard. As each instruction is assigned an issue slot, mark off the corresponding resource on the scoreboard. When scheduling a given node n, put it in the first available issue slot after the delays of all instructions on which it depends have expired, noting that when the end of the kernel is passed, you wrap to the beginning, increasing the value for $I(n)$. Since we always have enough resources, the kernel is guaranteed to fit in the given length. Q.E.D.

Of course, a major problem with this theorem is that it ignores the question of whether we have enough machine registers to hold all the values that need to be propagated around the kernel. We will discuss this issue further in Section 9.5.

In the case of recurrences, we can also determine a minimum length for a cycle.

Theorem 9.3. Given a cycle of dependences $(n_1, n_2,...,n_k)$ then

$$L_k(S) \geq \frac{\sum_{i=1}^{k} delay(n_i)}{\sum_{i=1}^{k} cross(n_i, n_{i+1})} \quad \text{(EQ 9.9)}$$

Hence,

$$L_k(S) \geq \max_{c}\left(\frac{\sum_{i=1}^{k} delay(n_i)}{\sum_{i=1}^{k} cross(n_i, n_{i+1})}\right) \quad \text{(EQ 9.10)}$$

where c ranges over all dependence cycles in the loop.

Proof. The quantity in the numerator of Equation 9.9 is the number of machine cycles required to execute a single traversal of the recurrence, while the denominator counts the number of loop iterations crossed by the recurrence. Thus the right hand side of Equation 9.9 measures the number of cycles per iteration required to satisfy the kernel. Clearly, no correct kernel can have fewer cycles per iteration and compute the recurrence. Q.E.D.

Definition 9.5. The quantity of the right hand side of Equation 9.9 is called the *slope* of the recurrence.

We will now turn to a kernel scheduling algorithm for loops with recurrences.

FIGURE 9.4 Algorithm for kernel scheduling.

use the all-pairs shortest path algorithm to find the cycle in the schedule

graph with the smallest slope;

Compute the lower bound *LB* for the loop as the maximum of the slope of this recurrence as given by the right hand side of Equation 9.10 and the hardware constraint as given in Equation 9.8.

length = *LB*;
failed = *true;*
while *failed* **do**
    try to schedule the loop to length *length* setting failed to *false* if the step fails, *true* otherwise;
    if *failed* **then**
        length := *length* + 1
    else
        report the schedule and *length*
    fi;
od;

sched = the selected schedule;

fill in all the non-recurrence parts using list scheduling;

9.4.3 Code Generation

We define the *range* of an iteration of the schedule as follows:

$$range(S) = \max_{n \in N}(I(n)) + 1 \qquad \textbf{(EQ 9.11)}$$

Thus defined, the *range* of a schedule is the number of iterations required for all the instructions corresponding to a single iteration of the original loop to issue.

We can use the notion of range to examine the sizes that will be required for the prologue and epilogue of a loop. To get a full kernel, with actual copies of each of the instructions in steady state, we will need to have instructions from *range*(*S*) different iterations, including the one currently starting. Thus if $r = range(S)$, the first complete iteration to execute is the rth. Therefore, the length of the prologue must be

$$(r-1)L_k(S)$$

To generate the prologue, simply lay out $r - 1$ copies of the kernel, with any instruction n such that $I(n) = i < r - 1$, replaced by a no-op in copies 1 through i. Note that the prologue may be compressed substantially by

rescheduling using a list scheduling algorithm, so the length estimate above is an upper bound.

We can use a similar method to get a bound on the length of the epilogue. Once the last full iteration of the kernel is complete (this is the iteration on which instructions on the last iteration of the original code begin to issue) it can be completed in $r - 1$ iterations plus the time to finish any instructions that do not finish in that last iteration. We estimate the additional time needed in the following equation:

$$\Delta(S) = (\max_{n \in N} ((I(n)-1)L_k(S)+S(n)+delay(n)) - rL_k(S))^+ \quad \textbf{(EQ 9.12)}$$

where the superscript "+" denotes the *positive part* of the expression defined in Chapter 3 to be equal to the quantity to which it applies if that quantity is greater than zero and zero otherwise.

Thus the length of the epilogue has the following upper bound:

$$(r - 1)L_k(S) + \Delta(S)$$

As in the case of the prologue, the length of the epilogue can be reduced by list scheduling.

Thus, to generate code for the epilogue, simply lay out $r - 1$ copies of the kernel, replacing any instruction n such that $I(n) = i \prec r - 1$ with a no-op in copies i+1 to $r - 1$.

9.5 Register Resources

Software pipelining as described so far does not take register resources into account. If a value produced by one instruction is the input to another instruction a number of cycles later, it will require a register to be available for this purpose. If there are not enough registers, loads and stores will need to be inserted, or the schedule adjustes. This is a difficult problem that has not been solved in a completely satisfactory manner. A number of heuristic techniques have been advanced, however. For example, it may be possible to schedule a loop and then, if register resouces are not enough, spill values into memory and reschedule.

Even when enough registers exist, naming clashes can be a problem. Such clashes are caused by trying to use the same register to hold two different values on different original iterations whose live ranges overlap.

However, as we saw in Chapter 8, naming problems of this sort can often be resolved by loop unrolling.

9.6 Loops with Control Flow

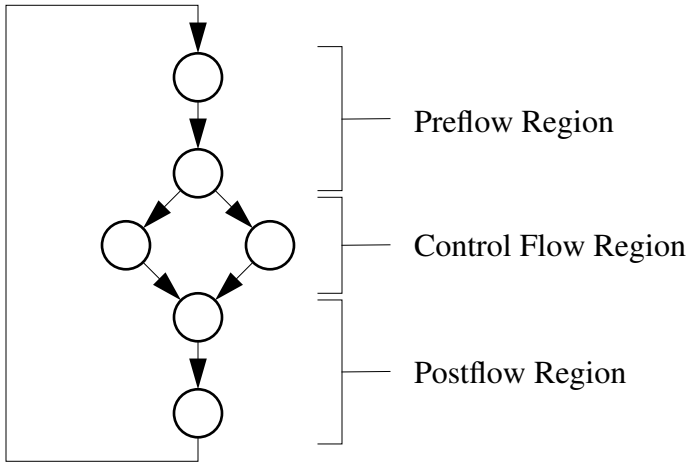
Up to now we have considered only loops that have no control flow. As we saw earlier, control flow presents more than one path to be scheduled, which complicates software pipelining in two ways:

1. Because there are different control flow paths of different schedule lengths, it is no longer possible to assume that the same number of cycles will be required on every iteration, as we do in the case of no control flow.
2. All instructions under a control flow branch must issue before the join of the two parts of a control flow branch, otherwise we will not know which instructions to remain to be issued without having saved the result of the control flow decision in a register.

Thus it is difficult to merge control flow regions with the instructions outside the control flow region to get maximum overlap. Of course, it is always possible use trace scheduling on the loop body, which amounts to betting which of two alternative control flow paths is more likely to be executed.

One approach to overcoming this problem is to schedule the control flow region first, then treat the region like a black box. That is, assume that it is a single macro-instruction, requiring all of the resources for some number of cycles. Once it has been scheduled, it can be treated as a single instruction for the purpose of scheduling later instructions. Then the instructions outside the control flow region can be scheduled before or after the region using software pipelining. This strategy is illustrated in Figure 9.5.

FIGURE 9.5 Software pipelining in the presence of control flow.



Here, each of the paths in the control flow region is scheduled independently, then the entire control flow region is reduced to a superinstruction, taking all resources for a number of cycles equal to the length of the longest path. Once this is completed, the preflow region, superinstruction and postflow region are scheduled using standard software pipelining.

Note that this strategy can be improved by viewing the superinstruction as issuing in a single cycle but requiring delays of varying lengths for use of its outputs. When combined with a pattern of resource requirements for the duration of the superinstruction, this might permit some instructions in the postflow region to be scheduled into all paths of the control flow region, shortening the length of the kernel.

A similar strategy can be used to shorten the preflow region. A dependence with delay d between an instruction i_1 in the preflow region and an instruction i_2 in the control flow region may not require that i_1 issue d cycles before the superinstruction. If i_2 has been scheduled on the kth cycle of the superinstruction, then i_1 can be issued as few as $d–k+1$ cycles before the superinstruction. If $d–k+1$ is negative, i_1 can be overlapped with the superinstruction if empty resources are available on both paths.

Multiple layers of control flow can be dealt with in the same manner by working from inside to outside.

9.7 Historical Comments and References

9.8 Bibliography