6

MODULO SCHEDULING

Loop parallelization, and particularly the parallelization of innermost loops, is the most critical aspect of any parallelizing compiler. Trace scheduling can be applied to loops, but has the disadvantage that loops must be unrolled to exploit parallelism between loop iterations, which can lead to code explosion and in general still does not exploit all of the parallelism available. In this chapter we discuss modulo scheduling, which was the first technique to address the scheduling of loops (both within and across iterations) directly and is still very widely used in practice. The original modulo scheduling technique applies only to loops where the loop body is a single basic block; we also present extensions to modulo scheduling that allow the technique to be applied to more general loops.

6.1 Introduction

Loop parallelization is the most critical aspect of any parallelizing compiler, since most of the time spent in executing a given program is spent in executing loops. In particular, the innermost loops of a program (i.e., the loops most deeply nested in the structure of the program) are typically the most heavily executed. It is therefore critical for any parallelizing compiler to try to expose and exploit as much parallelism as possible from these loops.

The simplest approach to extracting ILP from loops is to schedule the loop body. This method can find parallelism within a single loop iteration, but cannot exploit parallelism that may exist between different iterations of a loop. Furthermore, some loop bodies are just

A. Aiken et al., *Instruction Level Parallelism*,
DOI 10.1007/978-1-4899-7797-7_6

short; any instruction scheduling technique will struggle to extract parallelism if the loop body contains only a handful of instructions.

Historically, *loop unrolling*, a standard non-local transformation, is used to expose parallelism beyond iteration boundaries and expose a large number of instructions for scheduling. When a loop is unrolled, the loop body is replicated to create a new loop. Compaction of the unrolled loop body helps exploit parallelism across iterations as the operations in the unrolled loop body come from previously separate iterations. However, usually loops cannot be fully unrolled, either because the loop bounds are unknown at compile time or, very frequently, because high degrees of unrolling result in serious compiler performance problems due to space consumption. Of course, less unrolling also limits the amount of instruction-level parallelism exposed.

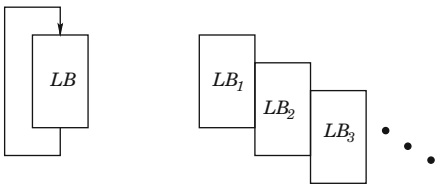


Figure 6.1 : Loop pipelining

In order to extract higher levels of parallelism in software without unrolling, the various iterations of a loop can be pipelined (akin to pipelining of operations in hardware [PD76, Kog77b, Kog81]) subject to dependences and resource constraints. This technique is known as *loop pipelining*. An example of loop pipelining is shown in Figure 6.1. Figure 6.1(b) shows the pipelined execution of the different iterations of the loop shown in Figure 6.1(a), where LB_i represents the loop body of the i-th iteration. Loop pipelining has its roots in the hand-coding practices for microcode compaction [TTTY78, Cha81, Tou84].

Like loop pipelining, *Modulo scheduling* [RG81, Hsu86] initiates successive iterations of a loop at a regular, fixed interval called the *initiation interval* or *II*. Unlike loop pipelining, modulo scheduling also schedules the loop body, including allowing the schedule for the loop body to be made *longer* by introducing holes in the schedule if that results in a smaller initiation interval. The combination of initiation interval and selected schedule for the loop body must respect the

dependences of the original loop and the resource constraints of the target machine.

Modulo Scheduling was the first method proposed for *software pipelining* (the term software pipelining, originally coined to describe a specific transformation [Lam88], is nowadays commonly used interchangeably with loop pipelining to describe any and all transformations that deal with extracting instruction-level parallelism from loops). The objective of software pipelining is to find a recurring pattern, or *kernel*, across iterations, and to replace the original loop with this kernel, plus prologue and epilogue code. The problem of loop pipelining then reduces to how to determine the (best) parallelized kernel for a loop.

6.2 Unrolling

Before discussing modulo scheduling we first discuss loop unrolling, which is useful both to motivate the benefits of modulo scheduling but is also an important loop transformation that we will use in the subsequent development. As mentioned above, the earliest technique for extracting instruction level parallelism from within and across multiple iterations was *unrolling* or *unwinding* of the loop. If the number of iterations in the loop is a compile time constant, the loop can be *fully unrolled* by simply repeating the loop body a number of times equal to the number of iterations of the loop. For a `for`-style loop, we of course also need at least to insert the update of the iteration variable between the copied iterations, but in practice it is more common to modify the copies of the loop body to replace references to the loop index with an expression for the loop index's value in each loop iteration. For example, if the loop index is a variable i that ranges from 1 to 10 over 10 iterations of the loop, then occurrences of i in the first unrolled iteration would would be replaced by 1, occurrences of i in the second unrolled iteration would be replaced by 2, and so on. In either case the loop itself (i.e., the exit tests and jumps associated with the loop structure) is eliminated, which reduces the number of instructions executed at run-time and therefore potentially reduces the execution time even if no parallelism is extracted. Because of this property, full unrolling is sometimes used as an optimization even in architectures exhibiting no parallelism. An example of full unrolling is given in Figure 6.2.

```
do i = j, j+3                a[j] = 0.0
    a[i] = 0.0               a[j+1] = 0.0
end do                       a[j+2] = 0.0
                             a[j+3] = 0.0
```

Figure 6.2 : Full unrolling: before and after unrolling

Note that the loop bounds themselves need not be known at compile time for full unrolling to be feasible: all that is required is that the number of iterations be constant and known at compile time. Also note that this implies that `repeat` or `while`-style loops generally cannot be fully unrolled.

Of course, full unrolling is not always possible, since the number of iterations is usually not a compile-time constant. Furthermore,

even in cases where the number of iterations is a compile-time constant, unrolling is impractical if the number of iterations is large. Even if space per se is not a concern and thus the code explosion resulting from full unrolling of loops could be tolerated (at least for innermost loops) such unrolling may actually be undesirable from a performance point of view. Indeed, the unrolled loop may adversely affect instruction cache performance, possibly degrading overall performance despite the reduction in the number of instructions executed.

For these reasons, full unrolling is usually not done. However, a partial unrolling is often useful, and in fact partial unrolling can always be performed in both `for`-style loops (where the termination condition is independent of what is computed in the loop body, such as in Figure 6.2) and in `while`-style loops (where the termination condition depends on what is computed on each iteration, for example checking for a string or list terminator). Partial unrolling can take two forms. The simplest and most general approach consists of unrolling the loop by replicating the loop body, including the exit test, a fixed number of times (called the *unrolling factor*), replacing the original loop body with this unrolled loop body. If the loop has a step, the loop step is also changed by multiplying the original step by the unrolling factor. An example is given in Figure 6.3.

```
                        do i = 1, N, 2
                            a[i] = 0.0
                        end do

    After unrolling:
                        do i = 1, N, 6
                            a[i] = 0.0
                            if i >= N exit
                            a[i+2] = 0.0
                            if i + 2 >= N exit
                            a[i+4] = 0.0
                        end do
```

Figure 6.3 : Unrolling a simple loop three times

If the compiler can determine that the number of iterations executed is a multiple of the unrolling factor,[1] we can perform a further optimization by removing the intermediate exit tests in the unrolled loop body [Ell85, CNO+88]. Even if the number of loop iterations is statically unknown, the transformation can still be done if the number of loop iterations is known at runtime when the loop begins executing. The original loop is replaced by two loops. The first loop (possibly unrolled with exit tests left in) executes just enough iterations so that the remaining number of iterations is evenly divisible by the desired unrolling factor. The second loop is then unrolled using the chosen factor and the intermediate exit tests are removed. For loops with a sufficiently large number of iterations, this transformation potentially results in substantial performance benefits. An example is given in Figure 6.4. Due to the overhead involved this method is not used as often as one might expect. Also note that this method is not applicable to `while`-style loops where the number of loop iterations is unknown even during execution of the loop (i.e., it is known only when executing the final test that determines the loop terminates, not prior to executing the loop).

Once the loop has been unrolled, any of the scheduling techniques discussed in previous chapters (e.g., list scheduling, trace scheduling or percolation scheduling) can be applied to extract ILP both within

[1] In practice this means that we set the unrolling factor such that it is an even divisor of the number of iterations of the loop, if known.

```
                    do i = 1, N
                      a[i] = 0.0
                    end do

After test removal:
                    do j = 1, N mod 3
                      a[j] = 0.0
                    end do
                    do i = j, N, 3
                      /* Assume j is available (after increment)
                         after first loop completes */
                        a[i] = 0.0
                        a[i+1] = 0.0
                        a[i+2] = 0.0
                    end do
```

Figure 6.4 : Test removal when number of iterations is only known at runtime

and across iterations of the original loop. A limitation of this approach is that parallelism cannot be extracted across iterations of the unrolled loop. For example, say a loop L is unrolled three times to create a loop L^3 (e.g., as in Figure 6.4). The scheduling techniques we have discussed so far can parallelize the loop body of L^3, effectively overlapping iterations 1–3 and 4–6 of L, but no parallelism can be exploited between, say, iterations 3 and 4 of L because they are in different iterations of L^3. In the rest of this chapter and subsequent chapters we discuss techniques that can extract parallelism across all iterations of a loop.

6.3 Preliminaries

A data dependence between operations from the same iteration is a *intra-iteration dependence*; a data dependence between operations from different iterations is a *inter-iteration* or *loop carried* dependence. For example, in Figure 6.5, the dependence between operation v_2 and operation v_1 is an intra-iteration dependence; whereas the dependence between v_2 and v_3, shown by a dashed arrow in the data dependence graph (DDG) shown in Figure 6.5 b), is a loop carried dependence. The dependence *distance* (in number of iterations) between v_2 and v_3 is shown in angled brackets.

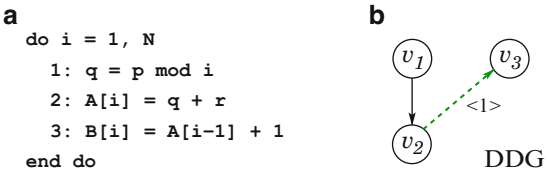


Figure 6.5 : Intra-iteration and loop carried dependences

Definition 6.1. *Given a dependence graph $G(V, E)$, a path from an operation v_1 to an operation v_k is a sequence of operations $\langle v_1, \ldots, v_k \rangle$, such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$.*

Definition 6.2. *A* cycle *in a dependence graph is a path $\langle v_1, \ldots, v_k \rangle$ such that $v_1 = v_k$ and the path contains at least one edge. A cycle is* simple *if, in addition, $v_1, \ldots, v_{k-1}$ are distinct.*

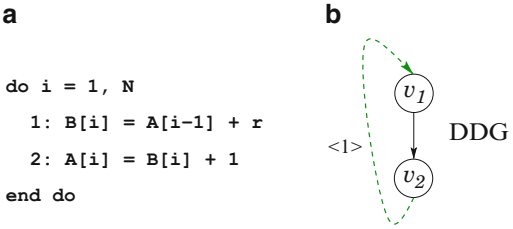


Figure 6.6 : Loop recurrences

A loop carried dependence in conjunction with intra-iteration dependences may form a simple cycle in the dependence graph. For example, in Figure 6.6, the intra-iteration dependence and the loop carried dependence between the operations v_1 and v_2 form a simple cycle. A loop has a *recurrence* if an operation in one iteration of the loop has a direct or indirect dependence upon the same operation from a previous iteration, e.g., in Figure 6.6 operations v_i^k and v_i^{k-1} constitute a recurrence, where v_i^k represents the i-th operation of the k-th iteration. In general, a recurrence may span several iterations, i.e., an operation v_i^k may depend on an operation v_i^{k-j}, where $j \geq 1$. The existence of a recurrence manifests itself as a simple cycle in the dependence graph. Subsequently, a cycle refers to a simple cycle in a dependence graph.

The *length* of a cycle c, denoted by $len(c)$, is the sum of the latencies of operations in c. The *delay* of a cycle c, denoted by $del(c)$, is the

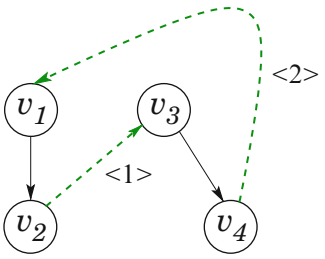


Figure 6.7 : A simple cycle

time interval between the start of operation v_1 and the start of operation v_{k-1} in a given schedule plus one, where operations $v_1, v_{k-1} \in c$, see Definition 6.2. Let $dist(c)$ denote the sum of the distances (corresponding to all the loop carried dependences) along c. For example, in Figure 6.7 the path $\langle v_1, v_2, v_3, v_4, v_1 \rangle$ constitutes a simple cycle. The length of the cycle is 4 and $dist(c) = 1 + 2 = 3$.

Note that the delay of a cycle c may not be equal to its length as the delay of a cycle is dependent on the actual scheduling of operations constituting the cycle. In general, for any cycle c in a dependence graph, $len(c) \leq del(c)$.

Finally, consider two operations o_1 and o_2 where o_2 comes after o_1 in the program and o_2 also depends on o_1. Recall that there are three distinct kinds of dependence that could exist between o_1 and o_2 (Definition 3.1): A true (or data) dependence when o_2 reads a location that o_1 writes, an anti-dependence when o_2 writes a location that o_1 reads, and an an output dependence when o_1 and o_2 write the same memory location. Anti- and output dependences can set up recurrences in the data dependence graph which potentially degrade the ability to extract ILP. To avoid this problem, the *dynamic single assignment form* [Fea88, Fea91, Rau92] may be used to minimize the anti- and output dependences. We will focus on true dependences in our examples, but the techniques we present apply to all dependences, regardless of type.

6.4 Modulo Scheduling Algorithm

Modulo scheduling takes as input a loop with a single basic block for the loop body and searches for a combination of an initiation interval and a schedule for the loop body that satisfies both the machine's

resource constraints and the data dependences of the loop. The essential idea is to fix an initiation interval i and then apply a basic block scheduling algorithm (e.g., list scheduling) to the loop body, with two modifications. First, when the initiation interval is i and operation o is scheduled, that means that an instance of operation o will be issued every i control steps. Thus, we record the fact that the resources required by o are in use every ith step. Second, the basic block scheduler only considers the intra-iteration dependences. When a schedule for the loop body is found we must still verify that the loop carried dependences are satisfied. If modulo scheduling encounters a situation where an operation cannot be scheduled due to resource constraints or a schedule for the loop body does not satisfy loop carried dependences, the initiation interval is increased by 1 and the entire process is repeated. The algorithm is guaranteed to terminate, because when the initiation interval reaches the length of the original loop body, modulo scheduling degenerates to scheduling only the loop body and not overlapping iterations, which will always succeed.

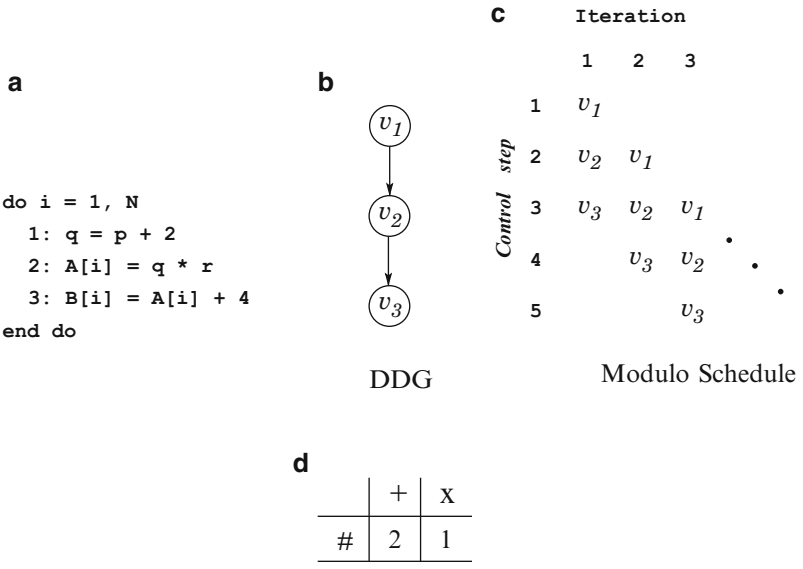


Figure 6.8 : An overview of modulo scheduling

An illustration of modulo scheduling is shown in Figure 6.8. Consider the loop and its data dependence graph shown in Figures 6.8(a) and 6.8(b) respectively. For the resources given in Figure 6.8(d) and a single cycle latency for each resource, a modulo schedule is shown in

Figure 6.8(c). The reader may find it instructive to verify the schedule with respect to the dependences and resource usage.

Let $\mathcal{R}$ be the resources available for scheduling loop L. Resources are grouped into classes, each of which consists of identical resources. Let $\mathcal{R}(i)$ denote the i-th resource class, and let the number of resources in the i-th class be denoted by $num(i)$. For example, in Figure 6.8(d), $\mathcal{R}(1)$ represents the set of adders, where $num(1) = 2$. Similarly, $\mathcal{R}(2)$ represents the set of multipliers, where $num(2) = 1$. For simplicity of exposition, each resource is assumed to have single cycle latency, but the techniques discussed here extend easily to deal with multi-cycle resources [Rau95].

Let V denote the set of operations in L's loop body. The function $r : V \rightarrow \{1, 2, \ldots\}$ gives the number of the resource class $r(v)$ used by each $v \in V$. (For simplicity, we assume each operation requires only one resource.) For example, in Figure 6.8(d), $r(v_1) = 1$ and $r(v_2) = 2$.

Algorithm 6.1. Modulo Scheduling
The input to this algorithm is a dependence graph $G(V, E)$ of the body a loop L. The output is an initiation interval ii and a modulo schedule assigning a control step $l(v)$ to each $v \in V$. An iteration is initiated every ii cycles and executes the schedule for the loop body given by the $l(v)$. Pseudo-code for modulo scheduling is given in Figure 6.9.

The algorithm given in Figure 6.9 can be improved in a number of ways. It is rarely necessary to start with an initiation interval of 1, because it is usually possible to quickly compute a better lower bound on the smallest possible initiation interval. Modulo scheduling implementations first look at the loop body and compute the *resource-constrained initiation interval (ResII)*, a lower bound on the initiation interval due to resource constraints:

$$\text{ResII} = \left\lceil \max_i \frac{|\{v | v \in V \text{ and } r(v) = i\}|}{num(i)} \right\rceil \tag{6.1}$$

Next, the *recurrence-constrained initiation interval (RecII)*, a lower on the initiation interval based on the length of recurrences in the dependence graph, is computed:

$$\text{RecII} = \left\lceil max_{c \in C} \frac{len(c)}{dist(c)} \right\rceil \tag{6.2}$$

where C is the set of all simple cycles in the dependence graph.[2]

[2] The recurrences in the dependence graph can be enumerated using the algorithms in [Tie70, MD76]. RecII can also be computed in polynomial time via, for instance, the Bellman-Ford algorithm [Bel58, CLR90].

```
ii ← 0
A: while ii < |V| do
   ii ← ii − 1
   // U(i, j) is the usage of the i-th resource class in control step j
   U(i, j) ← 0 for all i and j
   ℓ(v) ← 1 for all v ∈ V
   V' ← V
   while V' ≠ ∅ do
      // S is the set of unscheduled operations whose predecessors have been scheduled
      S ← {v v ∈ V' and Pred(v) ∩ V' = ∅}
      choose v ∈ S in
         // try to find a control step l(v) where resources permit v to be scheduled
         for x in 1..ii do
            if U(r(v), l(v) mod ii) ≥ num(r(v)) then
               ℓ(v) ← ℓ(v) + 1
            endif
         endfor
         // if we have failed to schedule v in ii consecutive
         // steps, then it cannot be scheduled with ii as the
         // initiation interval. We increment ii and try again.
         if U(r(v), l(v) mod ii) ≥ num(r(v)) then
            continue A
         endif
         // Otherwise, we schedule v in cycle l(v)
         for each w ∈ Succ(v) do
            ℓ(w) ← max(ℓ(w), ℓ(v) + 1)
         endfor
         V' ← V' − {v}
         U(r(v), l(v) mod ii) ← U(r(v), l(v) mod ii) + 1
      endchoose
   endwhile
   // Check that all loop carried dependences are satisfied by the schedule
   for each loop carried dependence (v, v') ∈ E with distance d do
      if l(v) > l(v') − ii ∗ d then
         continue A // go back to outer loop, increment ii, and try again
      endif
   endfor
   return ii and l(v) for each v ∈ V // a feasible schedule has been found
endwhile
```

Figure 6.9 : Modulo Scheduling Algorithm

The *minimum initiation interval (MII)* is the max of the resource- and recurrence-constrained initiation intervals.

$$MII = \max(ResII, RecII) \tag{6.3}$$

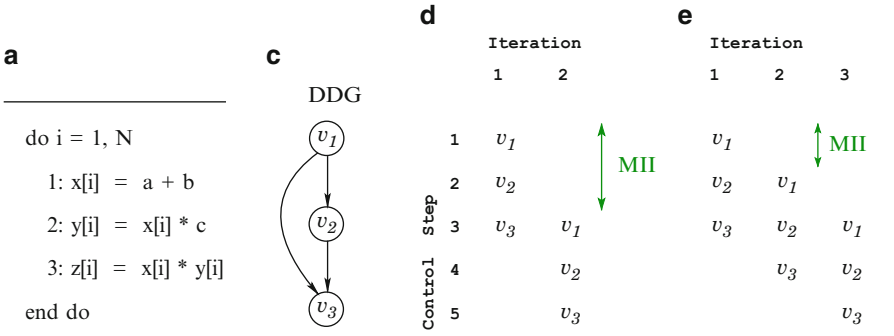
The minimum initiation interval is, again, a lower bound on the value of a feasible initiation interval: certainly there are no legal

schedules for smaller initiation intervals, but there is not necessarily a legal schedule for the MII, either. However, we can safely begin the search in Figure 6.9 with ii initialized to MII $-$ 1 (so that ii will have value MII on the first iteration of the outer loop).

The following example illustrates modulo scheduling in the absence of recurrences.

Example 6.1. Consider the loop shown in Figure 6.10(a) and its dependence graph as shown in Figure 6.10(c). From the dependence graph one observes that there are no loop carried dependences in the given loop. Therefore, in this case *MII* = *ResII*.

The number of available resources (adders, etc.) per control step is given in Figure 6.10(b). From Equation 6.1, ResII for the given resource constraints is 2. The modulo scheduled loop is shown in Figure 6.10(d), with MII as the initiation interval. In Figure 6.10(d), control steps 1 and 2 correspond to the *prologue* of the transformed loop; the prologue is executed once when the modulo scheduled loop is started. Control steps 3 and 4 constitute the *kernel* of the transformed loop; the kernel is the repeating pattern that is executed over and



	+	x
#	1	1

Figure 6.10 : Modulo scheduling in the absence of recurrences

over by the modulo scheduled loop, in this case executing operation v_3 from iteration i and operations v_1 and v_2 from iteration $i+1$. Control step 5 is the *epilogue* of the loop, which is executed just once to complete the final iteration before the loop terminates.

From the schedule one observes that 1 cycle is saved per iteration (except the first iteration) in the modulo schedule as compared to the sequential execution of the entire loop. Further overlap of iterations is not feasible as it would violate the resource constraints, e.g., in Figure 6.10(e), operation v_3 of the first iteration and operation v_2 of the second iteration cannot be scheduled in the same control step as there is only one multiplier available.

In Figure 6.9, the set S is the *ready set*, the set of operations where all dependence predecessors have been scheduled but the ready operations themselves have not yet been scheduled. Modulo scheduling involves selection of an operation amongst the operations in the ready set. Thus, like list scheduling, Algorithm 6.1 represents a family of algorithms parameterized by the method for selecting an operation from the ready set.

In the presence of *multi-function resources*, which can execute more than one type of operation, determining ResII becomes non-trivial. In such cases, a bestResII can be computed by performing an optimal bin-packing of the reservation tables for all the operations. The drawback of such an approach is its exponential complexity.

Next, we present an example of modulo scheduling in the presence of recurrences.

Example 6.2. Consider the loop shown in Figure 6.11(a) and its DDG as shown in Figure 6.11(b). From Figure 6.11(b) one observes that the dependence graph contains one cycle, $c = \langle v_2, v_5, v_6, v_7, v_2 \rangle$.

The length of the cycle ($len(c)$) is 4 and its distance ($dist(c)$) is 1. First, one computes the lower bound on the minimum initiation interval. Assuming the number of resources given in Figure 6.10(b), ResII for the DDG given in Figure 6.11(b) is 3 and RecII is 4. From Equation 6.3, one gets $MII = 4$. A corresponding schedule of the loop body is shown in Figure 6.11(c). From Figure 6.11(c), note that the delay of cycle c is 5 ($= del(c)$). Since $del(c) > len(c)$, the check that loop-

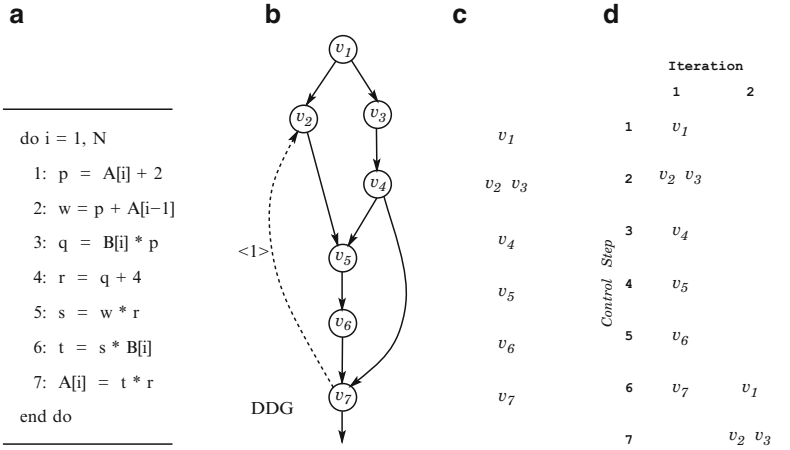


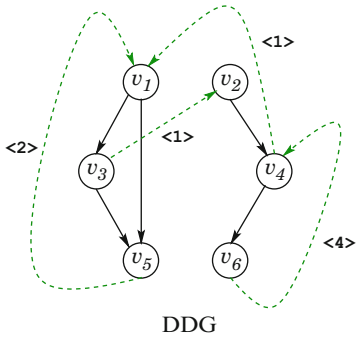
Figure 6.11 : Modulo scheduling in the presence of recurrences

carried dependences are preserved fails and so we determine that we have not been able to find a valid schedule with an initiation interval of 4. Therefore, the value of ii is incremented by one and the loop is rescheduled. The final modulo schedule is shown in Figure 6.11(d). As an exercise, the reader can verify that $ii = 5$ does not lead to any recurrence violation.

6.4.1 Remarks

Sufficiency of simple cycles

Algorithm 6.1 considers only simple cycles (defined in Section 6.3 on page 138) while computing RecII. However, it is not obvious that it is sufficient to consider simple cycles to guarantee a valid schedule. We illustrate the impact of both simple and non-simple cycles on RecII with the following example. Consider the dependence graph of Figure 6.12. The graph contains four simple cycles. Let $c_k = c_i \circ c_j$ denote concatenation of cycles c_i and c_j such that there is a traversal that includes all the nodes and c_k contains at least one operation twice (hence, c_k is not a simple cycle).



Simple Cycles

$c_1 = \langle v_1, v_3, v_5, v_1 \rangle$

$c_2 = \langle v_1, v_5, v_1 \rangle$

$c_3 = \langle v_1, v_3, v_2, v_4, v_1 \rangle$

$c_4 = \langle v_4, v_6, v_4 \rangle$

Figure 6.12 : Simple cycles in a dependence graph

From Figure 6.12 and Equation 6.2 (on page 142) , we observe that the simple cycle c_3 determines the value of RecII $(= 2)$ for the graph. Let us now consider the cycle $c_5 = c_3 \circ c_4$. The length of c_5 is 6; the distance of c_5 is 4. The recurrence constrained initiation interval for c_5 is computed as follows:

$$\left\lceil \frac{6}{6} \right\rceil = 1 < RecII$$

Thus, we observe that RecII for simple cycles is always greater than that of non-simple cycles.

Theorem 6.1. *Given a dependence graph $G(V, E)$, it is sufficient to consider only simple cycles for computing RecII.*

Proof. Consider two simple cycles c_i and c_j of $G(V, E)$. It is easy to see that RecII must satisfy:

$$RecII > \max\left(\frac{len(c_i)}{dist(c_i)}, \frac{len(c_j)}{dist(c_j)}\right)$$

Now, let us consider a cycle $c_k = c_i \circ c_j$. The length of c_k is $len(c_i) + len(c_j)$; the distance of c_k is $dist(c_i) + dist(c_j)$. The recurrence constrained initiation interval of c_k is given by

$$\frac{len(c_i) + len(c_j)}{dist(c_i) + dist(c_j)}$$

since

$$\max\left(\frac{len(c_i)}{dist(c_i)}, \frac{len(c_j)}{dist(c_j)}\right) > \frac{len(c_i) + len(c_j)}{dist(c_i) + dist(c_j)}.$$

Therefore, it is sufficient to consider only simple cycles for computing RecII. The proof also holds for cycles obtained by concatenating more than two cycles. Such cases can be reduced hierarchically to the case discussed above. For example, a cycle $c_4 = c_1 \circ c_2 \circ c_3$ can be hierarchically reduced to $c_4 = {c'}_1 \circ c_3$, where ${c'}_1 = c_1 \circ c_2$. □

Finding the Initiation Interval

As discussed previously, the MII is a lower bound on an initiation interval that will yield a feasible schedule. Rau et al. [RG81] used linear search to find an initiation interval starting at MII; we have also used this approach in Algorithm 6.1. The FPS compiler [Tou84] used binary search to find an initiation interval where the lower bound is MII and the upper bound is the length of the locally compacted loop body. A problem with binary search is that the existence of a schedule is not monotonic in the initiation interval: even if a certain ii has no feasible schedule, there may be an $ii' < ii$ where ii' has a feasible schedule. One could also employ an enumerative branch-and-bound search of all possible schedules or use a trial-and error approach guided by some heuristics. However, the former is computationally very expensive, while the latter does not always guarantee compact schedules. Huff [Huf93] modeled the problem as a minimal cost-to-time ratio problem [Law76]. Feautrier [Fea94], Govindarajan et al. [GAG94], Eichenberger et al. [ED95], Altman et al. [AGR95] model modulo scheduling as an integer linear programming [dD94] problem. However, the exponential complexity of integer linear programming algorithms render such techniques impractical for large loops. Similarly, techniques such as *simulated annealing*, *Boltzmann machine algorithm* and *genetic algorithms* [Ree93] have been proposed but can become very time expensive for large loops.

Because binary search cannot be guaranteed to find the minimum feasible initiation interval, and because of the expense of the more theoretically advanced techniques, modulo scheduling implementations generally use the straightforward linear search for a feasible initiation interval beginning at a lower bound such as MII.

6.4.2 Limitations

Modulo scheduling constrains each iteration to have an identical schedule and schedules successive iterations at a fixed initiation

interval. However, in general, optimal schedules for many loops cannot be achieved by duplicating the schedule of the first iteration at fixed intervals. The constraint that each iteration has the same schedule is not necessary, but it is a fundamental component of modulo scheduling.

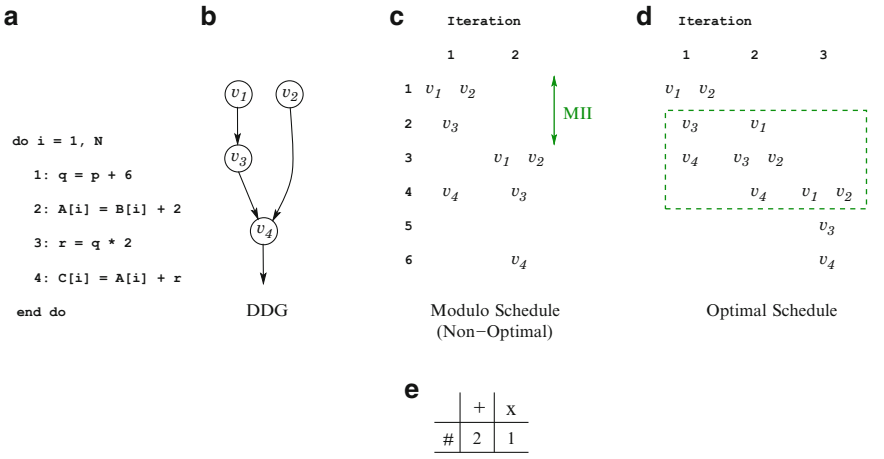


Figure 6.13 : Limitations of modulo scheduling

For example, consider the loop and its data dependence graph shown in Figure 6.13(a) and Figure 6.13(b), respectively. Let v_i^k denote the i-th operation of the k-th iteration. The modulo schedule of the data dependence graph in Figure 6.13(b) is shown in Figure 6.13(c). Given resources shown in Figure 6.13(e), ResII for the dependence graph is 2. Since there are no loop carried dependences $MII = ResII$. Note that operation v_4^1 cannot be scheduled in control step 3 as it would lead to a resource conflict with operations v_1^2, v_2^2 scheduled in control step 3. Thus, a fixed initiation interval in conjunction with the modulo constraint can potentially create "gaps" in the schedule.

In order to extract higher levels of parallelism, one must somehow relax the modulo scheduling constraints and:

- Allow successive iterations to be scheduled at different initiation intervals.
- Allow iterations to have different schedules.

An optimal schedule is shown in Figure 6.13(d). Observe that the first two iterations have different schedules. Subsequently, iteration

3 is scheduled like iteration 1, iteration 4 is scheduled like iteration 2 and so on. Further, the second and the third iterations are initiated with different initiation intervals. The schedule of the second iteration in Figure 6.13(d), helps close the gap in the first iteration in Figure 6.13(c), as v_4^1 can now be scheduled in control step 3 without any resource conflicts. Thus, relaxing the constraints of modulo scheduling facilitates compaction which yields better performance.

As scheduling of iterations progresses, the execution of operations tend to form a repeating kernel, as shown in Figure 6.13(d) by the 'boxed' set of operations. The schedule in Figure 6.13(d) can be computed by modulo scheduling if the loop is unrolled once before modulo scheduling is performed. The difficulty for modulo scheduling is that the optimal kernel has different schedules for odd and even numbered iterations. But the optimal schedule for the loop unrolled once has the same schedule for every iteration of the unrolled loop body, since each iteration of the unrolled loop includes one odd and one even iteration of the original loop. The problem is that it is not clear how to predict how much unrolling is required for modulo scheduling to yield optimal schedules, as it may depend on resource constraints, data dependences, and the interaction between the two. While such kernels are problematic for modulo scheduling, we discuss a technique in subsection 7.3 that can directly detect such kernels. We further discuss the use of loop unrolling to improve modulo scheduling in subsection 6.7.2.

6.5 Modulo Scheduling with Conditionals

As originally described in [RG81], modulo scheduling assumes that the loop body does not contain conditional or unconditional jumps. In this section we discuss techniques for modulo scheduling loops with conditional branches in the loop body.

6.5.1 Hierarchical Reduction

Hierarchical reduction [Lam88] was proposed to make loops with conditionals amenable for modulo scheduling. The hierarchical reduction technique, derived from [Woo79], reduces a conditional to a *compound operation* whose scheduling constraints (both inter-iteration and loop carried dependences) represent the combination of the scheduling constraints of the two branches. That is, the set of operations

that make up the conditional are grouped into a single operation. For each resource r, the requirement of the compound operation is the maximum of the use of r over the two branches. The latency of the compound operation is the latency of the longer branch. Finally, the dependences of the compound operation are all the dependences between constituent operations and other operations outside of the conditional.

The significance of hierarchical reduction is:

- Conditionals do not limit the overlapping of different iterations.
- It enables the movement of operations outside of a conditional around the conditional. Branches of different conditionals can be overlapped via hierarchical percolation scheduling (see Section 5.4.2).

The algorithm for hierarchical reduction can be summarized as follows: First, the `then` and `else` branches of a conditional statement are scheduled independently. Next, the entire conditional is reduced to a compound operation. In the presence of nested control flow, each conditional is scheduled hierarchically, starting with the innermost conditional.

The dependence graph thus obtained is free of conditionals and thus amenable to modulo scheduling using Algorithm 6.1. During code generation code is generated for both branches of a conditional. Any code scheduled in parallel with the conditional is duplicated in both branches.

Example 6.3. Consider the loop shown in Figure 6.14(a). The corresponding control flow graph is shown in Figure 6.14(b), where BB_i denotes the i-th basic block. From Figure 6.14(a), note that the operations v_2, v_3 and v_4 are dependent on operation v_1. Further, even though v_4 is in basic block BB_3, it can be scheduled in parallel with the conditional. As discussed above, the conditional, i.e., basic blocks BB_1 and BB_2, are replaced with the compound operation v'. The dependence graph of the transformed loop is shown in Figure 6.14(c).

The set of resources used by v' consists of an adder and a multiplier. Assuming the number of resources given in Figure 6.14(e),

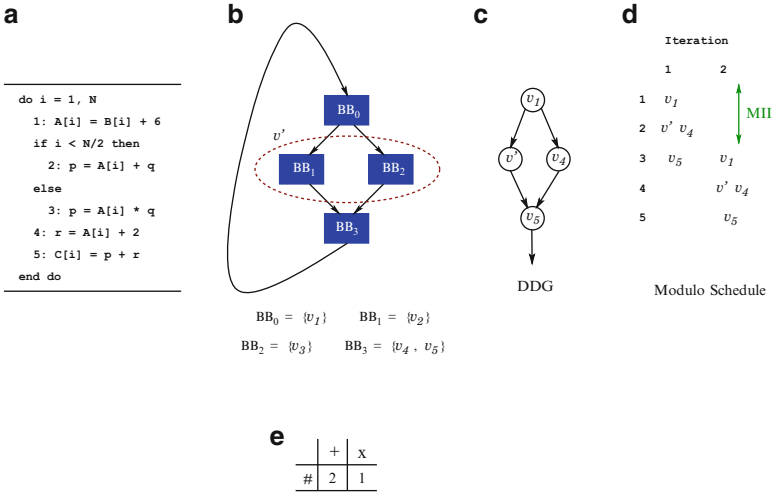


Figure 6.14 : Modulo scheduling with conditionals

modulo scheduling using Algorithm 6.1 results in the schedule shown in Figure 6.14(d).

6.5.2 Enhanced Modulo Scheduling

Predication (also known as *if-conversion*) is another technique for converting code with branches into data dependences, resulting in a single basic block [AKPW83, PS91]. A *predicated operation* consists of a predicate p and an operation o. The idea is that o is executed only if p is true. Typically the predicate is pre-computed, so that the execution of a predicated operation requires only checking one bit of state to determine whether the operation executes or not. For example, the conditional

if $x > 0$ **then** $x \leftarrow x + 1$ **else** $x \leftarrow 1$

could be written as the predicated code

true: $p \leftarrow x > 0$
p: $x \leftarrow x + 1$
$\neg p$: $x \leftarrow 1$

Predicated operations require special hardware support in the form of *predicate registers* to hold the condition for the operation. A predicate register is specified for each operation and predicate definition operations are used to set the predicate registers. Predication-based modulo scheduling techniques [RYYT89, DHB89] schedule operations along all execution paths together. Thus, the initiation interval for predicated execution is constrained by the resource usage of all the loop operations rather than those along the single execution path with maximum resource usage.

In the presence of multi-level branches, hierarchical predication [TLS90, MCH+92, MCB+93] or predicate matrices [MJ98] may be used to convert multi-level control dependences to data dependences. Early exit branches and multiple branches back to the loop header are handled in a similar fashion.

Enhanced Modulo Scheduling (EMS) [WBHS92] integrates hierarchical reduction with predicated execution to alleviate the limitations of each. Predication eliminates the need for pre-scheduling the conditional constructs, which is required in hierarchical reduction. Regenerating conditionals after modulo scheduling is complete eliminates the dependences on predicate registers assumed by predicated execution. Thus, EMS can be used on processors without support for predicated execution. We present an overview of the algorithm. First, the loop body (with conditionals) is converted into straight line predicated code using the RK algorithm [PS91]. The predicated loop body is then modulo scheduled using Algorithm 6.1 in conjunction with modulo variable expansion [Lam88] to rename registers with overlapping lifetimes. (Modulo variable expansion helps eliminate some data dependences, see subsection 6.7.1). Finally, the predicate definition operations are replaced with conditionals and predicated operations are placed in the appropriate branches.

Example 6.4. Consider the loop shown in Figure 6.15(a). The corresponding predicated code is shown in Figure 6.15(b). A predicate has an id and $type$, represented as a pair $\langle id, type \rangle$.[3] For example, operation v_4 in Figure 6.15(b) has a predicate with $id = p$ and $type = F$ (false—i.e., v_4 will execute if p is false). Operation v_3 sets the predicate if $x < N/2$ and clears the predicate otherwise. Operations that

[3] We follow the predicate notion proposed in [WBHS92].

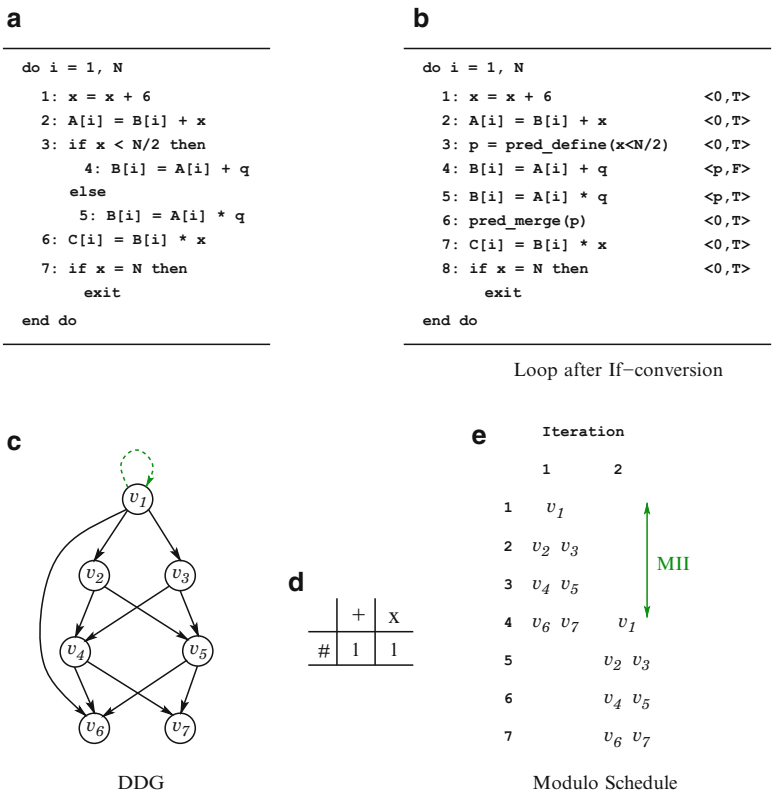


Figure 6.15 : Enhanced Modulo scheduling

are not dependent on a predicate such as v_1 are assigned the default predicate $\langle 0, type \rangle$. Operation v_7 is a loop bounds check and therefore does not define a new predicate.

The data dependence graph of the predicated loop is shown in Figure 6.15(c). Assuming the number of resources given in Figure 6.15(d), the MII for the predicated loop is 4. The predicated loop is then modulo scheduled, shown in Figure 6.15(e). Subsequently, conditionals are regenerated by replacing each predicate operation with a conditional; operations executed in parallel with a predicated operation are duplicated along both the branches of the corresponding conditional.

6.5.3 Modulo Scheduling with Multiple Initiation Intervals

A problem with hierarchical reduction is that if a conditional has branches of very different lengths then the initiation interval for the reduced loop body will be constrained by the worst path. Furthermore, the two paths of a conditional may have very different dependences (both intra-iteration and loop carried dependences). In general,

$$MII \geq \max_{\forall\, i} II_{path(i)} \tag{6.4}$$

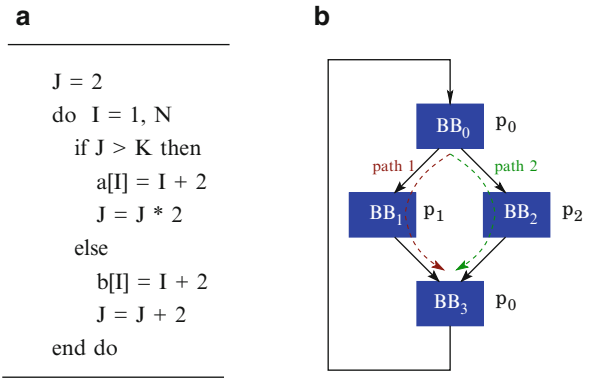


Figure 6.16 : A loop with control flow

where $II_{path(i)}$ is the initiation interval of path i. (Because hierarchical reduction aggregates resource and dependence constraints over all paths, note that the MII may be greater than the minimum initiation interval for any individual path.) From Equation 6.4 one observes that a path with a shorter initiation interval is penalized due to other paths. Similarly, when using predicated execution the MII is determined by the sum of the costs of all operations in the loop body. Thus, all paths are penalized due to the fixed MII. As an illustration, consider the example in Figure 6.16(a).

Assuming an availability of one adder and one multiplier, the MII of basic block BB_1 is 1 whereas the MII of basic block BB_2 is 2. A fixed MII using hierarchical modulo scheduling assigns an MII of 2 to both paths, which slows down the execution of path 1.

In [WP95], modulo scheduling with multiple initiation intervals was proposed for architectures with support for predicated execution. In Figure 6.16(b) the predicate for a basic block is shown next to

the block. Predicate p_0 is the predicate `true` (i.e., operations predicated with p_0 always execute). After if-conversion, path 1 corresponds to the operations predicated on p_0 and p_1, and path 2 to the operations predicated on p_0 and p_2. The initiation intervals for the two paths is shown in Figure 6.17.

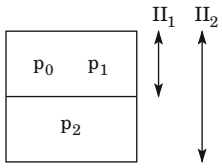


Figure 6.17 : Multiple initiation intervals

From Figure 6.17 one notes that during execution of path 1, only operations from path 1 are fetched and executed. However, during the execution of path 2, operations from both the paths are fetched and operations predicated on p_1 are squashed. Path 2 corresponds to a longer initiation interval II_2 due to the larger number of operations. Notice that path 1 is never penalized. The execution paths of a loop body often have different execution frequencies. In order to minimize penalty during the scheduling process, the algorithm assigns higher priorities to the more frequently executed paths, i.e., smaller initiation intervals are assigned to the more frequently executed paths. A detailed description of the algorithm can be found in [WP95].

6.6 Iterative Modulo Scheduling

In the modulo scheduling algorithms presented so far, operations are scheduled in some order and if at any point an operation cannot be scheduled the entire partial schedule is abandoned and the algorithm retries with a larger initiation interval. In this section we discuss *iterative modulo scheduling* [Rau95], a modulo scheduling algorithm that does not discard the schedule when it is discovered to be infeasible. The algorithm is iterative in the sense that it schedules and reschedules operations to find a schedule that simultaneously satisfies all the scheduling constraints. When the scheduler finds that there is no slot available for scheduling the current operation, it displaces one or more previously scheduled, conflicting operations. These are in turn re-scheduled as the search continues. If the search fails to yield

a valid schedule even after a large number of steps, then the initiation interval is increased and the entire process is repeated.

6.6.1 The Algorithm

Each time an operation is scheduled, the iterative modulo scheduling algorithm *tentatively* schedules the same operation of subsequent iterations at intervals of *II*. The algorithm allows the unscheduling of other tentatively scheduled operations from subsequent iterations due to resource or dependence conflicts. When possible, however, operations are scheduled so as not to unschedule any operation that was tentatively scheduled previously. Note that since it is only a tentative schedule, it does not block the forward progress of the scheduler in the event that the current operation cannot be scheduled because it conflicts with one or more operations from a subsequent iteration.

Algorithm 6.2. Iterative Modulo Scheduling (IMS)
The input to this algorithm is a dependence graph $G(V, E)$ of a loop $\mathcal{L}$ and a user-defined parameter *Budget* giving the maximum number of operations scheduled before giving up and trying a larger initiation interval. The dependence graph has two distinguished nodes - *start* and *stop* - which represent the entry and exit points of the graph. As before, the MII is determined using Equation 6.2 (on page 142). For simplicity, unscheduling is omitted from the pseudo-code, but is discussed below.

// Initialize *II*
II $\leftarrow$ *MII*
isScheduled $\leftarrow$ *false*
while $\neg$ *isScheduled* **do**
  Mark all operations unscheduled
  Insert all operations in the list of unscheduled operations
  for each $v \in V$ **do**
    Compute height-based priorities
  endfor
  Schedule *start* at time 0
  Budget $\leftarrow$ *Budget* -1
  while *all operations not scheduled* and *Budget* $\neq 0$ **do**
    // Pick the highest priority operation v_h

```
    v_h = HighestPriorityOperation()
    // Determine earliest start time, ASAP(v_h)
    ASAP(v_h) = CalculateEarlyStartTime(v_h)
    ALAP(v_h) = ASAP(v_h) + II - 1
    // Find time slot to schedule v_h
    t = FindTimeSlot (v_h, ASAP(v_h), ALAP(v_h))
    Schedule v_h at time t   (*)
    Budget ← Budget -1
  endwhile
  if all operations scheduled then
    isScheduled ← true
  else
    Set Budget to its initial value
    II ← II +1
  endif
endwhile
```

Like other modulo scheduling algorithms, IMS maintains the invariant that at each step all resource and dependence constraints are satisfied for the operations in the partially constructed schedule. Thus, if the algorithm reaches a point where all the operations from one iteration have been scheduled (without having to displace any tentatively scheduled operations from the subsequent iterations) then, by construction, the complete data dependence graph has been modulo scheduled. The repeating portion of the resulting schedule forms the body of the new loop; this loop is preceded by a prologue and followed by an epilogue which set up and complete correct execution of the new loop. This is a special case of software pipelining using kernel recoginition, which we describe in detail in Chapter 7.

Unlike traditional acyclic list schedulers, IMS may also *unschedule* an operation after it has been scheduled. Specifically, assume an operation o cannot be scheduled on line $(*)$ in the algorithm given above due to resource or dependence constraints with already-scheduled operations. In this situation IMS schedules o anyway at the selected time. If scheduling o violates a data dependence with operation o', then o' is unscheduled (i.e., removed from the schedule). If scheduling o violates a resource constraint with some set of operations $o_1, o_2, \ldots$ that all use that same resource at the same time, then one of the o_i is unscheduled to make room for o. Any operation that is unscheduled is put back on the list of unscheduled operations with its original priority.

Determining Scheduling Priority

Standard list scheduling techniques [ACD74] employ a *height-based* priority function [Hu61, RCG72], which has two important properties:

- If there is a dependence between two operations, then the priority function gives the predecessor operation a higher priority than the successor operation.
- Among operations that are independent, higher priority goes to operations with smaller *slack*, which is the difference between the ALAP and ASAP labels of an operation (see Chapter 3). Intuitively, an operation o with small slack has a narrow range of time steps in which it can be placed without lengthening the overall schedule and so o should be given high scheduling priority once all of o's dependence predecessors are scheduled.

The conventional height-based priority works only on acyclic dependence graphs and so must be extended in the context of IMS to account for loop carried dependences.

Definition 6.3. *A strongly connected component (SCC) of a directed graph $G(V, E)$ is a maximal set of vertices $U \subseteq V$ such that every pair of vertices in U are reachable from each other.*

IMS partitions the data dependence graph into a set of SCCs. Each SCC in the graph is replaced by a *super node*. Hence, the resulting graph obtained is acyclic. The height of all nodes (both normal nodes and super nodes) is computed by a depth-first walk of this acyclic graph. After this computation, the heights of the vertices in each SCC are computed iteratively by a post-order visit beginning at an entry point to the SCC. Consider a successor v of operation u with a distance of $D(u, v)$ (recall that the distance between two operations is the number of iterations separating them). Let the latency between the operations u and v be denoted by $\mathcal{D}(u, v)$. Assume that the operation u that is in the same iteration as v (the current iteration) has a height-based priority h. Since, u's successor v is actually $D(u, v)$ iterations later, its height-based priority relative to current iteration is effectively given by:

$$\hbar(u) = \begin{cases} 0 & \text{if } u = \textit{stop}, \\ \max\limits_{v \in \text{Succ}(u)} (\hbar(v) + \mathcal{D}_{\text{eff}}(u, v)) & \text{otherwise.} \end{cases} \tag{6.5}$$

where $\hbar(u)$ is the height of an operation u, *Succ(u)* is the set of successors of u and $\mathcal{D}_{eff}(u, v)$ (effective latency between operation u and v) is given by:

$$\mathcal{D}_{\text{eff}}(u, v) = \mathcal{D}(u, v) - II \times D(u, v)$$

Determining Earliest Start Time

The function *CalculateEarlyStartTime* determines the earliest time an operation can be scheduled as constrained by its predecessors. The calculation of *ASAP(v)* is affected by the fact that operations can be unscheduled—it may be that some of the predecessors of v are not scheduled when v is scheduled. We modify $ASAP(v_h)$ so that it can be calculated by considering only the scheduled immediate predecessors of the operation. The earliest time an operation v can be scheduled is given by :

$$\text{ASAP}(v) = \max_{u \in \text{Pred}(v)} (\text{ASAP}'(u) + \mathcal{D}(u, v)) \tag{6.6}$$

where $\text{ASAP}'(u)$ is given by :

$$\text{ASAP}'(u) = \begin{cases} 0, & \text{if } u \text{ is unscheduled} \\ \max(0, \text{SchedTime}(u) + \mathcal{D}_{\text{eff}}(u, v)) & \text{otherwise} \end{cases}$$

where, *SchedTime(u)* is the time at which operation u is scheduled.

Determining Candidate Time Slots

In iterative modulo scheduling, it is impossible to guarantee that all the predecessors of an operation o have been scheduled and have remained scheduled. That being the case, dependences with the predecessor operations are honored by not scheduling o before ASAP(o). To honor dependences with successor operations, there is also a latest time by which o must be scheduled. However, the two conditions

may be incompatible—there may be no time at which o can be scheduled without violating predecessor, successor, or resource constraints. In this situation, o is scheduled in a time step that satisfies predecessor constraints but may violate successor or resource constraints. Any successor constraints that are violated cause that successor operation to be unscheduled. For resource conflicts, there may be several operations that use the same resource; the lowest priority operation is selected.

6.7 Optimizations

In this section we discuss auxiliary optimizations that can play a large role in extracting parallelism from loops. Note that these optimizations can benefit trace scheduling as well, but have the greatest impact on loop parallelization.

6.7.1 Modulo Variable Expansion

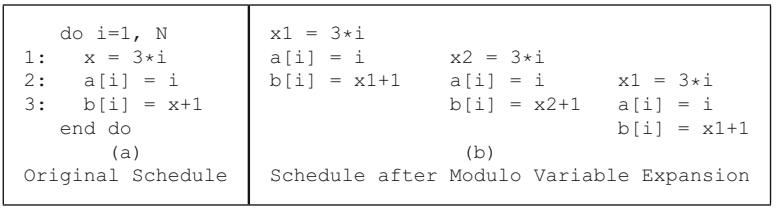


Figure 6.18 : Modulo Variable Expansion

Modulo variable expansion [Lam88] applies a static version of a traditional register renaming technique to remove some inter-iteration data dependencies. Consider the example in Figure 6.18(a) where a value is written into a register and used two cycles later. Assuming we have one adder and one multiplier available, the *ResII* is 1. However, due to the loop carried anti-dependence (recall Definition 3.1) on x a new iteration cannot be started before operation 3 of the preceding iteration (refer to Figure 6.18(a)) has completed. Therefore, the throughput is limited to one iteration every three cycles. The code can be sped up using different registers for the variable x in alternating iterations, as shown in Figure 6.18(b). The transformation identifies any variable such as x with a loop carried anti-dependence and

"expands" that variable into multiple variables so that each iteration can refer to a different location. By construction, the uses of the variable(s) in overlapping iterations are independent, and the loop can be pipelined with a shorter *II*. Modulo variable expansion minimizes renaming, and thus the number of additioinal memory locations required, by reusing the locations in non-overlapping iterations. Also, modulo variable expansion in association with register pipelining reduces memory traffic, and hence improves program execution time significantly.

6.7.2 Using Loop Unrolling to Enhance Modulo Scheduling

Modulo scheduling requires an integral initiation interval. The rounding of the initiation interval to an integer (recall Equation 6.4 on page 155) limits overlapping of operations from different iterations than is possible otherwise, thereby leading to performance degradation. Furthermore, the initiation interval is restricted to a minimum value of 1, limiting the performance of a modulo scheduled loop to one iteration per cycle. Loop unrolling (recall Section 6.2 on page 135), before applying MS, helps to mitigate the effect of rounding of the initiation interval by creating larger loop bodies which require more resources, thus increasing ResII [LH95]. The larger the ResII, the smaller the impact of rounding up the initiation interval to the next integer. Similarly, RecII may not be an integer if the length of a cycle is not an integral multiple of the distance of the cycle. In such cases, loop unrolling is performed to reduce the distance of the recurrence.

Unrolling facilitates (upward) percolation of operations from successive iterations as the operations are free of the iteration boundaries. Further, unrolling increases the rate at which iterations are executed as illustrated in the following example. Thus, in effect, a non-integer initiation interval is achieved as more than one iteration of the original loop is initiated during the same initiation interval.

Example 6.5. Consider the loop and its dependence graph shown in Figure 6.19(a) and Figure 6.19(b) respectively. Assuming the number of resources given in Figure 6.19(f), the corresponding modulo

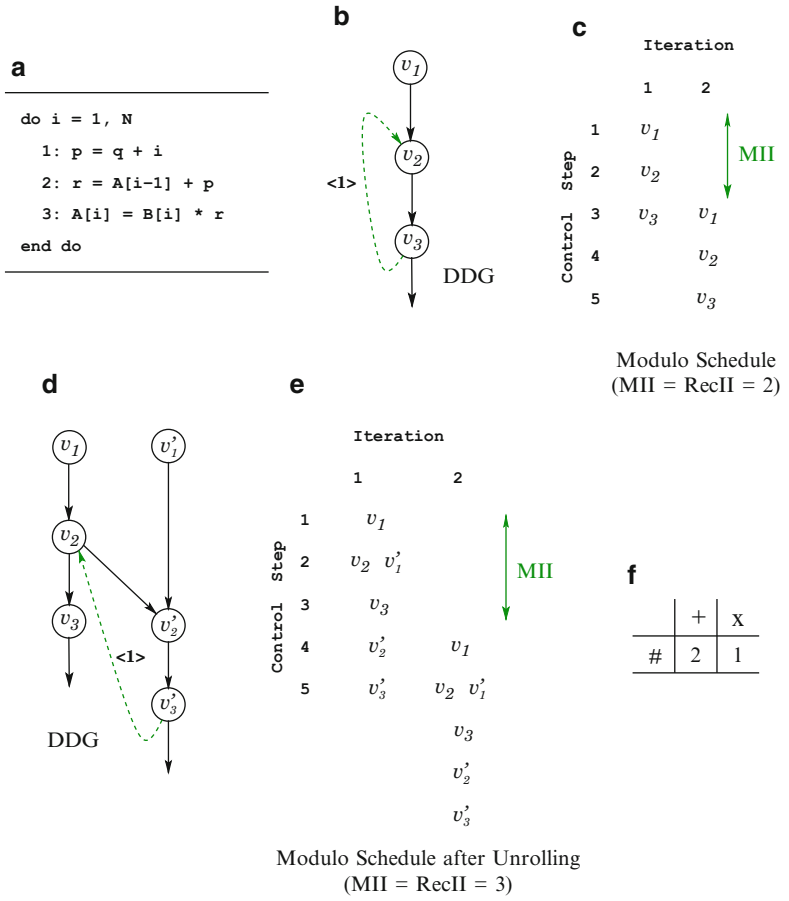


Figure 6.19 : Modulo scheduling with unrolling

schedule is shown in Figure 6.19(c) where successive iterations are initiated every two control steps.

In order to achieve a smaller initiation interval, we unroll the loop once (i.e., add a second copy of the loop body to the loop). The dependence graph of the unrolled loop body is shown in Figure 6.19(d); operations, denoted by v'_i, correspond to the new operations in the loop body, introduced as a result of loop unrolling. The modulo schedule of the unrolled loop is shown in Figure 6.19(e). From Figure 6.19(e) we observe that two iterations of the original loop are initiated every

three control steps. Therefore the effective initiation interval is 1.5. Thus, unrolling the loop prior to modulo scheduling reduces the initiation interval.

Additionally, unrolling may enable other optimizations which can reduce the resource requirements and dependence height [MCGH92]. Though loop unrolling exposes higher levels of instruction level parallelism, it must be used carefully. Loop unrolling may result in an increase in instruction cache misses and register pressure,[4] which in the worst case can more than cancel the expected benefit of the transformation. Note that the gain in performance decreases progressively with increasing code size. Heydemann et al. [HBK+03] proposed the *UFC (Unrolling Factor computation under Constraints)* method to compute unrolling factors of a set of loops while taking into account code size.

FURTHER READING

A comparative study of the various modulo scheduling techniques is presented in [CLG02, KN11]. Modulo scheduling of control-intensive loops in non-numerical programs is discussed in [LH96]. A cache-aware modulo scheduling technique is discussed in [SG97]. A clustered modulo scheduling technique for VLIW architectures with distributed cache is discussed in [SG01]. Ding et al. proposed a new modulo scheduling technique to boost cache reuse [DCS97]. Code generation schemas for modulo scheduled loops are discussed in [RST92]. In [LF02], Llosa and Freudenberger discuss a reduced code size modulo scheduling technique; Merten and Hwu proposed a new architectural mechanism, called *Modulo schedule buffers*, to contain code expansion during modulo scheduling [MH01].

Clustered VLIW architectures are typically characterized with clusters of a few functional units and small private register files. In [NE98], Nystrom and Eichenberger proposed a cluster assignment for modulo scheduling; other approaches for cluster assignment during modulo scheduling include [ACSG01, ACS+02]. Instruction scheduling in general for clustered VLIW architectures is discussed in [SG00b]. Various modulo scheduling approaches have been proposed for such architectures [FLT99, SG00c]. The effectiveness of loop unrolling for modulo scheduling in the context of clustered architectures is discussed in [SG00a].

A large number of approaches have been proposed over the last three decades for loop pipelining. In [GS94], Gasperoni and Schwiegelshohn propose an algorithm

[4] The longer unrolled loop body may result in different register allocation that increases register pressure.

for scheduling loops on parallel processors. In [WE93] Wang and Eisenbeis proposed a technique called *Decomposed Software Pipelining* wherein software pipelining is considered as an instruction level transformation from a vector of one-dimension to a matrix of two-dimensions. In particular, the software pipelining problem is decomposed into two subproblems—one is to determine the row-numbers of operations in the matrix and another is to determine the column-numbers. In [RGSL96], Ruttenberg et al. compared optimal and heuristic methods for modulo scheduling. Allan et al. surveyed various techniques for software pipelining in [AJLA95].

Several techniques [JPSW82, GFO92, Ree93] have also been proposed to find the global minimum of the search space of feasible schedules. However, the high complexity of such approaches makes them currently very expensive in practice.