Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops

B. Ramakrishna Rau

Hewlett-Packard Laboratories, 1501 Page Mill Road, Bldg. 3L, Palo Alto, CA 94304

Abstract

Modulo scheduling is a framework within which a wide variety of algorithms and heuristics may be defined for software pipelining innermost loops. This paper presents a practical algorithm, iterative modulo scheduling, that is capable of dealing with realistic machine models. This paper also characterizes the algorithm in terms of the quality of the generated schedules as well the computational expense incurred.

Keywords: modulo scheduling, instruction scheduling, software pipelining, loop scheduling.

1 Introduction

It is well known that, as a rule, there is inadequate instruction-level parallelism (ILP) between the operations in a single basic block and that higher levels of parallelism can only result from exploiting the ILP between successive basic blocks. Global acyclic scheduling techniques, such as trace scheduling [13, 23] and superblock scheduling [19], do so by moving operations from their original basic blocks to preceding or succeeding basic blocks. In the case of loops, the successive basic blocks correspond to the successive iterations of the loop rather than to a sequence of distinct basic blocks.

Various cyclic scheduling schemes have been developed in order to achieve higher levels of ILP by moving operations across iteration boundaries, i.e., either forward to previous iterations or backward to succeeding iterations. One approach, "unroll-before-scheduling", is to unroll the loop some number of times and to apply a global acyclic scheduling algorithm to the unrolled loop body [13, 19, 23]. This achieves overlap between the iterations in the unrolled loop body, but still maintains a scheduling barrier at the back-edge. The resulting performance degradation can be reduced by increasing the extent of the unrolling, but it is at the cost of increased code size.

Software pipelining [8] refers to a class of global cyclic scheduling algorithms which impose no such scheduling barrier. One way of performing software pipelining, the "move-then-schedule" approach, is to move instructions, one by one, across the back-edge of the loop, in either the forward or the backward direction [11, 12, 20, 15, 28]. Although such code motion can yield improvements in the schedule, it is not always clear which operations should be moved around the back edge, in which direction and how many times to get the best results. The process is somewhat arbitrary and reminiscent of early attempts at global acyclic scheduling by the ad hoc motion of code between basic blocks [42]. On the other hand, this currently represents the only approach to software pipelining that at least has the potential to handle loops containing control flow in a near-optimal fashion, and which has actually been implemented [28]. How close it gets, in practice, to the optimal has not been studied and, in fact, for this approach, even the notion of "optimal" has not been defined.

The other approach, the "schedule-then-move" approach, is to instead focus directly on the creation of a schedule that maximizes performance, and to subsequently ascertain the code motions that are implicit in the schedule. Once again, there are two ways of doing this. The first, "unroll-while-scheduling", is to simultaneously unroll and schedule the loop until one gets to a point at which the rest of the schedule would be a repetition of an existing portion of the schedule [3]. Instead of further unrolling and scheduling, one can terminate the process by generating a branch back to the beginning of the repetitive portion. Recognition of this situation requires that one maintain the state of the scheduling process, which includes at least the following information: knowledge of how many iterations are in execution and, for each one, which operations have been scheduled, when their results will be available, what machine resources have been committed to their execution into the future and are, hence, unavailable, and which register has been allocated to each result. All of this has to be identical if one is to be able to branch back to a previously generated portion of the schedule. Computing, recording and comparing this state presents certain engineering challenges that have not yet been addressed by a serious implementation. On the other hand, by focusing solely on creating a good schedule, with no scheduling barriers and no ad hoc, a priori decisions regarding inter-block code motion, such unroll-while-scheduling schemes have the potential of yielding very good schedules even on loops containing control flow.

Another "schedule-then-move" approach is modulo scheduling [34], a framework within which algorithms of this kind may be defined[1]. The framework specifies a set of constraints that must be met in order to achieve a legal modulo schedule. The objective of modulo scheduling is to engineer a schedule for one iteration[2] of the loop such that when this same schedule is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or distinct iterations. This constant interval between the start of successive iterations is termed the **initiation interval (II)**. In contrast to unrolling approaches, the code expansion is quite limited. In fact, with the appropriate hardware support, there need be no code expansion whatsoever [36]. Once the modulo schedule has been

MICRO 27- 11/94 San Jose CA USA
© 1994 ACM 0-89791-707-3/94/0011..$3.50

[1] The original use of the term "software pipelining" by Charlesworth was to refer to a limited form of modulo scheduling However, current usage of the term has broadened its meaning to the one indicated here

[2] As we shall see shortly, in certain cases it may be benefical to unroll the loop body a few times prior to modulo scheduling, in which case, the "single iteration" that we are discussing here may correspond to multiple iterations of the original loop However, unrolling is not an essential part of modulo scheduling

created, all the implied code motions and the complete structure of the code, including the placement and the target of the loopback branch, can all be determined [33].

Modulo scheduling an innermost loop consists of a number of steps, only one of which is the actual modulo scheduling process.

- In general, the body of the loop is an acyclic control flow graph. With the use of either profile information or heuristics, only those control flow paths that are expected to be frequently executed can be selected as is done with hyperblock scheduling [25]. This defines the region that is to be modulo scheduled.
- Within this region, memory reference data flow analysis and optimization are performed in order to eliminate partially redundant loads and stores [32, 10]. This can improve the schedule if either a load is on a critical path or if the memory ports are the critical (most heavily used) resources.
- At this point, the selected region is IF-converted, with the result that all branches except for the loop-closing branch disappear [4, 29, 10]. With control flow converted to data dependences involving predicates [37, 5], the region now looks like a single basic block.
- Anti- and output dependences are minimized by putting the computation into the dynamic single assignment form [32].
- If control dependences are the limiting factor in schedule performance, they may be selectively ignored thereby enabling speculative code motion [41, 24].
- Back-substitution of data and control dependences may be employed to further reduce critical path lengths [38, 10].
- Next, the lower bound on the initiation interval is computed. If this is not an integer, and if the percentage degradation in rounding it up to the next larger integer is unacceptably high, the body of the loop may be unrolled prior to scheduling.
- At this point, the actual modulo scheduling is performed.
- If rotating registers [37, 5] are absent, the kernel (i.e., the new loop body after modulo scheduling has been performed) is unrolled to enable modulo variable expansion [21].
- The appropriate prologue and epilogue code sequences are generated depending on whether this is a DO-loop, WHILE-loop or a loop with early exits, and on whether predicated execution and rotating registers are present in the hardware [36]. (The schedule for the kernel may be adapted for the prologue and epilogues. Alternatively, the prologue and epilogues can be scheduled along with the rest of the code surrounding the loop while honoring the constraints imposed by the schedule for the kernel.)
- Rotating register allocation [35] (or traditional register allocation if modulo variable expansion was done) is performed for the kernel. The prologue and epilogues are treated along with the rest of the code surrounding the loop in such a way as to honor the constraints imposed by the register allocation for the kernel.
- Finally, if the hardware has no predicated execution capability [37, 5], reverse IF-conversion [46] is employed to regenerate control flow.

The subject of this paper is the modulo scheduling algorithm itself, which is at the heart of this entire process. This includes the computation of the lower bound on the initiation interval. The reader is referred to the papers cited above for a discussion of the other steps that either precede or follow the actual scheduling.

Although the modulo scheduling framework was formulated over a decade ago [34], at least two product compilers have incorporated modulo scheduling algorithms [30, 10], and any number of research papers have been written on this topic [16, 21, 41, 39, 44, 45, 18], there exists a vague and unfounded perception that modulo scheduling is computationally expensive, too complicated to implement, and that the resulting schedules are sub-optimal. In large part, this is due to the fact that there has been little work done to evaluate and compare alternative algorithms and heuristics for modulo scheduling from the viewpoints of schedule quality as well as computational complexity.

This paper takes a first step in this direction by describing a practical modulo scheduling algorithm which is capable of dealing with realistic machine models. Also, it reports on a detailed evaluation of the quality of the schedules generated and the computational complexity of the scheduling process. For lack of space, this paper does not even attempt to provide a comparison of the algorithm described here to other alternative approaches for software pipelining. Such a comparison will be reported elsewhere. Also, the goal of this paper is not to justify software pipelining. The benefits of this, just as with any other compiler optimization or transformation, are highly dependent upon the workload that is of interest. Each compiler writer must make his or her own appraisal of the value of this capability in the context of the expected workload.

The remainder of this paper is organized as follows. Section 2 discusses the algorithms used to compute the lower bound on the initiation interval. Section 3 describes the iterative modulo scheduling algorithm. Section 4 presents experimental data on the quality of the modulo schedules and on the computational complexity of the algorithms used, and Section 5 states the conclusions.

2 The minimum initiation interval (MII)

Modulo scheduling requires that a candidate II be selected before scheduling is attempted [34]. A smaller II corresponds to a shorter execution time. The **minimum initiation interval (MII)** is a lower bound on the smallest possible value of II for which a modulo schedule exists. The candidate II is initially set equal to the MII and increased until a modulo schedule is obtained. The MII can be determined either by a critical resource that is fully utilized or a critical chain of dependences running through the loop iterations. The MII can be calculated through an analysis of the computation graph for the loop body. One lower bound is derived from the resource usage requirements of the computation. This is termed the **resource-constrained MII (ResMII)**. The **recurrence-constrained MII (RecMII)** is derived from latency calculations around elementary circuits in the dependence graph for the loop body. The MII must be equal to or greater than both the ResMII and the RecMII. Thus

$$MII = Max (ResMII, RecMII).$$

Any legal II must be equal to or greater than the MII. It should be noted that, in the face of recurrences and/or complex patterns of resource usage, the MII is not necessarily an achievable lower bound [33].

2.1 The resource-constrained MII (ResMII)

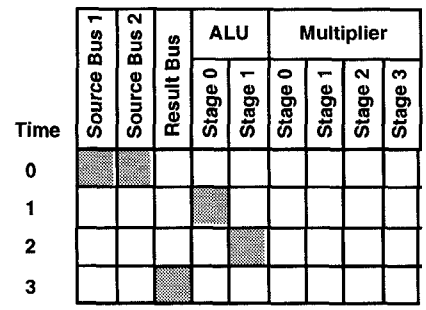
The resource-constrained lower bound on the II, **ResMII**, is calculated by totaling, for each resource, the usage requirements imposed by one iteration of the loop. For this purpose, we shall consider resources that are at the level of a pipeline stage of a functional unit, a bus or a field in the instruction format. The resource usage of a particular opcode is specified as a list of resources and the attendent times at which each of those resources is used by the operation relative to the time of issue of the operation. Figure 1a is a pictorial representation of the resource usage pattern of a highly pipelined ALU operation, with an execution latency of four cycles, which uses the two source operand buses on the cycle of issue, uses the two pipeline stages of the ALU on the next two cycles, respectively, and then uses the result bus on its last cycle of execution. Likewise, Figure 1b shows the resource usage pattern of a multiply operation on the multplier pipeline. This method of modelling resource usage is termed a **reservation table** [9].

From these two reservation tables, it is evident that an ALU operation (such as an add) and a multiply cannot be scheduled for issue at the same time since they will collide in their usage of the source buses. Furthermore, although a multiply may be issued any number of cycles after an add, an add may not be issued two cycles after a multiply since this will result in a collision on the result bus.

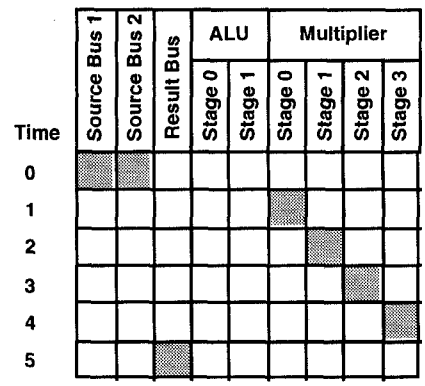
When performing scheduling with a realistic machine model, a data structure similar to the reservation table is employed to record that a particular resource is in use by a particular operation at a given time. We shall refer to this as the **schedule reservation table** to distinguish it from those for the individual operations. When an operation is scheduled, its resource usage is recorded by translating, in time, its own reservation table by an amount equal to the time at which it is scheduled, and then overlaying it on the schedule reservation table. Scheduling it at that time is legal only if the translated reservation table does not attempt to reserve any resource at a time when it is already reserved in the schedule reservation table. When backtracking, an operation may be "unscheduled" by reversing this process.

The nature of the reservation tables for the opcode repertoire of a machine determine the complexity both of computing the ResMII and of scheduling the loop. A **simple reservation table** is one which uses a single resource for a single cycle on the cycle of issue. A **block reservation table** uses a single resource for multiple, consecutive cycles starting with the cycle of issue. Any other type of reservation table is termed a **complex reservation table**. Block and complex reservation tables cause increasing levels of difficulty for the scheduler. Both reservation tables in Figure 1 are complex. If, however, the ALU and multiplier possessed their own source and result buses and if all operations that used these two pipelines used precisely the same reservation table, then both reservation tables could be abstracted by simple reservation tables.

A particular operation may be executable on multiple functional units, in which case it is said to have multiple **alternatives**, with a different reservation table corresponding to each one. Furthermore, these functional units might not be equivalent. For instance, a floating-point multiply might be executable on two functional units of which only one is capable of executing divide operations.



(a)



(b)

Figure 1. Reservation tables for (a) a pipelined add, and (b) a pipelined multiply.

The exact ResMII can be computed by performing a bin-packing of the reservation tables for all the operations. Bin-packing is a problem which is of exponential complexity. Complex reservation tables and multiple alternatives make it worse yet and it is impractical, in general, to compute the ResMII exactly. Instead, an approximate value must be computed. Accordingly, the ResMII is computed by first sorting the operations in the loop body in increasing order of the number of alternatives, i.e., degrees of freedom. As each operation is taken in order from this list, the number of times it uses each resource is added to the usage count for that resource. For each operation, that alternative is selected which yields the lowest partial ResMII, i.e., the usage count of the most heavily used resource at that point. When all operations have been considered, the usage count for the most heavily used resource constitutes the ResMII.

2.2 The recurrence-constrained MII (RecMII)

A loop contains 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. The dependences in question may either be data dependences (flow, anti- or output) or control dependences. Clearly, in the chain of dependences between the two instances of the operation, one or more

dependences must be between operations that are in different iterations. We shall refer to such dependences as **inter-iteration dependences**[1]. Dependences between operations in the same iteration are termed **intra-iteration dependences**. A single notation can be used to represent both types of dependences. The **distance** of a dependence is the number of iterations separating the two operations involved. A dependence with a distance of 0 connects operations in the same iteration, a dependence from an operation in one iteration to an operation in the next one has a distance of 1, and so on. All undesirable anti- and output dependences are assumed to have been eliminated, in a preceeding step, by the use of expanded virtual registers (EVRs) and dynamic single assignment [32]. Briefly, an EVR extends the concept of a virtual register to one that can retain the entire sequence of values written to that EVR. Since earlier values are never overwritten and destroyed, anti-dependences can be eliminated, even in cyclic code in which the same operation is executed repeatedly. (Of course, like conventional virtual registers, EVRs cannot be implemented in hardware. This discrepancy is handled by the register allocator. Rotating registers [37, 5] provide hardware support for EVRs in innermost loops, but are not essential.)

Table 1: Formulae for calculating the delay on dependence edges.

Type of dependence	Delay	Conservative Delay
Flow dependence	Latency(pred)	Latency(pred)
Anti-dependence	1-Latency(succ)	0
Output dependence	1+Latency(pred)-Latency(succ)	Latency(pred)

The dependences can be represented as a graph, with each operation represented by a vertex in the graph and each dependence represented by a directed edge from an operation to one of its immediate successor operations. There may be multiple edges, possibly with opposite directions, between the same pair of vertices. The dependence distance is indicated as a label on the edge. Additionally, each edge possesses a second attribute, which is the **delay**, i.e., the minimum time interval that must exist between the start of the predecessor operation and the start of the successor operation. In general, this is influenced by the type of the dependence edge and the execution latencies of the two operations as specified in Table 1. For a classical VLIW processor with non-unit architectural latencies, the delay for an anti-dependence or output dependence can be negative if the latency of the successor is sufficiently large. This is because it is only necessary that the predecessor start at the same time as or finish before, respectively, the successor finishes. A more conservative formula for the computation of the delay, which assumes only that the latency of the successor is not less than 1, is also shown in Table 1. This is more appropriate for superscalar processors.

The recurrence-constrained lower bound on II, **RecMII**, is calculated using this dependence graph. The existence of a recurrence manifests itself as a circuit in the dependence graph. Assume that the sum of the delays along some elementary circuit[2] c in the graph is Delay(c) and that the sum of the distances along that circuit is Distance(c). The existence of such a circuit imposes the constraint that the scheduled time interval between an operation on this circuit and the same operation Distance(c) iterations later must be at least Delay(c). However, by definition, this time interval is Distance(c)*II. Consequently, we have the constraint that

$$\text{Delay}(c) - \text{II}*\text{Distance}(c) \leq 0.$$

This is the constraint upon the II imposed by this one recurrence circuit. The RecMII is determined by considering the worst-case constraint across all circuits. One approach is to enumerate all the elementary circuits in the graph [40, 26] as was done in the Cydra 5 compiler, calculate the smallest value of II that satisfies the above inequality for that circuit, and to use the largest such value across all circuits.

The second approach, the one used in this study, is to pose the problem as a minimal cost-to-time ratio cycle problem [22] as proposed by Huff [18]. The algorithm ComputeMinDist computes, for a given II, the MinDist matrix whose [i, j] entry specifies the minimum permissible interval between the time at which operation i is scheduled and the time at which operation j, in the same iteration, is scheduled. If there is no path from i to j in the dependence graph, the value of the entry is $-\infty$. If MinDist[i, i] is positive for any i, it means that i must be scheduled later than itself, which is clearly impossible. This indicates that the II is too small and must be increased until no diagonal entry is positive. On the other hand, if all the diagonal entries are negative, it indicates that there is slack around every recurrence circuit and that the II is larger than it need be. Since we are interested in finding the minimum legal II, at least one of the diagonal entries should be equal to 0. The smallest value of II, for which no diagonal entry is positive and at least one is zero, is the RecMII.

ComputeMinDist begins by initializing MinDist[i, j] with the minimum permissible time interval between i and j considering only the edges from i to j. If there is no such edge, MinDist[i, j] is initialized to $-\infty$. If e is an edge from i to j and if Distance(e) is zero, then edge e specifies that MinDist[i, j] be at least Distance(e). If, however, Distance(e) = d > 0, then the interval between the operation i from one iteration and the operation j from d iterations later must be at least Distance(e). Since the operation j from the same iteration as i is scheduled d*II cycles earlier, MinDist[i, j] must be at least as large as Distance(e) - d*II. Once MinDist has been initialized, the minimal cost-to-time ratio cycle algorithm is used to compute MinDist.

Since the algorithm ComputeMinDist is $O(N^3)$ and expensive for large values of N, the number of operations in the loop, it is desirable that it be invoked at few times as possible. In a production compiler, since one is interested not in the RecMII but only in the MII, the initial trial value of II should be the ResMII. If this yields no positive diagonal entry in the MinDist matrix, then it is the MII. Otherwise, the candidate MII is incremented until there are no positive entries on the diagonal. The value of the increment is doubled each time the MII is incremented. The candidate MII at this point is greater than or equal to the RecMII. A binary search is performed between this last, successful candidate MII and the previous unsuccessful value until the RecMII is found.

[1] Such dependences are often referred to as loop-carried dependences.

[2] An elementary circuit in a graph is a path through the graph which starts and ends at the same vertex (operation) and which does not visit any vertex on the circuit more than once.

```
procedure ModuloSchedule (BudgetRatio: real);

{  BudgetRatio is the ratio of the maximum number of   }
{  operation scheduling steps attempted (before giving }
{  up and trying a larger initiation interval) to the  }
{  number of operations in the loop.                   }

begin

{  Initialize the value of II to the }
{  Minimum Initiation Interval       }

   II := MII();

{  Perform iterative scheduling, first for II = MII and }
{  then for successively larger values of II, until all }
{  operations have been scheduled                       }

   Budget := BudgetRatio*NumberOfOperations;
   while (not IterativeSchedule(II, Budget)) do
         II := II + 1;

end;  { ModuloSchedule }
```

Figure 2. The procedure ModuloSchedule.

The statistics presented in Section 4 on the number of operations in a loop show that N can be quite large and, so, $O(N^3)$ complexity is a matter of some concern. This problem can be addressed by considering small subsets of the overall dependence graph when computing the RecMII. A **strongly connected component (SCC)** of a graph is a maximal set of vertices and the edges between them such that a path exists in the graph from every vertex to every other vertex. By definition, all the operations on a recurrence circuit must be part of the same SCC. The important observation is that the RecMII can be computed as the largest of the RecMII values for each individual SCC in the graph. As the statistics in Section 4 demonstrate, there are very few SCCs that are large, and $O(N^3)$ is quite a bit more tolerable for the small values of N encountered when N is the number of operations in a single SCC.

The same algorithm, ComputeMinDist can be used. The only difference is that it is fed the dependence graph for one SCC at a time rather than that for the entire loop. Each time ComputeMinDist is invoked with a new SCC, the initial starting value of the candidate MII is the resulting MII as computed with the previous SCC. For the first SCC, the initial value of MII is the ResMII.

3 Iterative modulo scheduling

3.1 The basic algorithm

Although a number of iterative algorithms and priority functions were investigated [33], simple extensions of the acyclic list scheduling algorithm and the commonly used height-based priority function proved to be near-best in schedule quality and near-best in computational complexity. The need for an iterative algorithm and the intuition underlying the selected heuristics are explained elsewhere [33]. The iterative modulo scheduling algorithm is shown in Figures 2-4. It assumes that two pseudo-operations, START and STOP, are added to the dependence graph. START and STOP are made to be the predecessor and successor, respectively, of all the other operations in the graph. Procedure ModuloSchedule calls IterativeSchedule with successively larger values of II, starting with an initial value equal to the MII, until the loop has been scheduled. IterativeSchedule looks very much like the conventional acyclic list scheduling algorithm. The points of difference are as follows.

- In view of the fact that an operation can be unscheduled and then rescheduled, operation scheduling, rather than instruction scheduling, is employed[1]. Also, the acyclic list scheduling notion that an operation becomes "ready" and may be scheduled only after its predecessors have been scheduled, has little value in iterative modulo scheduling since it is possible for a predecessor operation to be unscheduled after its successor has been scheduled.
- The function HighestPriorityOperation, which returns the unscheduled operation that has the highest priority in accordance with the priority scheme in use, may return the same operation multiple times if that operation has been unscheduled in the interim. This does not occur in acyclic list scheduling. The priority scheme used is discussed in Section 3.2.
- The calculation of Estart, the earliest start time for an operation as constrained by its dependences on its predecessors, is affected by the fact that operations can be unscheduled. When an operation is picked to be scheduled next, it is possible that one or more of its predecessors is no longer scheduled. Moreover, when scheduling the first operation in a SCC, it must necessarily be the case that at least one of its predecessors has not yet been scheduled. The formula for calculating Estart is discussed in Section 3.3.
- Adherence to the modulo constraint is facilitated by the use of a special version of the schedule reservation table [34]. If scheduling an operation at some particular time involves the use of resource R at time T, then location ((T mod II),R) of the table is used to record it. Consequently, the schedule reservation table need only be as long as the II. Such a reservation table has, subsequently, been named a **modulo reservation table (MRT)** [21].
- Since resource reservations are made on a MRT, a conflict at time T implies conflicts at all times $T \pm k*II$. So, it is sufficient to consider a contiguous set of candidate times that span an interval of II time slots. Therefore, MaxTime, which is the largest time slot that will be considered, is set to MinTime + II - 1, whereas in acyclic list scheduling it is effectively set to infinity.
- FindTimeSlot picks the time slot at which the currently selected operation will be scheduled. If MaxTime is infinite (and if a regular, linear schedule reservation table is employed), as it will be for acyclic scheduling, the functioning of FindTime Slot is just as it would be for list scheduling; the while-loop always exits having found a legal, conflict-free time slot. Since a MRT is used with modulo scheduling, MaxTime is at most (MinTime + II - 1). It is possible for the while-loop to terminate without having found a conflict-free time slot. At this point, it is clear that it is not possible to schedule the current operation without unscheduling one or more operations. The method for selecting which operations to unschedule is discussed in Section 3.4.

[1] Instruction scheduling operates by picking a current time and scheduling as many operations as possible at that time before moving on to the next time slot. In contrast, operation scheduling picks an operation and schedules it at whatever time slot is both legal and most desirable. Either style of scheduling can be used in iterative modulo scheduling, but the latter seems more natural.

```
function IterativeSchedule(II, Budget: integer): boolean;

{ Budget is the maximum number of operations scheduled }
{ before giving up and trying a larger initiation      }
{ interval. II is the current value of the initiation  }
{ interval for which modulo scheduling is being        }
{ attempted.                                           }

   var
      Operation, Estart:integer;
      MinTime, MaxTime, TimeSlot. integer;

begin
{  compute height-based priorities }
   HeightR;

{  schedule START operation at time 0 }
   schedule(START, 0);
   Budget := Budget - 1;

{  Mark all other operations as }
{  having never been scheduled  }
   for Operation := 2 to NumberOfOperations do
      NeverScheduled[Operation] := true;

{  Continue iterative scheduling until either all   }
{  operations have been scheduled, or the budget is }
{  exhausted.                                       }
   while (the list of unscheduled operations is not empty)
     and (Budget > 0) do
      begin
      { Pick the highest priority operation }
      { from the prioritized list           }
         Operation := HighestPriorityOperation();

      {  Estart is the earliest start time for }
      {  Operation as constrained by currently }
      {  scheduled predecessors                }
         Estart := CalculateEarlyStart(Operation);

         MinTime := Estart;
         MaxTime := MinTime + II - 1;

      {  Select time at which Operation }
      {  is to be scheduled             }
         TimeSlot := FindTimeSlot(Operation, MinTime,
                        MaxTime);

      {  The procedure Schedule schedules Operation at }
      {  time TimeSlot. In so doing, it displaces all  }
      {  previously scheduled nodes that conflict with }
      {  it either due to resource conflicts or        }
      {  dependence constraints. It also sets          }
      {  NeverScheduled[Operation] equal to false.     }

         Schedule(Operation, TimeSlot);
         Budget := Budget - 1;

      end; { while }

   IterativeSchedule := (the list of unscheduled
                         operations is empty);

end; { IterativeSchedule }
```

Figure 3. The function IterativeSchedule.

3.2 Computation of the scheduling priority

As is the case for acyclic list scheduling, there is a limitless number of priority functions that can be devised for modulo scheduling. Most of the ones used have been such as to give priority, one way or other, to operations that are on a recurrence circuit over those that are not [16, 21, 10]. This, to reflect that fact that it is more difficult to schedule such operations since all but the first one scheduled in a SCC are subject to a deadline. Instead, we shall use a priority function that is a direct extension of the height-based priority [17, 31] that is popular in acyclic list scheduling [1].

```
function FindTimeSlot(Operation, MinTime,
                      MaxTime: integer)  integer;

   var
      CurrTime, SchedSlot: integer;

begin
   CurrTime := MinTime;
   SchedSlot := null;
   while (SchedSlot = null) and (CurrTime <= MaxTime) do
      if ResourceConflict(Operation, CurrTime) then
      {  There is a resource conflict at    }
      {  CurrTime. Try the next time slot. }
         CurrTime := CurrTime + 1
      else
      {  There is no resource conflict at CurrTime. }
      {  Select this time slot.Note that dependence }
      {  conflicts with successor operations are    }
      {  ignored. Dependence constraints due to     }
      {  predecessor operations were honored in     }
      {  the computation of MinTime.                }
         SchedSlot := CurrTime;

{   If a legal slot was not found, then pick (in       }
{   decreasing order of priority) the first available  }
{   option from the following .                        }
{                                                      }
{   - MinTime, either if this is the first time that   }
{     Operation is being scheduled, or if MinTime is   }
{     greater than PrevScheduleTime[Operation], (where }
{     PrevScheduleTime[Operation] is the time at which }
{     Operation was last scheduled)                    }
{   - PrevScheduleTime[Operation] + 1                  }

   if SchedSlot = null then
      if (NeverScheduled[Operation]) or
         (MinTime > PrevScheduleTime[Operation]) then
         SchedSlot := MinTime
      else
         SchedSlot := PrevScheduleTime[Operation] + 1;

   FindTimeSlot := SchedSlot;

end; { FindTimeSlot }
```

Figure 4. The function FindTimeSlot.

Extending the height-based priority function for use in iterative modulo scheduling requires that we take into account inter-iteration dependences. Consider a successor Q of operation P with a dependence edge from P to Q having a distance of D. Assume that the operation Q that is in the same iteration as P has a height-based priority of H. Now, P's successor Q is actually D iterations later, and the STOP pseudo-operation D iterations later is II*D cycles later than the STOP pseudo-operation in the same iteration So, the height-based priority of successor Q is effectively H-II*D. The priority function used for iterative modulo scheduling, HeightR( ), is obtained by solving the system of implicit equations in Figure 5a.

If the MinDist matrix for the entire dependence graph has been computed, HeightR(P) is directly available as MinDist[P, STOP]. A less costly procedure is to iteratively solve the above implicit set of equations for HeightR(). An algorithm that is based on that for identifying the SCCs of a graph during a depth-first traversal of the graph [2] was employed This algorithm is described elsewhere [33].

$$\text{HeightR}(P) = \begin{cases} 0, & \text{if P is the STOP pseudo-op,} \\ \max_{Q \in \text{Succ}(P)} (\text{HeightR}(Q) + \text{Delay}(P,Q) - II*\text{Distance}(P,Q)), & \text{otherwise.} \end{cases}$$

(a)

$$\text{Estart}(P) = \max_{Q \in \text{Pred}(P)} \left(\begin{cases} 0, & \text{if Q is unscheduled} \\ \text{Max}(0, \text{SchedTime}(Q) + \text{Delay}(Q,P) - II*\text{Distance}(Q,P)), & \text{otherwise} \end{cases} \right)$$

(b)

Figure 5. (a) The equation for the height-based priority. (b) The equation for Estart.

HeightR( ) has a couple of good properties. As we shall see in Section 4, a large fraction of the loops are quite simple in their structure. For such loops there is a very good chance of scheduling them in one pass, but only if the operations are scheduled in topological sort order. HeightR( ) ensures this. Second, HeightR( ) gives higher priority to operations in those SCCs which have less slack. This makes HeightR( ) an effective heuristic in loops which have multiple, non-trivial SCCs.

3.3 Calculation of the range of candidate time slots

The MRT enforces correct schedules from a resource usage viewpoint. Correctness, from the viewpoint of dependence constraints imposed by predecessors, is taken care of by computing and using Estart, the earliest time that the operation in question may be scheduled while honoring the its dependences on its predecessors. In the context of recurrences and iterative modulo scheduling, it is impossible to guarantee that all of an operation's predecessors have been scheduled, and have remained scheduled, when the time comes to schedule the operation in question. So, Estart is calculated considering only those immediate predecessors that are currently scheduled. The early start time for operation P is given by the equation in Figure 5b, where Pred(P) is the set of immediate predecessors of P and SchedTime(Q) is the time at which Q has been scheduled.

Dependences with predecessor operations are honored by not scheduling an operation before its Estart. Dependences with successors operations are honored by virtue of the fact that when an operation is scheduled, all operations that conflict with it, either because of resource usage or due to dependence conflicts, are unscheduled. When these operations are scheduled subsequently, and Estart is computed for them, the dependence constraints are observed. At any point in time, the partial schedule for the currently scheduled operations fully honors all constraints between these scheduled operations.

It is pointless and redundant to consider more than II contiguous time slots starting with Estart. If a legal time slot is not found in this range because of resource conflicts, it will not be found outside this range. Therefore, MaxTime is set equal to Estart + II - 1.

3.4 Selection of operations to be unscheduled

Assume that a time slot is found, between MinTime and MaxTime, that does not result in a resource conflict with any currently scheduled operation. The only operations that will need to be unscheduled are those immediate successors with whom there is a dependence conflict. However, no operation need be unscheduled because of a resource conflict.

On the other hand, if every time slot from MinTime to MaxTime results in a resource conflict then we must make two decisions. First, we must choose a time slot in which to schedule the current operation and, second, we must choose which currently scheduled operations to displace from the schedule. The first decision is made with an eye to ensuring forward progress; in the event that the current operation was previously scheduled, it will not be rescheduled at the same time. This avoids a situation where two operations keep displacing each other endlessly from the schedule. If Estart is less than the previous schedule time, the operation is scheduled at Estart. If not, it is scheduled one cycle later than it was scheduled previously.

Regardless of which time slot we choose to schedule the operation, one or more operations will have to be unscheduled because of resource conflicts. In the event that there are multiple alternatives for scheduling an operation the choice of alternative determines which operations are unscheduled. Ideally, we would like to select that alternative which displaces the lowest priority operations. Instead of attempting to make this determination directly, all operations are unscheduled which conflict with the use of any of the alternatives. The current operation is then scheduled using one of the alternatives. The displaced operations will then be rescheduled, perhaps at the very same time, in the order specified by the priority function.

4 Experimental results

4.1 The experimental setup

The experimental input to the research scheduler was obtained from the Perfect Club benchmark suite [6], the Spec benchmarks [43] and the Livermore Fortran Kernels (LFK) [27] using the Fortran77 compiler for the Cydra 5. The Cydra 5 compiler examines every innermost loop as a potential candidate for modulo scheduling. Candidate loops are rejected if they are not DO-loops, if they can exit early, if they contain procedure calls, or if they contain more than 30 basic blocks prior to IF-conversion [10]. For those loops that would have been modulo scheduled by the Cydra 5 compiler, the intermediate representation, just prior to modulo scheduling but after load-store elimination, recurrence back-substitution and IF-conversion, was written out to a file that was then read in by the research scheduler. The input set to the research scheduler consisted of 1327 loops (1002 from the Perfect Club, 298 from Spec, and 27 from the LFK).

In the Cydra 5, 64-bit precision arithmetic was implemented on its 32-bit data paths by using each stage of the pipelines for two consecutive cycles. This results in a large number of block and complex reservation tables which, while they amplify the need

for iterative scheduling, are unrepresentative of future microprocessors with 64-bit datapaths. A compiler switch was used to force all computation into 32-bit precision so that, from the scheduler's point of view, the computation and the reservation tables better reflect a machine with 64-bit datapaths. The scheduling experiments were performed using the detailed, precise reservation tables for the Cydra 5 as well as the actual latencies (Table 2). The one exception is the load latency which was assumed to be 20 cycles rather than the 26 cycles that the Cydra 5 compiler uses for modulo scheduled loops.

Table 2. Relevant details of the machine model used by the scheduler in these experiments.

Functional Unit	Number	Operations	Latency
Memory port	2	Load Store Predicate set/reset	20 1 2
Address ALU	2	Address add / subtract	3
Adder	1	Integer/FLP add/subtract	4
Multiplier	1	Integer/FLP multiply Integer/FLP divide FLP square-root	5 22 26
Instruction pipeline	1	Branch	3

4.2 Program statistics

Presented in Table 3 are various statistics on the nature of the loops in the benchmarks utilized. The first column lists the measurement,, the second column lists the minimum value that the measurement can possibly yield, and the remaining columns provide various aspects of the distribution statistics for the quantity measured. The third column lists the frequency with which the minimum possible value was encountered, the fourth and the fifth columns specify the median and the mean of the distribution, respectively, and the last column indicates the maximum value that was encountered for that measurement.

As can be seen from Table 3, the number of operations per loop is generally quite small but there is at least one loop which has 163 operations. The fact that the median is less than the mean indicates a distribution that is heavily skewed towards small values, but having a long tail. The MII behaves in much the same way, as does the lower bound on the length of the modulo schedule for a single iteration of the loop. The lower bound on the modulo schedule length for a given II is the larger of MinDist[START, STOP] and the actual schedule length achieved by acyclic list scheduling. The large number of small loops appears to be due to the presence in the benchmarks of a large number of initialization loops.

Examining the distribution statistics in Table 3 for the quantity Max(0, RecMII-ResMII) we find an even more pronounced skew towards small values (mean = 4.54, maximum = 115). What is noteworthy is that for 84% of all loops this value is 0, for 90% it is less than or equal to 20, and for 95% it is less than or equal to 28. This has implications for the average computational complexity of the MII calculation; 84% of the time the RecMII is equal to or less than the ResMII and ComputeMinDist need only be invoked once per SCC in the loop.

A non-trivial SCC is one containing more than one operation. From a scheduling perspective, an operation from a trivial SCC need be treated no differently than one which is not in an SCC as long as the II is greater than or equal to the RecMII implied by the reflexive dependence edge. A loop can be more difficult to schedule if the number of non-trivial SCCs in it is large. Statistically, there tend to be very few SCCs per loop. In fact, 77% of the loops, the vectorizable ones, have no non-trivial SCCs. These statistics affect the average complexity of computing the MII.

Table 3. Distribution statistics for various measurements.

Measurement	Minimum Possible Value	Frequency of Minimum Possible Value	Median	Mean	Maximum Value
Number of operations	4	0.004	12.00	19.54	163.00
MII	1	0.286	3.00	11.41	163.00
Minimum Modulo Schedule Length	4	0.045	31.00	35.79	211.00
max(0, RecMII - ResMII)	0	0.840	0.00	4.54	115.00
Number of non-trivial SCCs	0	0.773	0.00	0.32	6.00
Number of nodes per SCC	1	0.930	1.00	1.30	42.00
II - MII	0	0.960	0.00	0.10	20.00
II / MII	1	0.960	1.00	1.01	1.50
Schedule Length (ratio)	1	0.484	1.02	1.07	2.03
Execution Time (ratio)	1	0.539	1.00	1.05	1.50
Number of nodes scheduled (ratio)	1	0.900	1.00	1.03	4.33

The number of operations per SCC plays a role in determining the average computational complexity of computing the RecMII and the MII. The distribution is heavily skewed towards small values. 93% of all SCCs consist of a single operation (typically, the add that increments the value of an address into an array), 95% have 2 operations or less and 99% consist of 8 operations or less. These statistics, along with those for the distribution of the difference between RecMII and ResMII, suggest that the complexity of calculating the RecMII may be expected to be small even though ComputeMinDist is $O(N^3)$ in complexity. The analysis in Section 4.4 bears this out.

4.3 Characterization of iterative modulo scheduling

The total time spent executing a given loop (possibly over multiple visits to the loop) is given by

EntryFreq*SL + (LoopFreq-EntryFreq)*II

where EntryFreq is the number of times the loop is entered, LoopFreq is the number of times the loop body is traversed, and SL is the schedule length for one iteration. The first two quantities are obtained by profiling the benchmark programs. This formula for execution time assumes that no time is spent in processor stalls due to cache faults or other causes. Except in the case of loops with very small trip counts, the coefficient of II is far larger than that of SL, and the execution time is determined primarily by the value of II. Consequently, II is the primary metric of schedule quality and SL is the secondary metric.

Let DeltaII refer to the difference between the achieved II and the MII. Table 3 shows that for 96% of all loops the lower bound of MII is achieved. Of the 1327 loops scheduled, 32 had a DeltaII of 1, 8 had a DeltaII of 2, and 11 had a DeltaII that was greater than 2. Of these, all but two had a DeltaII of 6 or less, and those two had a DeltaII of 20. Iterative modulo scheduling is quite successful in achieving optimal values of II. (It is worth noting that MII is not necessarily an achievable lower bound on II. The difference of the achieved II from the true, but unknown, minimum possible II may be even less than that indicated by these statistics.) These statistics also have implications for the average computational complexity of iterative modulo scheduling since the number of candidate MII values considered is proportional to $\log_2$(DeltaII).

These statistics also indicate that it is not beneficial to evaluate HeightR() symbolically, as a function of II, as is suggested by Lam for computing Estart [21]. In either case, symbolic computation is more expensive than a numerical computation. The advantage of the symbolic computation is that the re-evaluation of HeightR(), when the II is increased, is far less expensive than recalculating it numerically. However, the statistics on DeltaII show that this benefit would be derived for only 4% of the loops, whereas the higher cost of symbolic evaluation would be incurred on all the loops.

A somewhat more meaningful measure of schedule quality is the ratio of the achieved II to MII, i.e., the relative non-optimality of the II over the lower bound. The distribution statistics for this metric are shown in Table 3. Again, 96% of the loops have no degradation, 99% have a ratio of 1.1 or less, and the maximum ratio is 1.5.

The secondary measure of schedule quality is the length of the schedule for one iteration. The distribution statistics for the ratio of the achieved schedule length to the lower bound described earlier are shown in Table 3. For all but 5 loops, this ratio is no more than 1.5. (Note that this lower bound, too, is not necessarily achievable.)

In the final analysis, the best measure of schedule quality is the execution time which is computed by using the above formula. By using the lower bounds for SL and II in that formula, a lower bound on the execution time is obtained. Only 597 of the 1327 loops end up being executed for the input data sets used to profile the benchmark programs. Only these loops were considered when gathering execution time statistics. The distribution statistics for the ratio of the actual execution time to the lower bound are shown in Table 3 54% of the loops achieved the lower bound on execution time. All the loops together would only take 2.8% longer to execute than the lower bound. (Again, it is worth noting that we are comparing the actual execution time to a lower bound that is not necessarily achievable.)

Code quality must be balanced against the computational effort involved in generating a modulo schedule. It is reasonable to view the computational complexity of acyclic list scheduling as a lower bound on that for modulo scheduling, and it was a goal, when selecting the scheduling heuristics, to approach this lower bound in terms of the number of operation scheduling steps required and the computational cost of each step. In particular, since each operation is scheduled precisely once in acyclic list scheduling, this is the goal for modulo scheduling as well. At the same time, the modulo scheduling algorithm must be capable of coping with the complications caused by the presence of recurrences as well as block and complex reservation tables [33]. Consequently, this goal might not quite be achievable.

The last row in Table 3 provides some statstics on the scheduling inefficiency, i.e., the number of times an operation is scheduled as a ratio of the number of operations in the loop, *given that the II corresponds to the smallest value for which a schedule was found.* Under these circumstances, iterative modulo scheduling is quite efficient. For 90% of the loops, each operation is scheduled precisely once, the average value of the ratio is 1.03 and the largest value is 4.33. These statistics speak to the efficiency of the function IterativeSchedule. When considering the efficiency of the procedure ModuloSchedule, one must also take into account the scheduling effort expended for the unsuccessful values of II.

In procedure ModuloSchedule, the parameter BudgetRatio determines how hard IterativeSchedule tries to find a schedule for a candidate II before giving up. BudgetRatio multiplied by the number of operations in the loop is the value of the parameter Budget in IterativeSchedule. Budget is the limit on the number of operation scheduling steps performed before giving up on that candidate II.

In collecting the experimental data reported on above, BudgetRatio was set at 6, well above the largest value actually needed by any loop (which was 4.33). This was done in order to understand how well modulo scheduling can perform, in the best case, in terms of schedule quality. However, this large a BudgetRatio might not be the best choice. Generally, in order to find a schedule for a smaller value of II one must use a larger BudgetRatio. Too small a BudgetRatio results in having to try successively larger values of II until a schedule is found at a larger II than necessary. Not only does this yield a poorer schedule, but it also increases the computational complexity since a larger number of candidate values of II are attempted, and IterativeSchedule, on all but the last, successful invocation, expends its entire budget each time.

On the other hand, once the BudgetRatio has been increased enough that the minimum achievable II has been reached,

further increasing BudgetRatio cannot be beneficial in terms of schedule quality. However, it can increase the computational complexity if the minimum achievable II is larger than the MII. In this case, a certain number of unsuccessful values of II must necessarily be attempted. Increasing BudgetRatio only means that more compile time is spent on attempts that are destined to be unsuccessful. This suggests the possibility that there is some optimum value of BudgetRatio for which the execution time is near optimal and the computational complexity is also near its minimal value.

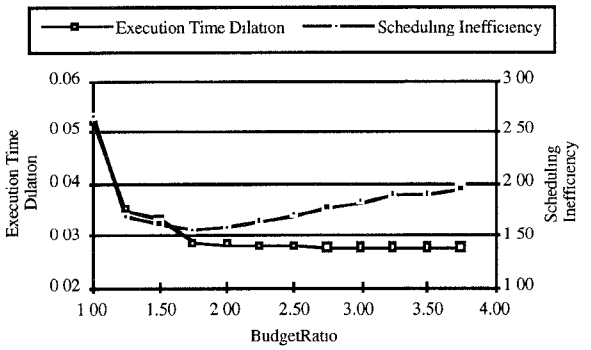


Figure 6. Variation of execution time and scheduling cost with the parameter BudgetRatio.

Figure 6 shows the dilation in the aggregate execution time over all the loops (as a fraction of the lower bound) and the aggregate scheduling inefficiency as a function of the BudgetRatio. The aggregate scheduling inefficiency is the ratio of the total number of operation scheduling steps performed in IterativeSchedule, across the entire set of loops, to the total number of operations in all the loops. Ideally, the scheduling inefficiency would be 1 and the execution time dilation would be 0. For each loop, a feasible II was found by performing a sequential search starting with II equal to MII. As surmised, execution time dilation decreases monotonically with BudgetRatio from 5.2%, down to 2.9% at a BudgetRatio of 1.75, and more gradually thereafter. The scheduling inefficiency, however, first decreases from 2.65 down to 1.55 at a BudgetRatio of 1.75 and then begins to increase slowly.

At a BudgetRatio of around 2, both the execution time dilation (2.8%) and the scheduling inefficiency (1.59) are down very close to their respective minimum values. If the set of benchmark loops used is viewed as representative of the actual workload, a BudgetRatio of 2 would be the optimum value to use when performing modulo scheduling for a processor with the machine model used in this study. (For the Perfect, Spec and LFK benchmarks individually, the optimum values for BudgetRatio are 2, 1.75 and 1.5, respectively.) If either the workload or the machine model are substantially different, a similar experiment would need to be conducted to ascertain the optimum value for BudgetRatio.

We see that we have come reasonably close to our goal of getting near-optimal performance at the same expense as acyclic list scheduling. Using a BudgetRatio of 2, we schedule on the average 1.59 operations per operation in the loop body. This means that, on the average, 0.59 operations are unscheduled for every operation in the loop. Although the cost of unscheduling an operation is less than the cost of scheduling it, we can, conservatively, assume that they are equal. So, the cost of iterative modulo scheduling is 2.18 (i.e., 1.59 + 0.59) times that of acyclic list scheduling, since the latter schedules each operation precisely once, and no operations are ever unscheduled.

This data enables an interesting comparison with "unroll-before-scheduling" schemes which rely on unrolling the body of the original loop prior to scheduling [13, 19, 23] and the "unroll-while-scheduling" schemes which unroll concurrently with scheduling [14, 7, 3]. To be competitive with iterative modulo scheduling, those schemes would need to get within 2.8% of the (possibly unachievable) lower bound on execution time without unrolling the loop body to more than 2.18 times its original size. In fact, "unroll-before-scheduling" schemes typically unroll the loop body many tens of times [23], leading to a computational complexity far greater than that of iterative modulo scheduling. Furthermore, the "unroll-while-scheduling" algorithms have the task of looking for a repeated scheduling state at every step. In the context of non-unit latencies and non-trivial reservation tables, this can be very expensive. Unfortunately, the complexity of such approaches having never been characterized makes a direct comparison with iterative modulo scheduling impossible.

4.4 Computational complexity of iterative modulo scheduling

We now examine the statistical computational complexity of iterative modulo scheduling as a function of the number of operations, N, in the loop. Iterative modulo scheduling involves a number of steps, the complexity of each of which is listed in Table 4. First, the SCCs must be identified. This can be done in O(N+E) time, where E is the number of edges in the dependence graph [2]. Although, in general, E is $O(N^2)$, for the dependence graphs under consideration here, one might expect that the in-degree of each vertex is not a function of N and that E is O(N). One can use linear regression to perform a least mean square error fit to the data with a polynomial in N. The best fit polynomial for E is given by 3.0036N. On the average there are about three edges in the graph per operation. This is higher than what one might expect because of the additional predicate input that each operation possesses. Since E is O(N), so is the complexity of identifying the SCCs.

Table 4. Computational complexity of various sub-activities involved in iterative modulo scheduling.

Activity	Worst-case computational complexity	Empirical computational complexity
SCC identification	O(N+E)	O(N)
ResMII calculation	O(N)	O(N)
MII calculation	-	O(N)
HeightR calculation	O(NE)	O(N)
Iterative scheduling	NP-complete	$O(N^2)$

The ResMII calculation first sorts the operations in increasing order of the number of alternatives, and then inspects the resource usage of each alternative for each operation exactly once. The complexity of the first step is O(N) using radix sort and so is that of the second step, since the number of alternatives per operation is not a function of N.

The computational complexity of the RecMII calculation is a function of the number of non-trivial SCCs in the loop, the number of operations in each SCC and the extent by which the RecMII is larger than the ResMII. It is difficult to characterize the worst-case complexity of this computation as a function of N since one might expect many of the above factors to be uncorrelated with N. This is borne out by the measured data. The empirical complexity obtained via a curve-fit is given by 11.9133N + 3.0474.

This is the expected number of times the innermost loop of ComputeMinDist is executed for a loop with N operations. However, the standard deviation of the residual error is 1842.7 which is larger than the predicted value over the measured range of N. In other words, the computational complexityvariable that is largely uncorrelated with N. To the extent that it is correlated, the empirical computational complexity of the MII calculation is linear in N.

The worst-case complexity of the algorithm for computing HeightR() is O(NE). The LMS curve-fit to the data shows that the expected number of times that the innermost loop of this algorithm is executed is given by 4.5021N. Empirically, the complexity of computing the scheduling priority is O(N).

The iterative modulo scheduling, itself, spends its time in two innermost loops. First, for each operation scheduled, all its immediate predecessors must be examined to calculate Estart. The expected number of times that this loop is executed, as a function of N, is 3.3321N. Second, for each operation scheduled, the loop in FindTimeSlot examines at most II time slots. The expected number of times this loop is executed is given by $0.0587N^2 + 0.2001N + 0.5000$.

Although the worst-case complexity of iterative scheduling is exponential in N, the empirical computational complexity of iterative scheduling is $O(N^2)$. From Table 4 we conclude that the statistical complexity of iterative modulo scheduling is $O(N^2)$ since no sub-activity is worse than $O(N^2)$.

5 Conclusion

In this paper we have presented an algorithm for modulo scheduling: iterative modulo scheduling. We have also presented a relatively simple priority function, HeightR( ) for use by the modulo scheduler. Our experimental findings are that iterative modulo scheduling, using the HeightR( ) scheduling priority function, and when assigned a BudgetRatio of 2

- requires the scheduling of only 59% more operations than does acyclic list scheduling,
- generates schedules that are optimal in II for 96% of the loops, and
- results in a near-optimal aggregate execution time for all the loops combined that is only 2.8% larger than the lower bound.

Iterative modulo scheduling generates near-optimal schedules. Furthermore, despite the iterative nature of this algorithm, it is quite economical in the amount of effort expended to achieve these near-optimal schedules. In particular, it is far more efficient than any cyclic or acyclic scheduling algorithm for loop scheduling which makes use of unrolling or code replication. If such algorithms replicate more than 118% of the loop body (which is just over one copy of the loop body) they will be more expensive computationally.

Modulo scheduling is a style of software pipelining which can provide very good cyclic schedules for innermost loops while keeping down the size of the resulting code. Along with IF-conversion, profile-based hyperblock selection, reverse IF-conversion, speculative code motion and modulo variable expansion, modulo scheduling can generate extremely good code for a wide class of loops (DO-loops, WHILE-loops and loops with early exits, with loop bodies that are arbitrary, acyclic control flow graphs, and dependences that result in the presence of data and control recurrences) for machines with or without predicates and with or without rotating registers.

Acknowledgements

This paper, and the underlying research, have benefited from the ongoing discussions with and suggestions from Vinod Kathail, Mike Schlansker and Sadun Anik. Vinod added to the Cydra 5 compiler the capability to write out the intermediate representation of software pipelineable loops for use as input to the research scheduler. The constructive comments, of one of the anonymous reviewers, were particularly helpful.

References

1. Adam, T.L., Chandy, K.M., and Dickson, J.R. A comparison of list schedules for parallel processing systems. *Communications of the ACM 17*, 12 (December 1974), 685-690.
2. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.
3. Aiken, A., and Nicolau, A. A realistic resource-constrained software pipelining algorithm. In *Advances in Languages and Compilers for Parallel Processing*, Nicolau, A., Gelernter, D., Gross, T., and Padua, D., (Editor). Pitman/The MIT Press, London, 1991, 274-290.
4. Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. Conversion of control dependence to data dependence. In *Proc. Tenth Annual ACM Symposium on Principles of Programming Languages*, (January 1983), 177-189.
5. Beck, G.R., Yen, D.W.L., and Anderson, T.L. The Cydra 5 mini-supercomputer: architecture and implementation. *The Journal of Supercomputing 7*, 1/2 (May 1993), 143-180.
6. Berry, M., Chen, D., Kuck, D., Lo, S., Pang, Y., Pointer, L., Roloff, R., Samah, A., Clementi, E., Chin, S., Schneider, D., Fox, G., Messina, P., Walker, D., Hsiung, C., Schwarzmeier, J., Lue, L., Orszag, S., Seidl, F., Johnson, O., Goodrum, R., and Martin, J. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications 3*, 3 (Fall 1989), 5-40.
7. Bodin, F., and Charot, F. Loop optimization for horizontal microcoded machines. In *Proc. 1990 International Conference on Supercomputing*, (Amsterdam, 1990), 164-176.
8. Charlesworth, A.E. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 Family. *Computer 14*, 9 (1981), 18-27.
9. Davidson, E.S., Shar, L.E., Thomas, A.T., and Patel, J.H. Effective control for pipelined computers. In *Proc. COMPCON '90*, (San Francisco, February 1975), 181-184.
10. Dehnert, J.C., and Towle, R.A. Compiling for the Cydra 5. *The Journal of Supercomputing 7*, 1/2 (May 1993), 181-228.
11. Ebcioglu, K. A compilation technique for software pipelining of loops with conditional jumps. In *Proc. 20th Annual Workshop on Microprogramming*, (Colorado Springs, Colorado, December 1987), 69-79.
12. Ebcioglu, K., and Nakatani, T. A new compilation technique for parallelizing loops with unpredictable

branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing*, Gelernter, D., Nicolau, A., and Padua, D., (Editor). Pitman/The MIT Press, London, 1989, 213-229.

13. Fisher, J.A. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers C-30*, 7 (July 1981), 478-490.
14. Fisher, J.A., Landskov, D., and Shriver, B.D. Microcode compaction: looking backward and looking forward. In *Proc. 1981 National Computer Conference*, (1981), 95-102.
15. Gasperoni, F., and Schwiegelshohn, U. Scheduling loops on parallel processors: a simple algorithm with close to optimum performance. In *Proc. International Conference CONPAR '92*, (1992), 625-636.
16. Hsu, P.Y.T. Highly Concurrent Scalar Processing. Ph.D thesis, University of Illinois, Urbana-Champaign, 1986.
17. Hu, T.C. Parallel sequencing and assembly line problems *Operations Research 9*, 6 (1961), 841-848.
18. Huff, R.A. Lifetime-sensitive modulo scheduling. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, (Albuquerque, New Mexico, June 1993), 258-267.
19. Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing 7*, 1/2 (May 1993), 229-248.
20. Jain, S. Circular scheduling: a new technique to perform software pipelining. In *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, (June 1991), 219-228.
21. Lam, M. Software pipelining: an effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, (June 1988), 318-327.
22. Lawler, E.L. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.
23. Lowney, P G., Freudenberger, S.M., Karzes, T.J., Lichtenstein, W.D., Nix, R.P., O'Donnell, J.S., and Ruttenberg, J.C. The Multiflow trace scheduling compiler. *The Journal of Supercomputing 7*, 1/2 (May 1993), 51-142.
24. Mahlke, S.A., Chen, W.Y., Bringmann, R.A., Hank, R.E., Hwu, W.W., Rau, B.R., and Schlansker, M.S. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems 11*, 4 (November 1993), 376-408.
25. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., and Bringmann, R.A. Effective compiler support for predicated execution using the hyperblock. In *Proc. 25th Annual International Symposium on Microarchitecture*, (1992), 45-54.
26. Mateti, P., and Deo, N. On algorithms for enumerating all circuits of a graph. *SIAM Journal of Computing 5*, 1 (1976), 90-99.
27. McMahon, F.H. The Livermore Fortran kernels: a computer test of the numerical performance range. Technical Report UCRL-53745. Lawrence Livermore National Laboratory, Livermore, California, 1986.
28. Moon, S.-M., and Ebcioglu, K. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proc. 25th Annual International Symposium on Microarchitecture*, (Portland, Oregon, December 1992).
29. Park, J.C.H., and Schlansker, M.S. On predicated execution. Technical Report HPL-91-58. Hewlett Packard Laboratories, 1991.
30. Ramakrishnan, S. Software pipelining in PA-RISC compilers. *Hewlett-Packard Journal*, (July 1992), 39-45.
31. Ramamoorthy, C.V., Chandy, K.M., and Gonzalez, M.J. Optimal scheduling strategies in a multiprocessor system. *IEEE Transactions on Computers C-21*, 2 (February 1972), 137-146.
32. Rau, B.R. Data flow and dependence analysis for instruction level parallelism. In *Fourth International Workshop on Languages and Compilers for Parallel Computing*, Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., (Editor). Springer-Verlag, . 1992, 236-250.
33. Rau, B.R. Iterative Modulo Scheduling. HPL Technical Report. Hewlett-Packard Laboratories, 1994.
34. Rau, B.R., and Glaeser, C.D. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. Fourteenth Annual Workshop on Microprogramming*, (October 1981), 183-198.
35. Rau, B.R., Lee, M., Tirumalai, P., and Schlansker, M.S. Register allocation for software pipelined loops. In *Proc. SIGPLAN'92 Conference on Programming Language Design and Implementation*, (San Francisco, June 17-19 1992).
36. Rau, B.R., Schlansker, M.S., and Tirumalai, P.P. Code generation schemas for modulo scheduled loops. In *Proc. 25th Annual International Symposium on Microarchitecture*, (Portland, Oregon, December 1992), 158-169.
37. Rau, B.R., Yen, D.W.L., Yen, W., and Towle, R.A. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. *Computer 22*, 1 (January 1989), 12-35.
38. Schlansker, M., and Kathail, V. Acceleration of first and higher order recurrences on processors with instruction level parallelism. In *Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, (Portland, Oregon, August 1993).
39. Su, B., and Wang, J. GURPR*: a new global software pipelining algorithm. In *Proc. 24th Annual International Symposium on Microarchitecture*, (Albuquerque, New Mexico, November 1991), 212-216.
40. Tiernan, J.C. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM 13*, (1970), 722-726.
41. Tirumalai, P., Lee, M., and Schlansker, M.S. Parallelization of loops with exits on pipelined architectures. In *Proc. Supercomputing '90*, (November 1990), 200-212.
42. Tokoro, M., Takizuka, T., Tamura, E., and Yamaura, I. A technique of global optimization of microprograms. In *Proc. 11th Annual Workshop on Microprogramming*, (Asilomar, California, November 1978), 41-50.
43. Uniejewski, J. SPEC Benchmark Suite: Designed for Today's Advanced Systems. *SPEC Newsletter 1*, 1 (Fall 1989).
44. Van Dongen, V., Gao, G.R., and Ning, Q. A polynomial time method for optimal software pipelining. In *Proc. International Conference CONPAR '92*, (1992).
45. Warter, N.J., Lavery, D.M., and Hwu, W.W. The benefit of predicated execution for software pipelining. In *Proc. 26th Annual Hawaii International Conference on System Sciences*, (Hawaii, 1993).
46. Warter, N.J., Mahlke, S.A., Hwu, W.W., and Rau, B.R. Reverse if-conversion. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, (Albuquerque, New Mexico, June 1993), 290-299.