Iterative Modulo Scheduling

B. Ramakrishna Rau
Compiler and Architecture Research
HPL-94-115
November, 1995

modulo scheduling,
instruction scheduling,
software pipelining,
loop scheduling

Modulo scheduling is a framework within which algorithms for the software pipelining of innermost loops may be defined. The framework specifies a set of constraints that must be met in order to achieve a legal modulo schedule. A wide variety of algorithms and heuristics can be defined within this framework. Little work has been done to evaluate and compare alternative algorithms and heuristics for modulo scheduling from the viewpoints of schedule quality as well as computational complexity. This, along with a vague and unfounded perception that modulo scheduling is computationally expensive as well as difficult to implement, have inhibited its incorporation into product compilers. This report presents iterative modulo scheduling, a practical algorithm that is capable of dealing with realistic machine models. The report also characterizes the algorithm in terms of the quality of the generated schedules as well the computational expense incurred.

Published in *The International Journal of Parallel Processing, Volume 24, Number 1, February 1996.* An earlier version of this report was published in the *Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, California, November 1994.*

1 Introduction

It is well known that there is inadequate instruction-level parallelism (ILP), as a rule, 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 [30, 45] and superblock scheduling [37], 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 backwards to previous iterations or forwards 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 [30, 45, 37]. 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 and scheduling effort.

Software pipelining [14] refers to a class of global cyclic scheduling algorithms which impose no such scheduling barrier[1]. 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 [22, 23, 39, 32, 51]. This is accompanied by the appropriate replication and unification of the operation as it moves past splits and merges in the control flow graph. For instance, if an operation is being moved down and around the backedge (i.e., to the next iteration), a copy of the operation must be made in the basic block following the loop as the operation moves past the split at the end of the loop body. Furthermore, since the operation is moved along the backedge and back into the top of the loop body, it must be matched by and unified with a copy that moves in from the basic block just before the loop. So that such an copy exists, the first iteration of the loop should have been peeled off. If an operation is moved down

[1] The original use of this term 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.

and around the backedge N times, the first N iterations of the loop should have been peeled off. Similar rules apply if an operation is moved speculatively up and around the backedge to a previous iteration.

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. At some point, further code motion around the back-edge will cease to improve the schedule and, in fact, will cause problems since the register pressure will increase. But it is unclear at what point further code motion becomes unproductive. Since we do not know what the optimal performance is, we could well be striving in vain for improvements that are just not possible, and we might have gone past the optimal point, resulting in reduced performance due to excessive code motion. The process is somewhat arbitrary and reminiscent of early attempts at global acyclic scheduling by ad hoc code motion between basic blocks [71]. 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 [51]. 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 although the Petri net approach [56, 4] and other finite-state automata approaches [52, 55, 7] constitute possible strategies for kernel recognition. 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 [61], a framework within which algorithms of this kind may be defined. 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[1] 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 [63]. Once the modulo schedule has been created, all the implied code motions and the complete structure of the code, including the placement and the target of the loop-closing branch, can all be determined. (This is discussed further in Section 2.3.)

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 [43] as is done with hyperblock scheduling [48, 47]. This defines a region that is to be modulo scheduled, and which, in general, has multiple back-edges and multiple exits.
- The early exit branches may either be retained or replaced by a pair of operations: one that sets the loop exit predicate to true, and one that sets a steering variable to a value that uniquely corresponds to this early exit. (By the correct use of the predicated execution capability, only those operations that should have executed, before exiting the loop, will execute.) After exiting the loop, a case statement on the steering variable guides flow of control to the appropriate exit target. In a similar manner, if there are multiple branches back to the loop header, they are coalesced by redirecting them to a single loop-closing branch [48, 47].

[1] 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.

- Within this selected region, memory reference data flow analysis and optimization are performed in order to eliminate partially redundant loads and stores as was done in the Cydra 5 compiler [59, 20] and in subsequent research investigations [13, 21]. This can improve the schedule either if 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 [5, 54, 20], with the result that all branches to targets within the region, except for the loop-closing branch disappear. With control flow converted to data dependences involving predicates [64, 40, 9], the region now looks either like a single basic block (if the early exit branches are eliminated) or like a superblock [48, 47]. In the event that the target architecture has special branch support for modulo scheduling, the loop-closing branch is replaced by either a **brtop** or **wtop** branch [64, 40, 9], depending on whether the loop is a DO-loop or WHILE-loop.
- Anti- and output dependences are minimized by putting the computation into the dynamic single assignment form [59]. (This is discussed further in Section 2.5.)
- If control dependences are the limiting factor in schedule performance, they may be selectively ignored thereby enabling speculative code motion [70, 46]. In the context of IF-converted code, this is termed predicate lifting; the predicate is replaced by a "larger" predicate, i.e., one that is true whenever the original one is true, but which also is true sometimes when the original one is not.
- Critical path reduction of data and control dependences may be employed to further reduce critical path lengths [66, 20, 67].
- Next, the lower bound on the initiation interval is computed. This is termed the minimum initiation interval (MII).
- If the MII 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. The extent of pre-unrolling can be computed quite easily so as to get the percentage degradation down below some acceptable threshold. For a given non-integer MII, f, the degree of pre-unroll, U, is determined by computing the smallest value of U such that the sub-optimality of the effective II, $\lceil f*U \rceil$, given by the expression $(\lceil f*U \rceil / (f*U) - 1)$, is less than some threshold of tolerable degradation (for instance, 0.05).
- At this point, the actual modulo scheduling is performed.

- The register requirements of the modulo schedule may be reduced by performing stage scheduling [27] which moves operations by integer multiples of II so as to reduce lifetime lengths.
- The predicates of operations whose predicates were lifted prior to scheduling are now made as small as possible, consistent with the schedule, in order to avoid unnecessary speculation which can result in increased register pressure. When possible the predicate is set back to its original value.
- If rotating registers [64, 9] are absent, the kernel (i.e., the new loop body after modulo scheduling has been performed) is unrolled to enable modulo variable expansion [42].
- 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 [63]. If there are early exit branches in the body of the loop, the corresponding epilogues must be crafted appropriately [43]. 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 [62] (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 [64, 40, 9], reverse IF-conversion [77] is employed to regenerate control flow.

Ideally, the processor provides architectural support in the form of predicated execution, suppression of spurious exceptions due to speculative execution, rotating registers and special branch opcodes. These and other capabilities useful for achieving high levels of ILP have been gathered together in HPL PlayDoh, an architecture designed to facilitate research in ILP [40].

The subject of this report 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 [61], at least two product compilers have incorporated modulo scheduling algorithms [57, 20], and any number of research papers have been written on this topic [34, 42, 70, 75, 76, 36], 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 either to thoroughly evaluate individual algorithms and heuristics for modulo scheduling, or to compare competing alternatives, from the viewpoints of schedule quality and computational complexity. The IMPACT group at the University of Illinois [11, 76] has compared Cydrome's approach to modulo scheduling [20] with the GURPR* algorithm [68] and with hierarchical scheduling [42]. Allan and her co-workers [4] have compared iterative modulo scheduling, as described in this report and previously [60], with the Petri Net approach to the unroll-while-schedule strategy [56].

This report makes a contribution 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 report does not attempt to provide a comparison of the algorithm described here to other alternative approaches for software pipelining. Also, the goal of this report 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 report is organized as follows. Section 2 provides an introduction to the modulo scheduling framework. Section 3 discusses the algorithms used to compute the lower bound on the initiation interval. Section 4 describes the iterative modulo scheduling algorithm in detail. Section 5 presents experimental data on the quality of the modulo schedules and on the computational complexity of the algorithms used. Section 6 discusses other algorithms for modulo scheduling and Section 7 states the conclusions.

2 Modulo scheduling

2.1 The dependence graph

In general, each operation in the body of the loop is dependent upon one or more operations. These dependences may either be data dependences (flow, anti- or output) or control dependences [29, 78]. 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 to an operation from the operation upon which it is dependent. There may be multiple edges, possibly with opposite directions, between the same pair of vertices.

Table 1: Formulae for calculating the delays on data dependence edges.

Type of dependence	**Delay**	**Conservative Delay**
Flow dependence	Latency(predecessor)	Latency(predecessor)
Anti-dependence	1-Latency(successor)	0
Output dependence	1+Latency(predecessor)-Latency(successor)	Latency(predecessor)

Each edge possesses an 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 an archetypal 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.

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. 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. The dependence distance is specified as a second attribute on the edge.

The existence of a recurrence manifests itself as a circuit in the dependence graph. The sum of the distances around the circuit must be strictly positive. A **strongly connected component (SCC)** of the dependence graph is a maximal set of vertices and the edges between them such that within this sub-graph a path exists from every vertex to every other vertex. Since every operation on a recurrence circuit is reachable from any other operation on that circuit, they must necessarily all be part of the same SCC.

2.2 The modulo scheduling framework

If we knew the exact trip count, i.e., the actual number of iterations of the loop when executed, we could achieve close to the best possible performance by unrolling the loop completely, treating the resulting code as a single basic block and generating the best possible schedule. This is not a practical strategy since we do not generally know the trip count at compile-time and, even if we did, this completely unrolled code would usually be unacceptably large[2]. Instead, imagine the following *conceptual* strategy. First, we unroll the loop completely. Then we schedule the code but with two constraints:

- all iterations have identical schedules, except that
- each iteration is scheduled to start some fixed number of cycles later than the previous iteration.

Assume that a legal schedule is obtained, somehow, that meets the above two constraints. The fixed delay between the start of successive iterations is termed the **initiation interval (II)**. (We

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

[2] However, if the trip count is a small compile-time constant, as often is the case, for instance, in graphics applications, complete unrolling of the loop can be a worthwhile strategy.

shall return to the questions of how the II and the schedule are actually determined.) Because of the constraints above, the schedule for the unrolled code is repetitive except for a portion at the beginning and a portion at the end.

This repetitive portion can be re-rolled to yield a new loop which is termed the **kernel**. The **prologue** is the code that precedes the repetitive part and the **epilogue** is the code following the repetitive part. By executing the prologue, then the kernel an appropriate number of times, and finally the epilogue, one would come close to re-creating the ideal, unconstrained schedule for the unrolled code. If this strategy is successful, a relatively small amount of code would be able to approximate the ideal (but impractical) strategy of unrolling the loop completely and scheduling it.

This approach to scheduling loops is modulo scheduling. The objective of modulo scheduling is to engineer a schedule for one iteration[1] of the loop such that when this same schedule is repeated at intervals of II cycles, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or distinct iterations. Since instances of the same operation from successive iterations are scheduled II cycles apart, legality of the schedule from a resource usage viewpoint is preserved by ensuring that, within a single iteration, no machine resource is used at two points in time that are separated by an interval which is a multiple of the II. In other words, within the schedule for a single iteration, the same resource is never used more than once at the same time modulo the II. We shall refer to this as the **modulo constraint** and it is from this constraint that the name modulo scheduling is derived. It is also the objective of modulo scheduling to generate a schedule having the smallest possible II, since this is the primary determinant of performance except in the case of very short trip counts.

2.3 Deducing the implied code motion from the modulo schedule

Let us consider more carefully how exactly the repetitive schedule is re-rolled conceptually and how the structure of the modulo scheduled code is determined. The schedule for a single iteration of the loop can be divided into **stages** consisting of II cycles each. The number of stages in one iteration is termed the **stage count** (SC). Likewise, the schedule for the conceptually unrolled loop, constrained in the manner described above, can also be divided into stages of II cycles. During each of the first SC-1 stages, a new iteration of the loop begins without the first one having

[1] As noted previously, due to pre-unrolling, a single iteration at this point may correspond to multiple iterations of the original loop.

as yet ended. From the SC-th stage on, one iteration completes for every one that starts until the last iteration has started. This is the repetitive part of the schedule which will be re-rolled into the kernel. Every iteration of the kernel performs one stage each (and a different one if SC > 1) from SC consecutive iterations of the original loop.

The schedule for the kernel is obtained from that for a single iteration of the loop by superimposing the schedules for each stage of the latter, i.e., by wrapping around the schedule for one iteration of the loop, modulo the II. Thus, the schedule for the kernel is exactly II cycles long. In the VLIW case, the code for the kernel will also be exactly II instructions long. Each VLIW instruction contains all of the operations that are scheduled in a single cycle of the kernel schedule. Note that these operations will, in general, come from different iterations of the loop.

The last cycle of the kernel coincides with the last cycle of each stage--one stage each from SC consecutive iterations. Assuming that the loop-closing branch has an n cycle latency, the branch *must* be scheduled in the n-th last cycle of the stage in which it is scheduled. This constitutes a scheduling contraint upon the branch operation if the stage boundaries within an iteration have already been determined. Alternatively, the branch operation may be freely scheduled and the stage boundaries are then uniquely determined. We shall adopt the former approach. In the event that the II is less than n, the kernel has less than n instructions and it becomes impossible to place the branch in the n-th last instruction. In such cases, the kernel must be unrolled enough so that the unrolled kernel has at least n instructions. In all cases, the target of the branch operation is the first instruction of the kernel.

By looking at the modulo schedule for a single iteration, we can decide what code motion is implicit in the schedule that we have adopted. The operations in the stage containing the branch operation are the only ones that have not been moved around the back-edge. All operations in earlier (later) stages have been moved backwards (forwards) around the back-edge to a previous (subsequent) iteration. The greater the distance of a stage from the one containing the branch, the larger is the number of times that the operations have been moved around the back-edge. The number of stages before (after) the one with the branch is equal to the number of iterations that were conceptually peeled off at the end (beginning) of the loop to permit these code motions across the back-edge. With this understanding, the code for the prologue and the epilogue(s) can be determined.

Thus, from the modulo schedule we get not only the desired schedule for the kernel but also the requisite amount of iteration peeling and code motion. The important point to note is that the requisite inter-block code motions are implicit in, and are determined by, the schedule which, in turn, is engineered so as to directly optimize a relevant figure-of-merit such as the II. This is to be contrasted with a philosophy in which operations are moved across the back-edge using ad hoc heuristics, in the hope that the resulting schedule will be near-optimal.

A detailed discussion of how the prologue, kernel and epilogue are structured, with and without predicated execution, with and without rotating registers, for DO-loops as well as for WHILE-loops and loops with early exits is discussed by Rau, et al. [63]. If there are early exit branches in the body of the loop, there are further subtleties in how the corresponding epilogues are crafted [43].

2.4 The minimum initiation interval (MII)

Modulo scheduling requires that a candidate II be selected before scheduling is attempted. 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 by analyzing 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

$$\text{MII} = \text{Max}\ (\text{ResMII}, \text{RecMII}).$$

Any legal II must be equal to or greater than the MII. As we shall see, in the face of recurrences and/or complex patterns of resource usage, the MII is not necessarily an achievable lower bound. The calculation of the ResMII and RecMII bounds are detailed in Section 3.

2.5 Dynamic Single Assignment

Anti- and output dependences can have an extremely detrimental effect upon the RecMII. In particular, every operation is anti-dependent, with a dependence distance of 1, upon all the operations that are flow dependent upon it. This is so since all of these flow dependent operations must be issued before their common source register is overwritten by the same operation from the next iteration. Together, these flow and anti-dependences can set up recurrence circuits in the data dependence graph which greatly increase the RecMII.

In acyclic (loop-free) code, one can eliminate anti-dependences via a program representation known as **static single assignment (SSA)** [29]. Very simply, this consists of never using the same virtual register as a destination more than once in the entire program. Since each virtual register has a result assigned to it exactly once, the problem of overwrites and the attendent anti- and output dependences cannot occur. A problem arises in the case of cyclic control flow graphs. Even after putting the code into the SSA form, we cannot avoid the fact that each time flow of control revisits a particular operation, the same virtual register is assigned a new value. Consequently, this operation must be made anti-dependent upon all operations that use the previous value in the destination register.

We eliminate these anti-dependences by enhancing the concept of a virtual register. An **expanded virtual register (EVR)** is an infinite, linearly ordered set of virtual registers with a special operation, **remap()**, defined upon it [59]. The elements of an EVR, v, can be addressed, read, and written as v[n], where n is any integer. For convenience, v[0] may be referred to as merely v. The effect of remap(v) is that whatever EVR element was accessible as v[n] prior to the remap operation will be accessible as v[n+1] after the remap operation. Except for registers holding loop invariants, all registers are implicitly remapped when the loop-closing branch is taken. Although textually it might appear that register v is being assigned to repeatedly, dynamically it is still single assignment; a different element of the EVR is assigned to on each occasion.

A program representation is in the **dynamic single assignment (DSA)** form [59] if the same virtual register (EVR element) is never assigned to more than once on any dynamic execution path. The static code may have multiple operations with the same virtual destination register as long as these operations are in mutually exclusive basic blocks or separated by (implicit) remap operations. In this form, a program has no anti- or output dependences due to register usage but may possess such dependences due to loads and stores to the same, or potentially the same, memory location.

Of course, as with conventional virtual registers, EVRs are a fiction that cannot be translated into hardware. The infinite number of virtual registers in each EVR, of which there are an unlimited number, must be mapped into a finite number of physical registers. In fact, only a finite, contiguous set of the elements of an EVR may be expected to be live at any point in time, and only these need to be allocated physical registers. This register allocation step will re-introduce anti-dependences, but it must be done in such a manner as to never introduce an anti-dependence that contradicts the schedule just created.

The rotating register file [64] is one way of directly providing hardware support for the EVR concept. The remapping is implemented by providing a circular register file with a pointer into it that is decremented each time a new iteration is started. Addressing into the register file is relative to this pointer. However, EVRs are of value in the intermediate representation even when there are no rotating registers provided in the hardware. The use of EVRs makes it possible to temporarily ignore anti- and output dependences and thereby engineer an optimal or near-optimal modulo schedule. Thereafter, the anti- and output dependences can be honored, without compromising the schedule by unrolling the kernel an appropriate number of times and performing modulo variable expansion, i.e., virtual register renaming [42]. In effect, remapping is simulated by code replication and register renaming. Register allocation for modulo scheduled loops is discussed in greater detail by Rau, et al. [62].

Although dynamic single assignment is generally better, it is worth noting that there are situations in which dynamic multiple assignment is preferable despite the anti-dependences that are introduced. If DSA is rigidly adhered to, copy operations must be introduced at the site of every ϕ-function [29], one for each predecessor block at the point of a control flow merge. (Alternatively, when there are two predecessor basic blocks, a select operation may be used, and if there are more than two predecessor blocks, a tree of selects must be used.)

These copy operations can increase the schedule length if they are on the critical path or if they require the use of a critical resource. The copy operations can be eliminated by renaming the virtual registers on the input side of the ϕ-function to be the same as the output. Often this will not introduce anti-dependences since the multiple static assignments are mutually exclusive. In those cases where an anti-dependence is introduced, its effect on the schedule length can either be better or worse than the effect of the copy operations. It is currently not well understood how this choice should be made.

The important point, however, is that EVRs make it possible to retain only those anti-dependences that are desirable from the viewpoint of schedule length. In the absence of EVRs, the loop may be unrolled and put into the static single assignment form. If the loop is unrolled adequately, the loop-carried anti-dependences will not degrade the schedule. Unfortunately, the extent of unrolling required is known precisely only *after* modulo scheduling is performed, whereas the unrolling, if performed, must be done *before* modulo scheduling.

2.6 Modelling and tracking resource usage

The resource usage of a particular opcode is specified as a list of the resources used by the operation, and the times at which each of those resources is used relative to the time of issue of the operation. 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. Figure 1a is a pictorial representation of the resource usage behavior 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 behavior of a multiply operation on the multplier pipeline. This method of modelling resource usage is termed a **reservation table** [15].

A particular operation (opcode) may be executable in multiple ways, with a different reservation table corresponding to each one of these ways. In this case, the operation is said to have multiple **alternatives**. Frequently, but not always, each alternative corresponds to the use of one of the multiple functional units on which this operation can be executed. In general, 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.

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 [72]. 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 (time 0). 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.

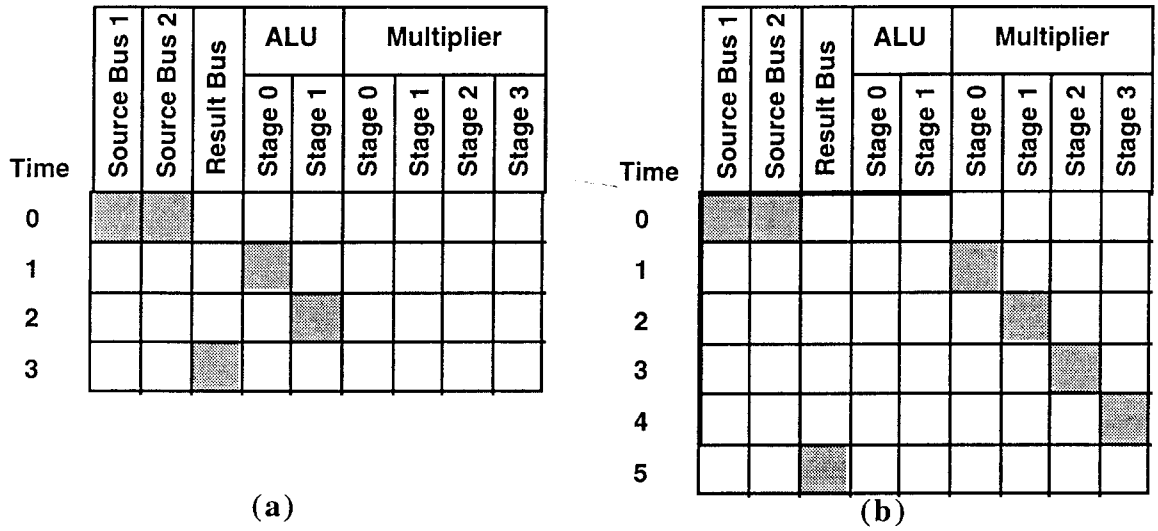

Figure 1. Reservation tables for (a) a pipelined add, and (b) a pipelined multiply.

From Figure 1, 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. The set of forbidden intervals, due to resource conflicts, between the initiation of two operations with the same reservation table can be collected together into what is termed the **collision vector** for that reservation table [15]. Likewise, the set of forbidden intervals, due to resource conflicts, between

the initiation of an operation with one reservation table and a second operation with another reservation table is termed the **cross-collision vector** for that pair of reservation tables.

Given the set of reservation tables for a particular machine, one can compute the complete set of collision and cross-collision vectors. In general, their are many sets of reservation tables that yield the same set of collision and cross-collision vectors. Some of these sets are smaller and simpler than others. It is possible to compute a synthetic set of reservations tables that are preferable to the original set [26]. For instance, in Figure 1, if the ALU and multiplier possessed their own, separate source and result buses and if all operations that used these two pipelines used precisely the same reservation tables, then both reservation tables could be replaced by simple reservation tables.

2.7 A simplistic modulo scheduling algorithm

Having eliminated all undesirable anti- and output dependences due to register usage and having computed the MII, the next step is to generate a modulo schedule. In the simplest case, a minor modification [61] of the commonly used, greedy, list scheduling[1] algorithm [1, 41] serves the purpose. The only change is that an operation P may not be scheduled at a time t such that if P were scheduled at time $t \pm k*II$ for any $k \geq 0$, it would have a resource conflict with a previously scheduled operation (from the same iteration). Normally in list scheduling, only the case $k = 0$ is considered.

An acyclic scheduler keeps track of resource usage by means of the schedule reservation table, which may be conceptualized as a table in which each column corresponds to a particular resource, and each row corresponds one cycle. If scheduling an operation at some particular time involves the use of resource R at time T, then entry [T,R] of the schedule reservation table is used to record that fact. In the case of modulo scheduling, adherence to the modulo constraint is facilitated by the use of a special version of the schedule reservation table [61]. If scheduling an operation at some particular time involves the use of resource R at time T, then entry [(T mod II),R] of the schedule reservation table is used to record it. Consequently, the schedule reservation table need only be as long as the II. Such a reservation table has been aptly dubbed a **modulo reservation table** (**MRT**) by Lam [42].

---

[1] List scheduling refers to the class of scheduling algorithms in which all of the operations are ordered into a list, once and for all, based on some priority function, and are then scheduled in the order specified by that list.

This minor variant of the list scheduling algorithm is, in fact, guaranteed to yield a legitimate schedule with an II equal to MII if the loop is such that the only recurrence circuits are trivial ones involving a single operation and if the machine is such that every operation has simple reservation tables. When either assumption is invalid, this scheduling algorithm can fail to find a schedule for an II that is equal to the MII.

2.8 Complicating factors in modulo scheduling

The existence of recurrences and complex reservation tables can, individually or jointly, result in the non-existence of a valid schedule at the MII. Furthermore, because of these two factors, a greedy, one-pass scheduling algorithm, such as list scheduling, cannot guarantee that it will find a modulo schedule for a given II, even when such a schedule exists.

2.8.1 Difficulty in finding a schedule for a feasible initiation interval

The new phenomenon that recurrences introduce to the scheduling process is that of a deadline, i.e., the requirement that an operation be scheduled *no later than* some particular time. When the first operation in a SCC is scheduled, it imposes constraints on how late all of its predecessor operations can finish and, therefore, on how late they can start. Since this operation is in a SCC, the rest of the operations in the SCC are its predecessors as well as its successors. These successors are, as yet, unscheduled (since the operation under consideration was the first one in the SCC to be scheduled) and all of them now have deadlines on the latest time by which they must be scheduled. From then on, the scheduling algorithm must be sensitive to these deadlines. If any one of these deadlines cannot be honored, the partial schedule generated thus far represents a dead end that cannot lead to a valid schedule.

In particular, if the first operation that is scheduled from a SCC is scheduled greedily, at the earliest possible time, the deadlines may be more constraining than they need have been, and a legal schedule may not be found. In such a case, it would have been preferable to have scheduled that operation somewhat later than the earliest possible time. Yet, as a rule, it is preferable to schedule operations as early as possible, in order to minimize the schedule length. This represents a dilemma for a greedy, one-pass scheduler.

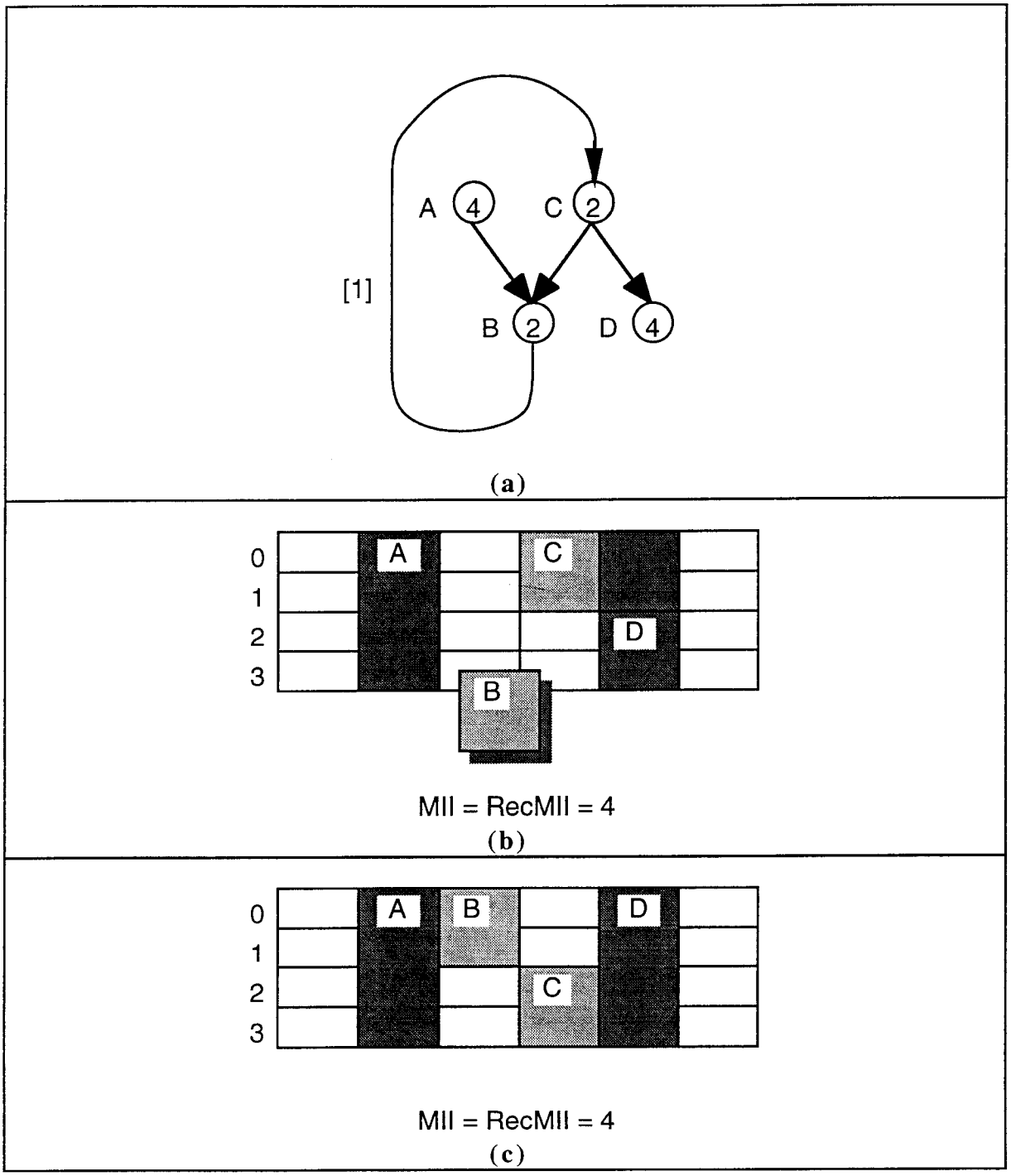

Figure 2. An example demonstrating the problem with greedy scheduling in the presence of recurrences. (a) The data dependence graph. (b) The resulting partial schedule when C is scheduled greedily. B cannot be scheduled. (c) The resulting valid schedule when C is scheduled two cycles later.

If list scheduling is employed and one gets into a situation wherein an operation cannot be scheduled by its deadline, scheduling cannot proceed any further; any resulting schedule is guaranteed to be invalid. In general, only an exhaustive, back-tracking search over the space of possible schedules is guaranteed to yield a valid schedule for the candidate II, if one in fact exists.

Consider the loop in Figure 2a. In this and the following examples, the label next to each vertex is its name, and the latency of each operation is indicated within the vertex. The delay on each edge is understood to be the latency of the source vertex, and the distance of each edge is understood to be zero unless it is explicitly shown in brackets. If list scheduling is employed on this example, the partial schedule in Figure 2b is obtained. The operation B cannot be scheduled since the earliest time at which it can be scheduled is 4 which is greater than the deadline imposed upon it by the fact that its successor operation C (from the next iteration) is scheduled at time 4 (4 cycles later than the C from the same iteration which is scheduled at time 0). Operation B must be scheduled at the latest by cycle 2 and this is impossible since A only finishes at the end of cycle 3.

The problem in this case is that operation C was scheduled at the earliest possible time, which generally is a good policy. In this case, if it had been scheduled 2 cycles later, at time 2, the deadline for operation B would have been extended to time 4 which would have made it possible to arrive at the valid schedule in Figure 2c. An exhaustive search would have yielded this schedule sooner or later. However, the computational complexity of such a process is exponential in the number of operations and, for all but the smallest loops, is not a practical approach.

Greedy scheduling can also lead to a dead-end as a result of the interaction of block or complex reservation tables. The example in Figure 3a consists of a chain of operations, A1 through M5, and one independent operation, M6. A1, A3 and A4 are non-pipelined adds that take two cycles each on the adder, M5 and M6 are non-pipelined multiply operations that take three cycles each on the multiplier, and C2 is a copy operation that uses the bus for one cycle.

The typical list scheduling algorithm would result in A1, M6, C2, and A3 being scheduled in that order yielding a partial schedule and an MRT state as shown in Figure 3b. A4 cannot be scheduled because the only free adder slots, at times 2 and 5 modulo 6, are not contiguous as required by the block reservation table for A4. If, however, A3 had been scheduled one cycle later that the earliest possible time, at time 4 as in Figure 3c, an acceptable schedule is obtained.

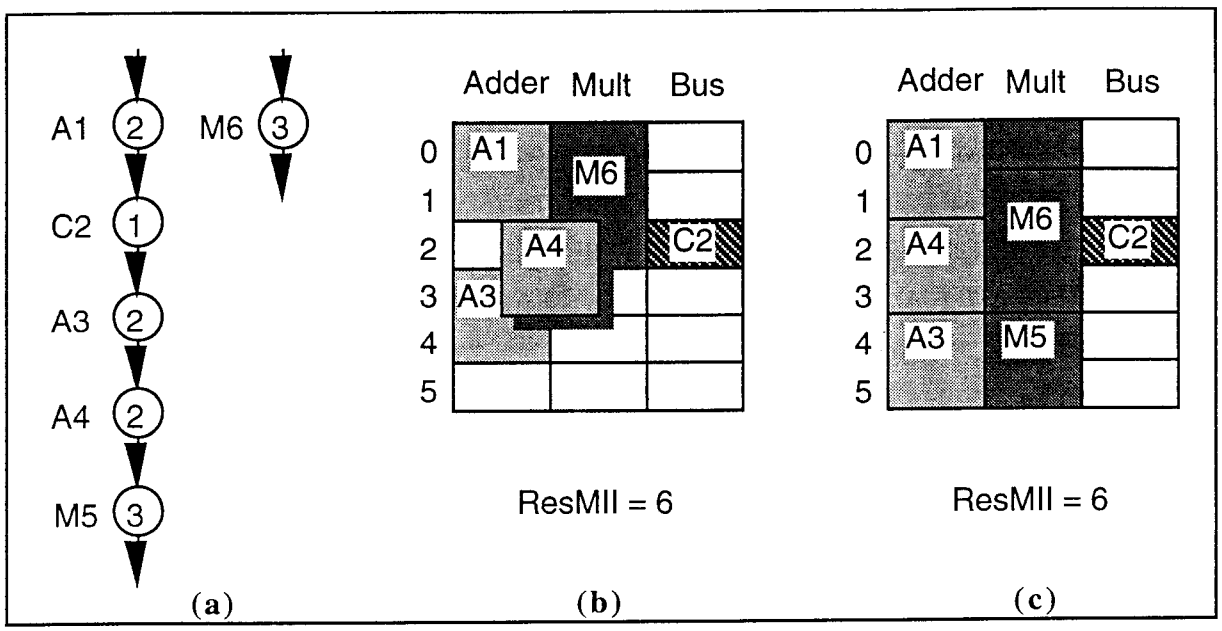

Figure 3. An example demonstrating the problem with greedy scheduling in the presence of block or complex reservation tables. (a) The data dependence graph. (b) The resulting partial schedule when A1, M6, C2, and A3 are scheduled greedily. A4 cannot be scheduled. (c) The resulting valid schedule when A3 is scheduled one cycle later.

These are problems that acyclic list scheduling does not have to cope with. As these examples demonstrate, a simple, modulo-constrained version of greedy list scheduling is inadequate. Although greedy scheduling is generally a good idea in order to minimize the schedule length, it can also lead to a partial schedule which is a dead end. Some form of backtracking is required, which yields a final schedule in which the appropriate operations have been scheduled in a non-greedy fashion. The specific form of backtracking adopted is discussed in Section 4.

2.8.2 Infeasibility of the minimum initiation interval

In the presence of block or complex reservation tables, or due to the existence of recurrence cycles, it can be the case that there is no valid schedule possible at the computed MII.

Consider the example of a loop consisting of the three independent operations shown in Figure 4a and their respective complex reservation tables. Between the three operations, each resource is used exactly twice, yielding a ResMII of 2. Figure 4b shows the MRT for an II of 2. After A1 and M2 have been scheduled, it is impossible to schedule MA3. A little reflection shows that the three

reservation tables are, in fact, incompatible for an II of 2, but are compatible for an II of 3 (Figure 4c).

A second example shows why a valid schedule may turn out to be impossible for II = MII, even in the absence of block or complex reservation tables, as a consequence of the contradictory constraints of the recurrence dependences and the (simple) reservation tables. Consider the loop (Figure 5a) consisting of four fully pipelined add operations, A1 through A4, each one dependent upon the previous one with a dependence distance of 0, and with A1 dependent upon A4 with a distance of 2. Assuming that the add latency is 2, this yields a RecMII of 4. Also, assuming each add uses each stage of the single pipelined adder for one cycle, the ResMII is equal to 4. Thus, the MII is 4.

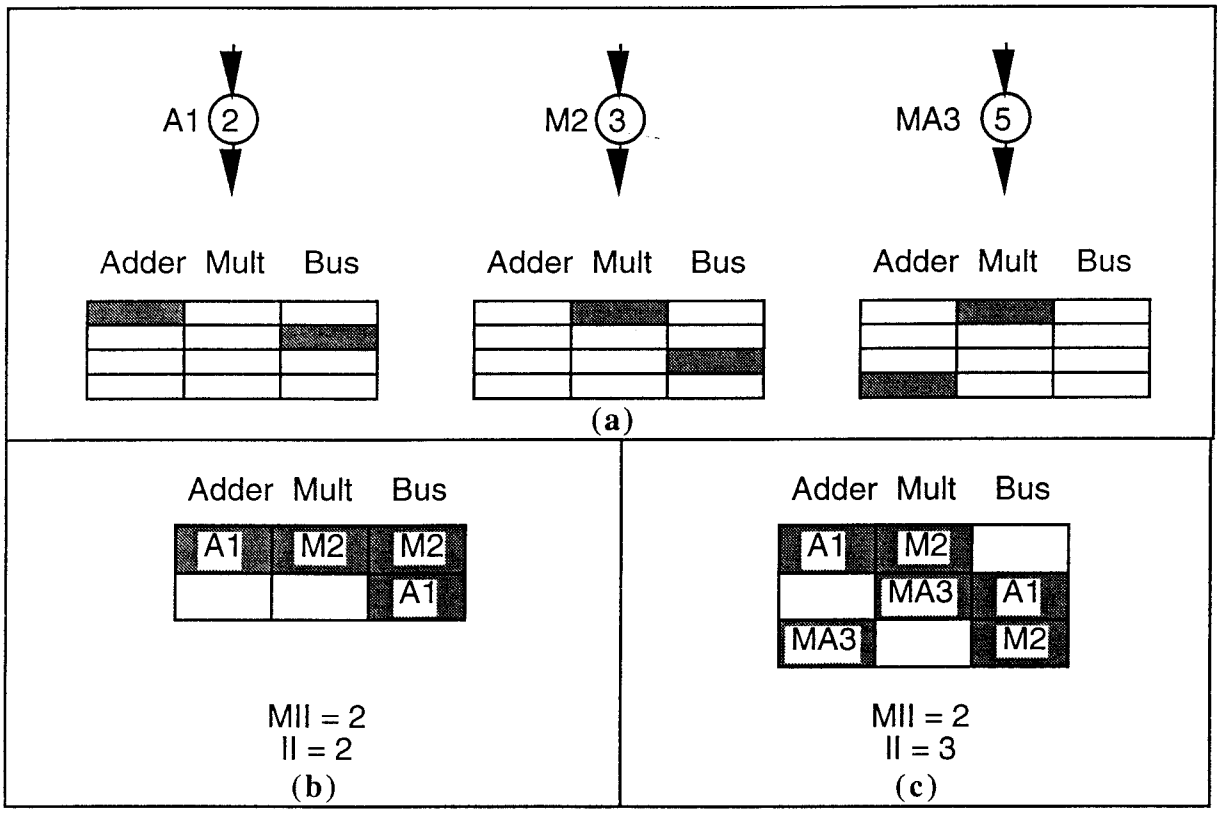

Figure 4. An example demonstrating the infeasibility of the MII in the presence of complex reservation tables. (a) The three operations and their reservation tables. (b) The MRT corresponding to the dead-end partial schedule for an II of 2 after A1 and M2 have been scheduled. (c) The MRT corresponding to a valid schedule for an II of 3.

However, a schedule cannot be obtained for II = MII = 4 (Figure 5b). As soon as A1 is scheduled at time 0, the latest start times for A2 through A4 are 2, 4 and 6, respectively, based on the latency of the add operation. These are also the earliest start times for those operations and, so, based on the dependence constraints, A2, A3 and A4 must be scheduled at cycles 2, 4 and 6, respectively. Unfortunately, 4 and 6 are equal to 0 and 2, respectively, modulo the II of 4, which means that A3 and A4 have resource conflicts with A1 and A2, respectively. The only schedule from a dependence viewpoint is unacceptable from a resource usage viewpoint. No valid schedule exists for II = 4.

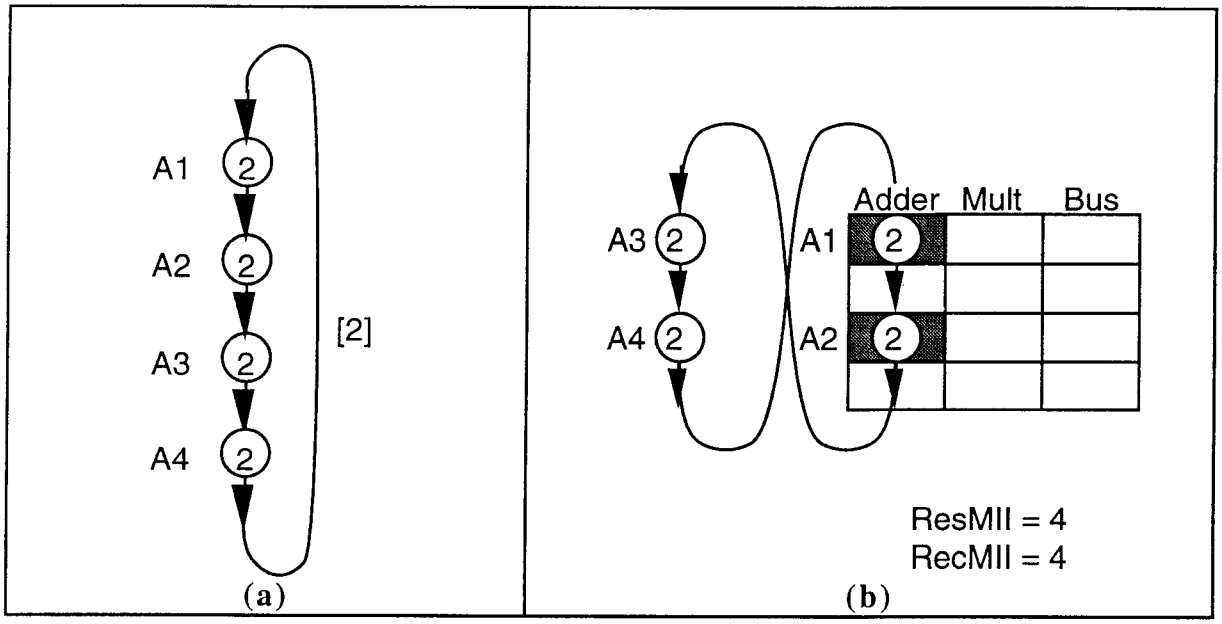

Figure 5. An example demonstrating the infeasibility of the MII due to the interaction between the recurrence constraints and the resource usage constraints. (a) The data dependence graph. (b) The MRT corresponding to the dead-end partial schedule for an II of 4 after A1 and A2 have been scheduled.

In such cases, an exhaustive search will eventually reveal the absence of a valid schedule by coming up empty-handed. However, the exponential amount of computational effort needed to discover this fact, is unacceptable. A reasonable strategy is to give up on a candidate II after expending a pre-allocated budget of time, and to try again at a larger value of II.

3 Calculation of the minimum initiation interval (MII)

The minimum initiation interval, MII, is the larger of the resource-constrained MII, ResMII, and the recurrence-constrained MII, RecMII. In general, it will be some non-integer, f. However, modulo scheduling requires an integer-valued II. If f is not an integer, and if the loop is not to be unrolled, an integer-valued MII is obtained by rounding the f up to the next larger integer. Alternatively, if the loop is to be pre-unrolled with an unroll factor of U, as discussed in Section 1, the corresponding MII is $\lceil f*U \rceil$. We now address the method by which the ResMII and the RecMII are computed.

3.1 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. The exact ResMII can be computed by performing an optimal bin-packing of the reservation tables for all the operations, a computation that 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 approximation to this exact lower bound is computed.

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, and initializing the usage count of each resource to zero. At each step in the algorithm, the partial ResMII is defined to be the usage count of the most heavily used resource at that point. The operations are processed in sort order. For an operation with multiple alternatives, each alternative is selected in turn. The number of times it uses each resource is added to the current usage count for that resource in order to determine the partial ResMII that would result if this alternative were actually used. For each operation, that alternative is selected which yields the lowest partial ResMII. When all operations have been considered, the usage count for the most heavily used resource constitutes the ResMII. It is important to note that this selection between the alternatives is only for the purpose of computing the ResMII. During scheduling, the scheduler has complete freedom to select the alternative that will actually be used.

The method above yields an integer-valued ResMII. In general, the ResMII is a real-valued number and ought to be computed as such. For instance, if the floating-point adders are the most heavily used functional units, there are three of them and there are 8 add operations, the ResMII is 8/3. More generally, if the functional units can be partitioned into equivalence classes, such that all of

the functional units in a given equivalence class can perform exactly the same set of operations, and if the operations are such that each one can be performed by precisely one equivalence class, then the ResMII can be computed by taking ratio of the number of operations to be executed on a given equivalence class and dividing that number by the number of functional units in that equivalence class. The largest such ratio, across all of the equivalence classes, is the ResMII.

Unfortunately, many real machines do not possess any such structure, and there exists no good technique for computing the real-valued ResMII. One approximate way to do so is to *conceptually* unroll the loop U times, calculate the integer-valued ResMII for this unrolled loop, and then divide that by U.

3.2 The recurrence-constrained MII (RecMII)

The RecMII is calculated using the dependence graph discussed in Section 2.1. 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 as discussed in Section 2.5.

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 existence of a recurrence manifests itself as a circuit in the dependence graph. Assume that the sum of the delays along some elementary circuit[1] 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, since the schedule for each iteration is the same as that for the previous iteration, except that it is delayed by II cycles, 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 in an irredundant fashion all the elementary circuits in the graph [69, 49], calculate the smallest value of

[1] 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.

II that satisfies the above inequality for each individual circuit, and to use the largest such value across all circuits. Thus,

$$\text{RecMII} = \max_{c \in C} \left(\frac{\text{Delay}(c)}{\text{Distance}(c)} \right).$$

Along with a few additional optimizations, this is the approach used in Cydrome's compiler for the Cydra 5 [20].

A second approach is to pose the task of computing the RecMII as a linear programming problem [16, 17] where the objective function to be minimized is the II and the constraints are that for each pair of operations, P and S, that are connected by an edge from P to S, the following inequality be satisfied:

$$T(S) + II*\text{Distance}(P,S) - T(P) \geq \text{Delay}(P,S)$$

where T(•) is the time at which the operation is scheduled.

The third approach, the one used in this study, is to pose the problem as a minimal cost-to-time ratio cycle problem [44] as proposed by Huff [36]. A version of this algorithm, which can be used to compute the RecMII, as well as for other purposes, is shown in Figure 6.

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, *from 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. In general, the RecMII computed in this way will not be an integer. If this is the case, and if only an integer-valued RecMII is of interest, the next larger integer is selected at the RecMII. In this case, no entry along the diagonal of the MinDist matrix will be 0.

```
function ComputeMinDist (II: integer; NumberOfOperations: integer): boolean;

{  Compute the longest path between every pair of vertices.  }
{                                                            }
{  Return TRUE if there is a positive cycle.                 }
{                                                            }
{  This function is repeatedly invoked to compute the RecMII }
{  as part of a binary search for the smallest value of II   }
{  for which there is no positive cycle.                     }

   label
      1;

   var
      i, j, k: integer;
      dist: integer;
      PositiveCycle: boolean;

begin

{  Initialize the distance matrix with the adjusted delay    }
{  between pairs of operations that have an arc between them.}
   for i := 1 to NumberOfOperations do
      for j := 1 to NumberOfOperations do
         MinDist[i, j] := -∞;
         for each edge e that goes from i to j do
            MinDist[i, j] := Max(MinDist[i, j], (Delay[e] - II * Distance[e]));

   PositiveCycle := false;
   for k := 1 to NumberOfOperations do
   {  Now consider all paths via vertex k as well }
      for i := 1 to NumberOfOperations do
         for j := 1 to NumberOfOperations do
            dist := MinDist[i, k] + MinDist[k, j];
            if dist > MinDist[i, j] then
               MinDist[i, j] := dist;
               if (i = j) and (dist > 0) then
                  PositiveCycle := true;
                  goto 1;
1:
   ComputeMinDist := PositiveCycle;

end; { ComputeMinDist }
```

Figure 6. ComputeMinDist: an algorithm for computing the minimum permissible time interval between every pair of operations.

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 -∞. 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 for that candidate II. If MinDist has a positive entry along the diagonal, the candidate II is smaller than the RecMII. If all diagonal entries are greater than zero, the candidate II is greater than the RecMII. Thus, by repeatedly invoking ComputeMinDist, we can determine the RecMII.

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 as few times as possible. If one is interested only in the MII, and not the RecMII, the initial trial value of II is 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, which is initially 1, 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 to the desired degree of floating-point precision. If only the integer MII is required, we start with the integer ResMII and only examine integer values for MII during the above binary search. (To compute the RecMII, the only difference is that the first candidate value tested is 1 and the last unsuccessful value tested is 0.) The number of times ComputeMinDist must be invoked for this strategy can be shown to be

$$\begin{cases} 1, & \text{if RecMII} \leq \text{ResMII}, \\ 2 * \lceil \log_2(1+\text{RecMII}-\text{ResMII}) \rceil + F, & \text{otherwise.} \end{cases}$$

where F is the number of bits of fractional precision desired.

The statistics presented in Section 5 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. Since every operation on a recurrence circuit is reachable from any other operation on that circuit, they must all 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 5 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.

4 Iterative modulo scheduling

Since a one-pass scheduling algorithm will often not yield a valid schedule, and since an exhaustive search is generally unacceptable, the only option is to use some form of non-exhaustive search through the space of schedules. Each state in this search space corresponds either to a complete schedule, in which every operation has been scheduled, or a partial schedule, in which only some of the operations have been scheduled. Starting from a state in which no operation is scheduled, the goal is to find a complete schedule that is either optimal or near-optimal.

Search strategies can be categorized by whether or not they are deterministic, goal-directed searches. Examples of algorithms, that search through the search space in a semi-random fashion, are simulated annealing, the Boltzmann machine algorithm and genetic algorithms [65] and these have been applied to the modulo scheduling problem [38, 8, 19]. Such algorithms do not run the risk of getting trapped in a local minimum but, in the context of a production compiler, take an unacceptably long time to find a near-optimal solution. In contrast, goal-directed search algorithms are guided by heuristics in a deterministic fashion through the search space towards, what one hopes, is a global optimum. Although good heuristics improve the probability that the optimum found is the global optimum, the possibility exists, nevertheless, that the optimum may well be only a local one.

One approach, guaranteed to find an optimal solution, is to pose the task of finding a modulo schedule as an integer linear programming problem [28, 6, 25]. Although linear programming has exponential complexity in the worst-case, this is generally not the case in practice. Even so, the execution time of integer linear programming algorithms is far too great for them to be seriously considered for use in production compilers. On the other hand, this approach is useful for evaluating the quality of heuristics-based scheduling algorithms.

In the case of modulo scheduling, the problem of finding a local optimum is not an important one. To a first order, any legal schedule that is found for the desired II, is as good as any other. The

bigger problem is that of finding oneself in a state, corresponding to a partial schedule, from which no forward progress is possible, i.e., no additional operation can be scheduled without violating some dependence or resource constraint. Goal-directed searches must have a strategy to deal with this situation.

One option is to backtrack some distance along the path in the search space that was taken to this dead-end and to then move forward again, branching off along a different path this time. Unless an exhaustive search is being performed, additional heuristics must be employed to determine how far to backtrack and which (different) path to take this time around. The task of selecting a better path to take from a given state on a subsequent attempt differs from the corresponding task on the first attempt in that one has knowledge of some number of failed attempts. Somehow, some information gathered from these failed attempts must guide the heuristic that selects a different path. Unfortunately, there are no known heuristics that are successful at exploiting such information. Indeed, it is not even understood what information is relevant to a better decision on a subsequent attempt.

Another option is to use a second heuristic to jump from the dead-end state corresponding to one partial schedule to some other state corresponding to another partial schedule. Thereafter, the first heuristic takes over once again in the search for a complete, feasible schedule. The hope is that a valid schedule will be found at the candidate II after searching only a small fraction of the entire space of schedules. Such a scheduling algorithm, if successful, would be more attractive than either a one-pass or an exhaustive-search algorithm. Of course, there is the possibility that such an algorithm, lacking the systematic search of the exhaustive algorithm, could either get locked into a repetitive orbit, circling around aimlessly through the space of partial schedules, never finding a valid one even though it has expended more effort than the exhaustive-search algorithm. It is crucial that good heuristics be employed to control both the search for a valid schedule and the choice of a next state when the search arrives at a dead-end. The iterative modulo scheduling algorithm, described in this report, is of this type.

It employs a deterministic, goal-directed search for a legal schedule at the candidate II. This bears a strong resemblance to the well-known list scheduling algorithm using height-based priorities. A second heuristic is used to jump from the dead-end state corresponding to one partial schedule to some other state correponding to another partial schedule. When the scheduler finds that there is no available slot for scheduling the currently selected operation, it displaces one or more previously scheduled, conflicting operations. These are then, in turn, re-scheduled when the deterministic,

goal-directed search for a legal schedule continues. This behavior, of repeatedly scheduling and re-scheduling operations, in search of a "fixed-point" solution which simultaneously satisfies all of the scheduling constraints, is why this algorithm for modulo scheduling is termed iterative.

If the search fails to yield a valid schedule even after a large number of steps, it is reasonable to assume that no feasible schedule exists at this II. The only option is to increase the candidate II by some amount and try again, in the belief that a schedule will be easier to find for a larger II. This seems reasonable from two viewpoints. First, a larger II results in a later deadline for every recurrence operation and, hence, more time slots in which that operation can be scheduled. In the extreme, once the II is equal to the schedule length of the (non-modulo) list schedule, even the one-pass list scheduling algorithm should work. Second, a larger II provides more usage time slots for every resource on the modulo reservation table, making it simpler to find a non-overlapping placement of the reservation tables for each operation.

4.1 An intuitive motivation for iterative modulo scheduling

Although a number of iterative algorithms and priority functions were investigated, a simple extension of the acyclic list scheduling algorithm and of the commonly used height-based priority function proved to be near-best in schedule quality and near-best in computational complexity.

To understand the intuition behind the algorithm, we resort to the view that we are, conceptually, scheduling the loop as we unroll it, with an eye to kernel recognition. What is different is that we shall be proactive in causing the repeating pattern, the kernel, to appear. We shall do so by enforcing two constraints upon the schedule that we create for the loop as we simultaneously unroll it.

- First, we shall insist that each iteration[1] begin exactly II cycles after the previous one.
- Second, each time we schedule an operation in one iteration, we shall tentatively schedule the same operation for subsequent iterations at intervals of II, even if it entails the unscheduling of other tentatively scheduled operations from subsequent iterations due to resource or dependence conflicts. However, when possible, each operation is scheduled so as to not unschedule any operation that was tentatively scheduled previously.

[1] Note that an iteration at this point might correspond to multiple original iterations if the loop was pre-unrolled as discussed in Section 1. In terms of the original iterations, this constraint states that there is a periodic sequence of initiation intervals such that the length of one period is II.

Starting iterations at fixed intervals of II is similar to limiting the "span" [3] and the use of the pacemaker function [56] which ensure that, in the steady state, the average rate of starting iterations is equal to the average rate at which they finish so that a repeating pattern will form. One difference is that this constraint imposes the steady state rate right from the outset, thereby expediting the formation of the kernel. A further difference is that, although we enforce a regular rate of starting iterations, and one that is equal to the rate at which they complete, we place no constraints upon the number of iterations that might be executing simultaneously. As a result, it permits the necessary amount of "slip" to develop between operations. For instance, if operation B is scheduled more than 3*II cycles later than operation A in the same iteration, then operation A is scheduled three times before B is scheduled for the first time.

The tentative scheduling of operations from subsequent iterations is the key difference with respect to other kernel recognition schemes. It provides guidance to the scheduler, when scheduling one iteration, as to what is desirable from the viewpoint of subsequent iterations in order to quickly achieve a repeating kernel. However, 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.

The result is that a repeating pattern, with a period of II, is enforced for the remainder of the schedule even at the expense of having only a partial schedule. If at any point, all operations from one iteration have been scheduled (without having to displace any tentatively scheduled operations from subsequent operations), then all operations from all subsequent iterations have, by construction, also been scheduled, and with a repetitive pattern. Thus, it is not necessary to explicitly check whether we have reached a previously visited state, i.e., kernel recognition becomes unnecessary. Retroactively, one can also reschedule operations from all previous iterations to conform to the schedule of the kernel. Now, because every iteration has the same schedule, staggered in time by II cycles, the repeating portion of the scheduled unrolled loop can be rolled up into a kernel with a loop around it, preceded by a prologue and followed by an epilogue.

Of course, in practice there is no need to actually unroll, schedule and then re-roll. The multiple iterations that unrolling algorithms schedule, are simulated in the case of modulo scheduling by repeatedly re-scheduling one II's worth of instructions on the MRT. One can view the iterative modulo scheduling process as consisting of sliding an II-long window down the code as it is unrolled and scheduled. In unroll-and-schedule terms, the MRT can be viewed as consisting the

last instruction of the schedule thus far (the current instruction) plus the next II-1 instructions that solely consist of tentatively scheduled operations. The rest of the schedule is tentatively the same as these last II instructions. The use of an MRT enforces both the constraint that a new iteration begin every II cycles and the constraint that all subsequent instances of an operation be scheduled at regular intervals of II. The iterative aspect of this modulo scheduling algorithm refers to the willingness, when necessary, to schedule an operation even at the expense of unscheduling a previously scheduled operation.

4.2 The core algorithm for iterative modulo scheduling

The iterative modulo scheduling algorithm is shown in Figures 7, 8, 10 and 11. 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 (Figure 7) calls IterativeSchedule (Figure 8) 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 [1, 41]. The points of difference are as follows.

- Acyclic list schedulers, typically, employ instruction scheduling. Although this is not necessary, it is more natural for various reasons. In view of the fact that with iterative modulo scheduling an operation can be unscheduled and then rescheduled, operation scheduling, rather than instruction scheduling, is employed[1]. Also, the acyclic instruction 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, HeightR, is discussed in Section 4.3.

---

[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.

```
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              }

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

end;  { ModuloSchedule }
```

Figure 7. The procedure ModuloSchedule.

- Estart is the earliest start time for an operation as constrained by its dependences on its predecessors. MinTime is the earliest time considered for scheduling the current operation. In general, MinTime is greater than or equal to Estart. However, just as with acyclic scheduling, we shall always set MinTime equal to Estart. The calculation of Estart 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 4.4.

- Adherence to the modulo constraint is facilitated by the use of a special version of the schedule reservation table [61] known as a modulo reservation table (MRT).

- 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 implicitly set to infinity. This is discussed in Section 4.5.

- FindTimeSlot picks the time slot at which the currently selected operation will be scheduled. If MaxTime is infinite (and if a traditional, 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 4.6.

4.3 Computation of the scheduling priority

As with 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 [34, 42, 20]. 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 [35, 58] that is popular in acyclic list scheduling [1].

For acyclic list scheduling, the height-based priority of an operation P, Height(P), is defined as

$$\text{Height}(P) = \begin{cases} 0, & \text{if P is the STOP pseudo-op,} \\ \underset{Q \in \text{Succ}(P)}{\text{Max}} \; (\text{Height}(Q) + \text{Delay}(P,Q)), & \text{otherwise.} \end{cases}$$

This priority function has two important properties. First, since it computes the longest path from P to the end of the graph (the STOP pseudo-operation), the larger Height(P) is, the smaller is the amount of slack available to schedule operation P. This means that the operation is more critical in that it can afford to experience less delay in scheduling if the schedule length is not to be increased. It is well known that giving priority to operations on the critical path is important to achieving a good schedule, and the height-based priority function does so.

```
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, MinTime, MaxTime, TimeSlot: integer;

begin
{  compute height-based priorities }
   HeightR;

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

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

   Insert all operations into the list of unscheduled operations;

{  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 operations 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 8. The function IterativeSchedule.

If minimizing schedule length were the only objective then we could emphasize the critical path more directly by first scheduling all operations on the critical path, i.e., with zero slack, then operations with a slack of one, and so on. This would result in certain operations being scheduled before all of their predecessors have been scheduled. When the time comes to schedule the predecessors, it might be impossible to do so without rescheduling some successors so as to make place in the schedule for them. Whereas this might result in a good schedule, it would be more expensive computationally since certain operations will be scheduled multiple times.

From the viewpoint of efficiency, it is preferable to schedule operations in some topological sort order so that each operation is scheduled before any of its successors. The second nice property of the height-based priority function is that it defines a topological sort; a predecessor operation will have a larger height-based priority than every one of its successors[1]. If we perform acyclic list scheduling by scheduling operations in decreasing order of their height-based priority, we get a scheduler which is efficient and critical path sensitive at one and the same time.

```
procedure Height (opn);

begin
   if priority[opn] = -∞ then
      if opn has no children then
         priority[opn] := 0
      else
         for each child do
            priority[opn] := max(priority[opn], Height[child] + Delay[opn,child]);
end; { Height }
```

Figure 9: Algorithm for computing Height

Figure 9 displays the algorithm for recursively computing the height-based priorities of all the operations in an acyclic dependence graph. The priority for each operation is initialized to -∞. Then Height is invoked with the START operation as the argument. This results in the recursive computation of the priority of each of the successors of the START operation, after which the priority of the START operation is computed in accordance with the above equation. The result of

[1] This is strictly true only if we ignore anti- and output dependences or if we utilize the conservative formulae for computing the delay. Since we assume that the computation is in the dynamic single assignment form, there are no anti- or output dependences to cause problems.

this function call is the length of the critical path through the computation. In the process, the dependence graph is traversed via a depth-first search (DFS) tree rooted in the START vertex. The priority of every operation is computed in a post-order manner, ensuring that the priorities of all children have already been computed. If an operation (and its DFS sub-tree) have already been visited, the priority of the operation will no longer be $-\infty$ and Height will not be called recursively on that operation.

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 (the current iteration) has a height-based priority of H. Since, P's successor Q is actually D iterations later, its height-based priority, relative to the current iteration, is effectively H-II*D. Once again, let

$$\text{EffDelay}(Q,P) = \text{Delay}(Q,P) - II*\text{Distance}(Q,P).$$

Then, the priority function used for iterative modulo scheduling, HeightR, is given by

$$\text{HeightR}(P) = \begin{cases} 0, & \text{if P is the STOP pseudo-op,} \\ \max\limits_{Q \in \text{Succ}(P)} (\text{HeightR}(Q) + \text{EffDelay}(P,Q)), & \text{otherwise.} \end{cases}$$

If the MinDist matrix for the entire dependence graph has been computed, HeightR(P) is directly available as MinDist[P, STOP]. A less costly approach is to solve the above implicit set of equations for HeightR, by seeking the smallest fixed-point solution, using a procedure similar to the one in Figure 9 above for computing Height. However, the direct application of this recursive algorithm would fail--due to the recurrence cycles in the dependence graph, this algorithm would go into infinite recursion. On the other hand, if each SCC is viewed as a super-vertex, the resulting graph is acyclic, and the height of each vertex can be computed during a depth-first walk of the graph. For normal vertices, the height is computed, as before, in a post-order fashion. For the super-vertices, the situation is a bit more complex.

The height of a vertex that is part of a SCC cannot be computed in the normal post-order fashion because, in general, at this point the rest of the vertices in the SCC will not yet have been visited,

and nor will all of its successors outside the SCC. Since in an SCC, every vertex is a successor of every other vertex, we cannot calculate the height of any vertex in the SCC until all of the successors of the SCC have been visited and had their heights computed.

In the course of the depth-first search, the first vertex of an SCC to be encountered constitutes the root of the smallest DFS sub-tree containing the entire SCC. We shall refer to this vertex as the root of the SCC. The DFS sub-tree rooted in this vertex must, necessarily, also contain all of the successors of the SCC. Consequently, the final, post-order visit to the root of the SCC is the earliest point in time at which the correct heights for all of of the vertices in the SCC can be computed. The correct heights for all of the successors of the SCC (including other SCCs, by using recursive reasoning) will have been computed at this point. The heights for the vertices of this SCC are computed by repeatedly computing them for each vertex in accordance with the above formula, until the smallest fixed-point solution is found.

The efficiency of this iterative, fixed-point solution process is improved by traversing the vertices of the SCC, repeatedly, in the order defined by the post-order traversal of the DFS tree. To facilitate this, the vertices of an SCC are collected on a stack during the DFS traversal of the dependence graph by pushing such vertices on to a stack in post-order fashion. At the time of the post-order visit to the root of an SCC, all of the vertices of the SCC will be on the stack, contiguous, and at the very top of the stack. When solving for the heights of these vertices iteratively, the preferred order of evaluation is specified by the stack. Note that the first iteration of the fixed-point process actually occurs during the DFS traversal, concurrent with the accumulation of the SCC vertices on the stack. Once the fixed-point solution has been obtained, the entire SCC is popped off the stack, and the DFS traversal of the dependence graph continues.

The algorithm for computing HeightR is shown in Figure 10. For each vertex, v, priority[v] is initialized to $-\infty$ and New[v] is initialized to true. New[v] indicates that this vertex has not as yet been encountered during the DFS traversal. The procedure HeightR is invoked with the START operation as the argument. This procedure functions much like procedure Height except in two respects. First, on encountering a vertex, v, for the first time, it sets New[v] to false to ensure that this DFS sub-tree is not traversed for a second time from some other parent of v. Second, it treats vertices that are part of an SCC differently by pushing them on to the stack in post-order fashion. Furthermore, if this is the root of the SCC, then the procedure FinalizeSCCheights is invoked to obtain the fixed-point solution for the heights of all of the vertices in the SCC, and then the entire SCC is popped of the stack.

```
procedure FinalizeSCCheights;

begin
   First := the deepest operation in the stack which is in the same SCC
            as the operation on the top of the stack;

   repeat until
      for opn = First to TopOfStack do
         for each child of opn do
            priority[opn] := Max(priority[opn], priority[child] +
                                 Delay[opn,child] - Dist[opn,child] * II);
   no operation's priority changed on the most recent iteration;
end; { FinalizeSCCheights }
```

```
procedure HeightR (opn: integer);

begin
   New[opn] := false;

   if opn has no children then
      priority[opn] := 0
   else
      for each child of opn do
         if New[child] then
            HeightR(child);
         priority[opn] := Max(priority[opn], priority[child] +
                              Delay[opn,child] - Dist[opn,child] * II);

   if opn is part of an SCC then
      Push(opn);

   if opn is the DFS root of an SCC then
      FinalizeSCCheights;
      while (the operation on the top of the stack is in the same SCC as opn) do
         Pop;

end; { HeightR }
```

Figure 10: Algorithm for computing HeightR

The underlying depth-first search is O(N+E) complexity where N is the number of vertices and E is the number of edges. For the types of graphs under consideration, E is O(N). The complexity of each call to FinalizeSCCheights is O(r(n+e)), where n is the number of vertices in the SCC, e is the number of edges, and r is the number of iterations over the vertices. In the worst case, r is O(n). In practice, a couple of iterations over the vertices yields a fixed-point solution. Thus, in practice, the computation of HeightR has a complexity that is linear in the number of vertices in the dependence graph (see the empirical measurements in Section 5.4).

HeightR has a couple of good properties. As we shall see in Section 5, 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. Therefore, HeightR is also an effective heuristic in loops which have multiple, non-trivial SCCs.

4.4 Calculation of earliest and latest times for scheduling

When performing acyclic list scheduling, the operations are scheduled in topological sort order, each operation is scheduled only after its predecessors have been scheduled, and the earliest time at which an operation P can be scheduled is given by

$$\text{Estart}(P) = \max_{Q \in \text{Pred}(P)} (\text{SchedTime}(Q) + \text{Delay}(Q,P))$$

where Pred(P) is the set of immediate predecessors of P, SchedTime(Q) is the time at which Q has been scheduled, and Delay(Q,P) is the minimum allowable time interval between the start of operation Q and the start of operation P.

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. That being the case, Estart can be calculated in one of two ways: either considering only the scheduled immediate predecessors of the operation, or by considering all scheduled predecessors. We shall refer to these two definitions of Estart as the immediate Estart and the transitive Estart, respectively. Let

$$\text{EffDelay}(Q,P) = \text{Delay}(Q,P) - II*\text{Distance}(Q,P).$$

Then, the immediate early start time for each unscheduled operation P is given by

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

where Estart(START) = SchedTime(START) = 0. The calculation of the immediate Estart is simple and inexpensive.

For the transitive early start time, we again have that Estart(START) = SchedTime(START) = 0. For every unscheduled operation, P, the transitive Estart is defined by

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

This set of equations may be iteratively solved by initializing Estart for all unscheduled operations to 0 and then repeatedly computing Estart for each of them, using the above formula, until a fixed point is reached. Statistically, this requires approximately 2.5 iterations over all the unscheduled operations. Although the transitive Estart would appear to be a more precise definition of early start time than is the immediate Estart, it is also far more expensive computationally. Each time an operation is scheduled, the above formula must be evaluated approximately 2.5N times, where N is the number of operations in the loop.

Alternatively, the transitive Estart may be computed using the MinDist matrix that was discussed in Section 3.2. (ComputeMinDist should have been given the entire dependence graph to operate on, not just the individual SCCs.) Once MinDist has been computed, the latter definition of Estart(P) can be computed as

$$\text{Estart}(P) = \max_{Q \in \text{SchedOp}} (\text{SchedTime}(Q) + \text{MinDist}(Q,P)),$$

where SchedOp is the set of operations that are currently scheduled. Although this approach reduces the cost of computing the transitive Estart, it does require that the MinDist matrix be computed for the loop as a whole, which is expensive both in time and space. Regardless of which approach is used to calculate the transitive Estart, the delivered benefits of using the transitive Estart instead of the immediate Estart need to be large to justify the computational complexity. In this study, only the immediate Estart was employed.

In acyclic list scheduling, it is never the case that a successor, of the operation that is being scheduled, has already been scheduled. As a result, there is no constraint on how late the operation can be scheduled. For iterative modulo scheduling, it is indeed possible that one or more successors have already been scheduled when a given operation is (re)scheduled, and a constraint exists on how late the operation may be scheduled without violating the dependence constraints between the operation and its successors. Once again, one can define two versions of Lstart: the

immediate Lstart and the transitive Lstart. Since the equations and procedures for calculating them are quite analogous to those for Estart, we do not bother to state them here.

4.5 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 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. Consequently, Estart must be calculated, as described in Section 4.4, just prior to scheduling an operation. The version of Estart that is used is the immediate Estart.

Dependences with predecessor operations are honored by not scheduling an operation before its Estart. In other words, MinTime is set equal to Estart (Figure 8). Symmetry might argue that no operation be scheduled later than its Lstart, the latest time at which an operation can be scheduled without violating a dependence with a scheduled successor, the motivation being to minimize the number of successor operations that will have to be rescheduled. However, this policy will, in general, lead to dead-end states in which no further operation can be scheduled. Some explicit scheme for backtracking must then be employed.

Instead, backtracking is built into iterative modulo scheduling by ignoring the dependence constraints of successor operations. In effect, the Lstart for the operation being scheduled is assumed to be infinite, just as with acyclic scheduling. 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. This constitutes the backtracking action. When these operations are scheduled subsequently, and Estart is computed for them, the dependence constraints are observed at that point. At any point in time, the partial schedule for the currently scheduled operations fully honors all constraints between the operations that have been scheduled.

In any event, it is pointless and redundant to consider more than II contiguous time slots starting with Estart. By definition, all time slots between Estart and Lstart (which in our case is viewed as being infinite) are legal from the point of view of dependence constraints. If the operation cannot

be scheduled at any time slot between Estart and Lstart, it must be because of resource constraints. Since we are using a MRT, if a legal time slot is not found in the range from Estart to (Estart + II - 1) because of resource conflicts, it will not be found outside this range either. In general, MaxTime is set equal to the smaller of Lstart and (Estart + II - 1) and only time slots in the range from MinTime (= Estart) to MaxTime are considered. If Lstart is viewed as infinite, as is the case in IterativeSchedule, MaxTime is equal to (Estart + II - 1).

4.6 Selection of operations to be unscheduled

Assume that a time slot is found between MinTime and MaxTime such that, for at least one of the alternatives for the current operation, there is no resource conflict with any currently scheduled operation. If so, the current operation is scheduled at this time. The only operations that will need to be unscheduled are those scheduled immediate successors with whom there is a dependence conflict because the current operation has been scheduled too late to honor the delay on the edge between it and its immediate successor.

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 because of resource conflicts. The first decision is made with an eye to ensuring forward progress (Figure 11); 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.

In this latter case, regardless of in 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 the operation, the alternative selected determines which operations have a resource conflict and must, therefore, be unscheduled. Ideally, we would like to select that alternative which displaces the lowest priority operations. Instead of attempting to make this determination directly, *all* those operations are unscheduled which would conflict with the current operation if it were scheduled at the selected time, using *any* of the alternatives. In addition, all immediate successors with a dependence conflict must be unscheduled. The current operation is then scheduled using one of the alternatives.

In all cases, the displaced operations will subsequently be rescheduled, many of them perhaps even at the very same time, in the order specified by the priority function.

```
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 11. The function FindTimeSlot.

5 Experimental results

5.1 The experimental setup

The experimental input to the research scheduler was obtained from the Perfect Club benchmark suite [10], the Spec benchmarks [73] and the Livermore Fortran Kernels (LFK) [50] 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 [20]. 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. Furthermore, profile statistics were gathered for the execution frequency of each basic block with the prescribed data input. These statistics are used below in estimating the effect of the various heuristics upon execution time. The input set to the research scheduler consisted of 1327 loops (1002 from the Perfect Club, 298 from Spec, and 27 from the LFK) of which 597 are actually executed when the actual benchmarks are run with the prescribed data input.

The statistics presented in Section 5.3 show that a very large fraction of the loops are simple enough that they can be scheduled at the MII and without any operation needing to be rescheduled. Consequently, they are of little value in discriminating between better and worse heuristics. These easy loops were filtered out to yield a more taxing benchmark consisting of those loops which either could not be scheduled at the MII or which required at least one operation to be rescheduled. It consists of 168 loops of which 73 are actually executed when the original benchmarks are executed with the prescribed data input. We shall refer to this benchmark as the "Difficult" benchmark and the one consisting of 1327 loops as the "Complete" benchmark.

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

5.2 Program statistics

Presented in Table 3 are various statistics on the nature of the loops in the Complete benchmark. Table 4 presents the same statisitics for the Difficult benchmark. In each case, the first column lists the measurement that was made on each loop, the second column lists the minimum value that the measurement could possibly yield, and the remaining columns provide various aspects of the distribution statistics for the quantity measured. The third column lists the fraction of loops for 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 observed for that measurement.

Table 3. Distribution statistics for various measurements on the Complete benchmark.

Measurement	Minimum Possible Value	Frequency of Minimum Possible Value	Median	Mean	Maximum Value
Number of operations	4	0.004	12	19.54	163
MII	1	0.286	3	11.41	163
Minimum Modulo Schedule Length	4	0.045	31	35.79	211
max(0, RecMII - ResMII)	0	0.840	0	4.54	115
Number of non-trivial SCCs	0	0.773	0	0.32	6
Number of vertices per SCC	1	0.930	1	1.30	42

Table 4. Distribution statistics for various measurements on the Difficult benchmark.

Measurement	Minimum Possible Value	Frequency of Minimum Possible Value	Median	Mean	Maximum Value
Number of operations	4	0.000	30	37.29	141
MII	1	0.000	30	29.18	163
Minimum Modulo Schedule Length	4	0.000	50	62.68	211
max(0, RecMII - ResMII)	0	0.464	7	13.36	115
Number of non-trivial SCCs	0	0.220	1	1.13	6
Number of vertices per SCC	1	0.855	1	1.72	42

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, and that the median and mean are both much closer to the minimum possible value than to the maximum value, indicates a distribution that is heavily skewed towards small values and has a long tail. The MII behaves in much the same way[1], 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 many 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. 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.

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 5.4 bears this out.

---

[1] The fact that the largest loop has 163 operations in it is unrelated to the fact that the largest MII is also 163. The loop with the most operations is not the same one as the loop with the largest MII.

As is to be expected, Table 4 shows that the distribution for each statistic for the Difficult benchmark is skewed towards a larger value than is the corresponding distribution for the Complete benchmark.

5.3 Characterization of iterative modulo scheduling

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

$$\text{EntryFreq}*\text{SL} + (\text{LoopFreq}-\text{EntryFreq})*\text{II}$$

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.

Table 5 presents statistics on how well iterative modulo scheduling performs. The nature of the columns is the same as for Tables 3 and 4. The first row of statistics is for the quantity DeltaII, the difference between the achieved II and the MII. The remaining statistics are for the ratio of a particular measurement on a loop to the smallest value that that measurement could possibly take. For instance, the second row of statistics is for the ratio of the achieved II for a loop to the MII for that loop.

The statistics for DeltaII show 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 II values considered is equal to DeltaII+1.

Table 5. Distribution statistics for various measures of algorithmic merit with a BudgetRatio of 6 for the Complete benchmark.

Measurement	Minimum Possible Value	Frequency of Minimum Possible Value	Median	Mean	Maximum Value
DeltaII	0	0.960	0.00	0.10	20.00
II (ratio)	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 operations scheduled for the final, successful II attempted (ratio)	1	0.900	1.00	1.03	4.33
Number of operations scheduled, cumulative across all II attempts (ratio)	1	0.873	1.00	1.64	121.52

These statistics also suggest that it is not beneficial to evaluate HeightR symbolically, as a function of II, as suggested by Lam for computing Estart [42]. 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 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 the second row of statistics in Table 5. 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 the third row of statistics. 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 no-stall execution time which is computed by using the above formula. A lower bound on the execution time is obtained by using the lower bounds for SL and II in that formula. 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 5. 54% of the loops achieved the lower bound on execution time and no loop had an execution time ratio of more than 1.5. 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 no-stall 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. And yet, 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. Consequently, this goal might not quite be achievable.

The last two rows in Table 5 provide statistics 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. The second last row reports on the scheduling inefficiency *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 only 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 once the II has been selected correctly.

When considering the efficiency of the procedure ModuloSchedule, one must also take into account the scheduling effort expended for the unsuccessful values of II. The last row of Table 5 lists statistics on the scheduling inefficiency measured over all of the II values attempted. The most

significant difference is that the maximum scheduling inefficiency goes from a ratio of 4.33 to a ratio of 121.52. This latter measure of scheduling inefficiency is sensitive to the parameter BudgetRatio, in procedure ModuloSchedule, which 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, and Budget is the limit on the number of operation scheduling steps performed before giving up on that candidate II.)

In collecting the statistics reported above, BudgetRatio was set at 6, well above the largest value actually needed by any loop, which was 4.33 (as can be seen from the second last row of Table 5). 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 will 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 12 shows the dilation in the aggregate execution time over all the loops (as a fraction of the lower bound) and the aggregate scheduling inefficiency, both as functions 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 2.8% at a BudgetRatio of 2, and eventually levelling off at 2.78%. The scheduling

inefficiency, however, first decreases from 2.56 down to 1.50 at a BudgetRatio of 1.75 and then begins to increase slowly.

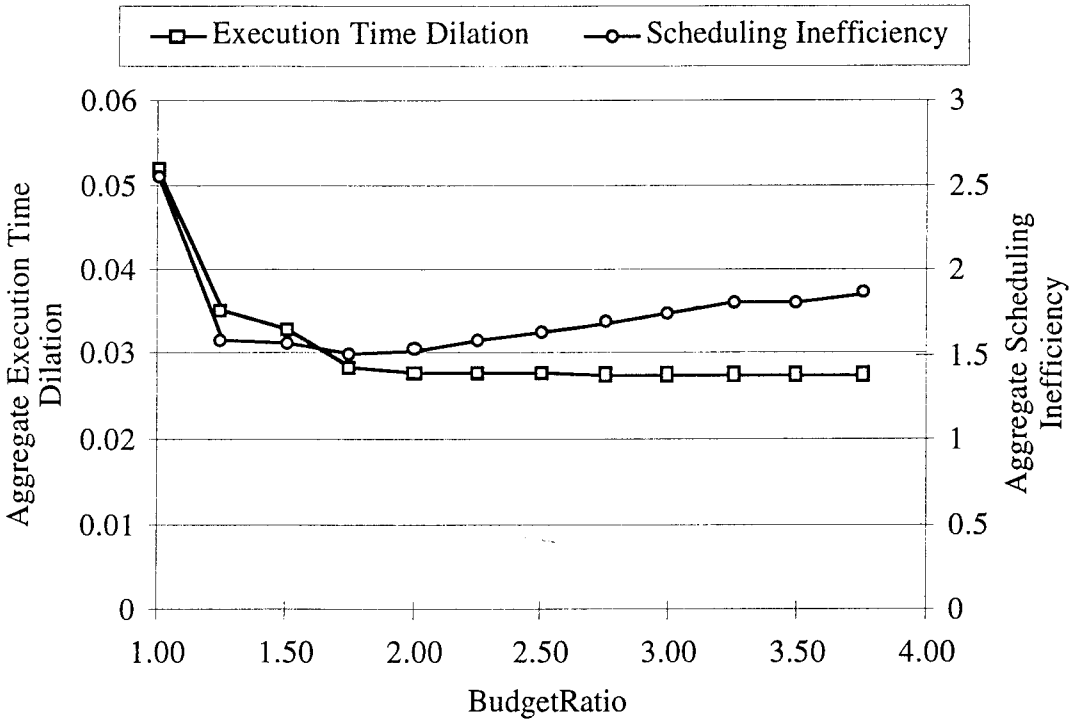

Figure 12. Variation of execution time and scheduling cost vs. the parameter BudgetRatio for the Complete benchmark.

At a BudgetRatio of around 2, both the execution time dilation (2.8%) and the scheduling inefficiency (1.53) 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, but using a BudgetRatio of 2 for the latter two would still yield near-optimality.) 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.

A BudgetRatio of 1 yields an execution time dilation of 5.2% and a scheduling inefficiency of 2.65. Whereas the 2.3% performance improvement gained by iterative modulo scheduling (with a BudgetRatio of 2) may not provide adequate motivation for the use of iterative modulo scheduling over some non-iterative form of modulo scheduling, the 42% reduction in scheduling inefficiency is more significant. The 10% of loops that cannot be scheduled at a scheduling inefficiency of 1 (even when the II is achievable) require that the II be increased beyond the minimum necessary, thereby contributing to the 42% difference in aggregate scheduling inefficiency. (The difference in sensitivity of scheduling inefficiency and execution time dilation to the BudgetRatio may be due to the fact that only 45% of the loops scheduled are actually executed. The two statistics are for different samples.)

Table 6. Distribution statistics for various measures of algorithmic merit with a BudgetRatio of 2 for the Complete benchmark.

Measurement	Minimum Possible Value	Frequency of Minimum Possible Value	Median	Mean	Maximum Value
DeltaII	0	0.956	0.00	0.11	20.00
II (ratio)	1	0.956	1.00	1.01	1.50
Schedule Length (ratio)	1	0.485	1.02	1.07	1.88
Execution Time (ratio)	1	0.539	1.00	1.05	1.50
Number of operations scheduled for the final, successful II attempt (ratio)	1	0.901	1.00	1.02	1.97
Number of operations scheduled, cumulative across all II attempts (ratio)	1	0.873	1.00	1.25	41.52

Table 6 presents the same statistics as in Table 5, but for a BudgetRatio of 2 instead of 6. With the exception of the scheduling inefficiency, the statistics are barely perturbed by reducing the BudgetRatio from 6 down to 2. However, the last two rows of statistics are appreciably smaller, especially with respect to the mean and the maximum value.

The best BudgetRatio for the Difficult benchmark, too, is 2. As BudgetRatio is increased from 1, the execution time dilation decreases monotonically from 11.5%, down to 5.63% at a BudgetRatio of 2, eventually levelling off at 5.55%. The scheduling inefficiency first decreases from 7.46 down to 3.07 at a BudgetRatio of 1.75 and then begins to increase once again. When the BudgetRatio is 2, the scheduling inefficiency is 3.21. For the Difficult benchmark, using acyclic list scheduling (BudgetRatio of 1) rather than iterative modulo scheduling with a BudgetRatio of 2, would result in a 5.9% increase in execution time and a 132% increase in scheduling inefficiency. Table 7 presents statistics similar to those in Table 6, but for the Difficult benchmark with a BudgetRatio of 2. As might be expected, all the distributions are skewed towards larger values.

Table 7. Distribution statistics for various measures of algorithmic merit with a BudgetRatio of 2 for the Difficult benchmark.

Measurement	Minimum Possible Value	Frequency of Minimum Possible Value	Median	Mean	Maximum Value
DeltaII	0	0.649	0.00	0.89	20.00
II (ratio)	1	0.649	1.00	1.04	1.50
Schedule Length (ratio)	1	0.316	1.05	1.11	1.88
Execution Time (ratio)	1	0.192	1.04	1.07	1.48
Number of operations scheduled for the final, successful II attempt (ratio)	1	0.214	1.08	1.16	1.97
Number of operations scheduled, cumulative across all II attempts (ratio)	1	0.000	1.20	2.96	41.52

From Figure 12 we see that we have come reasonably close to our goal of getting near-optimal performance at the same expense as acyclic list scheduling. For the Complete benchmark, and using a BudgetRatio of 2, we schedule on the average 1.53 operations per operation in the loop body. This means that, on the average, 0.53 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 2.06

(i.e., 1.53 + 0.53) times that of acyclic list scheduling, since the latter schedules each operation precisely once, and no operations are ever unscheduled. Even for the Difficult benchmark, iterative modulo scheduling with a BudgetRatio of 2 is only 5.42 times as expensive as acylic list scheduling.

This data enables an interesting comparison with "unroll-before-scheduling" schemes which rely on unrolling the body of the original loop prior to scheduling [30, 45, 37] and the "unroll-while-scheduling" schemes which unroll concurrently with scheduling [31, 12, 3, 56]. Iterative modulo scheduling gets to within 2.8% of the lower bound (for modulo scheduling) on execution time at an average cost equivalent to scheduling 2.06 operations per operation in the original loop. Some of these other approaches to software pipelining do have the potential to do better than modulo scheduling because each iteration is not constrained to have the same schedule, and because a different software pipeline can be used for each control flow path through the loop body. However, the improvement in code quality will need to be significant enough to warrant the extra scheduling cost. Typically, "unroll-before-scheduling" schemes unroll the loop body many tens of times [45], leading to a computational complexity far greater than that of iterative modulo scheduling. One might expect the "unroll-while-scheduling" algorithms to unroll to about the same extent. 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 expensive. Unfortunately, the performance and complexity of such approaches have never been characterized sufficiently, making a direct comparison with iterative modulo scheduling impossible.

5.4 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 8. 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 can be as much as $O(N^2)$, 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 3.0036 N. 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 as well as certain additional

precedence edges between operations which do not correspond to the flow dependences caused by the use of virtual registers. Since E is O(N), so is the complexity of identifying the SCCs.

Table 8. 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 reservation table for 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

$$11.9133\ N + 3.0474.$$

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

complexity is a variable 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 6.1474 N. 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 3.3321 N. 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

$$0.0327\ N^2 + 6.2799\ N.$$

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 8 we conclude that the statistical complexity of iterative modulo scheduling is $O(N^2)$ since no sub-activity is worse than $O(N^2)$.

6 Related work

There exists very little literature that discusses the modulo scheduling algorithm, itself, in the context of finite, realistic machine models. Previous modulo schedulers have tended to focus on the difficulty, due to the existence of deadlines, of scheduling operations that are on a recurrence circuit. This has been reflected in various schemes that schedule the SCCs first. In the prototype scheduler developed by the author at Cydrome, as well as in Cydrome's production compiler, all of the SCCs are iterative modulo scheduled together during a first phase. Then, during the second phase, the rest of the operations are added to the pool of operations to be scheduled and iterative modulo scheduling is continued [20]. The heuristics employed were different from those outlined in this report, were more expensive computationally, and often resulted in worse schedules.

Hsu first schedules the SCCs, using some form of exhaustive search through the space of schedules, and then performs stretch scheduling to schedule the remaining operations [34]. Stretch

scheduling relies on the fact that once an SCC has been successfully scheduled, it may thereafter be rescheduled as a unit, delayed in time by some multiple of II. This does not alter the state of the MRT, nor does it alter the relative scheduled times of the operations in the SCC. Once the SCCs have been scheduled successfully, they may be treated as atomic macro-operations, with complex reservation tables, embedded in the rest of the dependence graph. This dependence graph now appears to have no cycles in it (since all cycles are buried within one of the atomic units). The simple modulo scheduling, described in Section 2.7, may now be employed with the constraint that the time at which an SCC may be scheduled, modulo II, is fixed. If the operations' reservation tables are not simple, there is no guarantee that a schedule will be found. The schedules selected for the SCCs may be such that collectively they provide a given non-SCC operation with no legal schedule slot from a resource usage viewpoint. Stretch scheduling cannot alter this situation, and the II will have to be increased.

Lam's algorithm, too, utilizes the SCC structure but list schedules each SCC separately, ignoring the inter-iteration dependences [42]. Thereafter, just as Hsu does, each SCC is treated as a single macro-operation and the resulting acyclic dependence graph is scheduled using the simple modulo scheduling described in Section 2.7. With an initial value equal to the MII, the II is iteratively increased until a legal modulo schedule is obtained. By determining and fixing the schedule of each SCC in isolation, Lam's algorithm can result in a larger RecMII. The problem is that the independently generated schedules for the SCCs determine the II for the loop rather than vice versa. Also, because of the complex reservation tables of the macro-operations that represent the SCCs, it is more often impossible to schedule at the computation at the minimum achievable II.

On the other hand, the application of hierarchical reduction enables Lam's algorithm to cope with loop bodies containing structured control flow graphs without any special hardware support such as predicated execution. Just as with the SCCs, structured constructs such as IF-THEN-ELSE are list scheduled and treated as atomic objects. Each leg of the IF-THEN-ELSE is list scheduled separately and the union of the resource usages represents that of the reduced IF-THEN-ELSE construct. This permits loops with structured flow of control to be modulo scheduled. After modulo scheduling, the hierarchically reduced IF-THEN-ELSE pseudo-operations must be expanded. Each portion of the schedule in which m IF-THEN-ELSE pseudo-operations are active must be expanded into 2^m control flow paths with the appropriate branching and merging between the paths.

Since Lam takes the union of the resource usages in a conditional construct while predicated modulo scheduling takes the sum of the usages, the former approach should yield the smaller MII. However, since Lam separately list schedules each leg of the conditional creating pseudo-operations with complex reservation tables, the II that she actually achieves should deviate from the MII to a greater extent. Warter, et al., have implemented both techniques and have observed that, on the average, Lam's approach does result in smaller MII's but larger II's [75]. This effect increases for processors with higher issue rates.

Warter, et al., go on to combine the best of both approaches in their enhanced modulo scheduling algorithm. They derive the modulo schedule as if predicated execution were available, except that two operations from the same iteration are allowed to be scheduled on the same resource at the same time if their predicates are mutually exclusive, i.e., they cannot both be true. This is equivalent to taking the union of the resource usages. Furthermore, it is applicable to arbitrary, possibly unstructured, acyclic flow graphs in the loop body. After modulo scheduling, the control flow graph is re-generated using reverse IF-conversion [77]. Enhanced modulo scheduling results in MII's that are as small as for hierarchical reduction, but as with iterative modulo scheduling using predicates, the achieved II is rarely more than the MII.

One approach, guaranteed to find an optimal solution, is to pose the task of finding a modulo schedule as an integer linear programming problem [28, 6]. In fact, this strategy permits one to find a schedule that simultaneously minimizes the II and the number of registers required [33, 25]. Unfortunately, the execution time for solving integer linear programs is far too great for this approach to be seriously considered for use in production compilers. Various people have proposed the use of linear programming to find optimal schedules assuming unbounded resources [53, 16-18]. Since the resources are unbounded, what this approach computes, in reality, are quantities such as the RecMII, the earliest start times for all the operations, and a lower bound on the number of registers needed.

Huff [36] modified the Cydrome compiler to experiment with different priority schemes and demonstrated a scheduling heuristic that reduces register pressure relative to the greedy scheduler used in the Cydrome compiler. Wang, et al., too, have proposed heuristics-based algorithms for generating modulo schedules in a register-pressure sensitive manner [74]. Eichenberger has articulated algorithms, both optimal [24] and heuristics-based [27], for re-scheduling a loop after iterative modulo scheduling has generated an initial schedule in order to minimize the number of

registers needed. In both cases, the constraint is that operations are re-scheduled by a multiple of the II so that the MRT is unaltered.

7 Conclusion

In this report 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 53% 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 the loop body by more than 106% (which is just more than one single 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.

Currently, the major drawback to modulo scheduling using predicates is that in the case of loops containing multiple control flow paths through the loop body, the RecMII is computed based on the worst case recurrence circuit along all control flow paths, and the ResMII is computed taking the sum of the resource usage for each path. Counterbalancing this is the relatively compact code size of modulo schedules that use predicates.

Acknowledgements

This report, and the underlying research, have benefited from the ongoing discussions with and suggestions from Vinod Kathail, Mike Schlansker, Sadun Anik and Shail Aditya. 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. Balas Natarajan provided insight that was helpful in designing the algorithm for computing HeightR.

References

1. T. L. Adam, K. M. Chandy and J. R. Dickson. A comparison of list schedules for parallel processing systems. Communications of the ACM 17, 12 (December 1974), 685-690.

2. A. V. Aho, J. E. Hopcroft and J. D. Ullman. The Design and Analysis of Computer Algorithms. (Addison-Wesley, Reading, Massachusetts, 1974).

3. A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm, in Advances in Languages and Compilers for Parallel Processing, A. Nicolau, D. Gelernter, T. Gross and D. Padua (Editor). (Pitman/The MIT Press, London, 1991), 274-290.

4. V. H. Allan, U. R. Shah and K. M. Reddy. Petri Net versus modulo scheduling for software pipelining. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995).

5. J. R. Allen, K. Kennedy, C. Porterfield and J. Warren. Conversion of control dependence to data dependence. Proc. Tenth Annual ACM Symposium on Principles of Programming Languages (January 1983), 177-189.

6. E. R. Altman, R. Govindrajan and G. R. Gao. Scheduling and mapping: software pipelining in the presence of hazards. Proc. ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (La Jolla, California, June 1995), 139-150.

7. V. Bala and N. Rubin. Efficient instruction scheduling using finite state automata. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995).

8. S. Beaty. Genetic algorithms and instruction scheduling. Proc. 24th Annual International Symposium on Microarchitecture (Albuquerque, New Mexico, 1991), 206-211.

9. G. R. Beck, D. W. L. Yen and T. L. Anderson. The Cydra 5 mini-supercomputer: architecture and implementation. The Journal of Supercomputing 7, 1/2 (May 1993), 143-180.

10. M. Berry, *et al.* The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. The International Journal of Supercomputer Applications 3, 3 (Fall 1989), 5-40.

11. J. W. Bockhaus. An Implementation of GURPR*: A Software Pipelining Algorithm. M.S. Thesis. University of Illinois at Urbana-Champaign, 1992.

12. F. Bodin and F. Charot. Loop optimization for horizontal microcoded machines. Proc. 1990 International Conference on Supercomputing (Amsterdam, 1990), 164-176.

13. D. Callahan, S. Carr and K. Kennedy. Improving Register Allocation for Subscripted Variables. Proc. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (June 1990), 53-65.

14. A. E. Charlesworth. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 Family. Computer 14, 9 (1981), 18-27.

15. E. S. Davidson, L. E. Shar, A. T. Thomas and J. H. Patel. Effective control for pipelined computers. Proc. COMPCON '90 (San Francisco, February 1975), 181-184.

16. B. D. de Dinechin. An introduction to simplex scheduling. Proc. IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94 (Montreal, Canada, August 1994), 327-330.

17. B. D. de Dinechin. Simplex scheduling: more than lifetime-sensitive instruction scheduling. Technical Report 1994.22. PRISM, July 1994.

18. B. D. de Dinechin. Fast modulo scheduling under the simplex scheduling framework. Technical Report 1995.001. 1995.

19. A. De Gloria, P. Faraboschi and M. Olivieri. A non-deterministic scheduler for a software pipelining compiler. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, 1992), 41-44.

20. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing 7, 1/2 (May 1993), 181-228.

21. E. Duesterwald, R. Gupta and M. L. Soffa. A practical dataflow framework for array reference analysis and its use in optimizations. Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation (Albuquerque, New Mexico, June 1993), 68-77.

22. K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. Proc. 20th Annual Workshop on Microprogramming (Colorado Springs, Colorado, December 1987), 69-79.

23. K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture, in Languages and Compilers for Parallel Computing, D. Gelernter, A. Nicolau and D. Padua (Editor). (Pitman/The MIT Press, London, 1989), 213-229.

24. A. Eichenberger, E. S. Davidson and S. G. Abraham. Minimum register requirements for a modulo schedule. Proc. 27th Annual International Symposium on Microarchitecture (San Jose, California, November 1994), 75-84.

25. A. Eichenberger, E. S. Davidson and S. G. Abraham. Optimum modulo schedules for minimum register requirements. Proc. 1995 International Conference on Supercomputing (Barcelona, Spain, November 1995).

26. A. E. Eichenberger and E. S. Davidson. A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints. Technical Report CSE-TR-266-95. University of Michigan, 1995.

27. A. E. Eichenberger and E. S. Davidson. Stage scheduling: a technique to reduce the register requirements of a modulo schedule. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995).

28. P. Feautrier. Fine-grain scheduling under resource constraints. Proc. 7th Annual Workshop on Languages and Compilers for Parallel Computing, LNCS (Ithaca, New York, August 1994).

29. J. Ferrante, K. J. Ottenstein and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9, 3 (July 1987), 319-349.

30. J. A. Fisher. Trace scheduling: a technique for global microcode compaction. IEEE Transactions on Computers C-30, 7 (July 1981), 478-490.

31. J. A. Fisher, D. Landskov and B. D. Shriver. Microcode compaction: looking backward and looking forward. Proc. 1981 National Computer Conference (1981), 95-102.

32. F. Gasperoni and U. Schwiegelshohn. Scheduling loops on parallel processors: a simple algorithm with close to optimum performance. Proc. International Conference CONPAR '92 (1992), 625-636.

33. R. Govindrajan, E. R. Altman and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. Proc. 27th Annual International Symposium on Microarchitecture (San Jose, California, November 1994), 85-94.

34. P. Y. T. Hsu. Highly Concurrent Scalar Processing. Ph.D. Thesis. University of Illinois, Urbana-Champaign, 1986.

35. T. C. Hu. Parallel sequencing and assembly line problems. Operations Research 9, 6 (1961), 841-848.

36. R. A. Huff. Lifetime-sensitive modulo scheduling. Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation (Albuquerque, New Mexico, June 1993), 258-267.

37. W. W. Hwu, *et al.* The superblock: an effective technique for VLIW and superscalar compilation. The Journal of Supercomputing 7, 1/2 (May 1993), 229-248.

38. D. Jacobs, P. Siegel and K. Wilson. Monte Carlo techniques in code optimization. Proc. 15th Annual International Symposium on Microprogramming (1982).

39. S. Jain. Circular scheduling: a new technique to perform software pipelining. Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (June 1991), 219-228.

40. V. Kathail, M. Schlansker and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80. Hewlett-Packard Laboratories, February 1993.

41. W. H. Kohler. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. IEEE Transactions on Computers C-24, 12 (December 1975), 1235-1238.

42. M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (June 1988), 318-327.

43. D. M. Lavery and W. W. Hwu. Unrolling-based optimizations for software pipelining. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995).

44. E. L. Lawler. Combinatorial Optimization: Networks and Matroids. (Holt, Rinehart and Winston, 1976).

45. P. G. Lowney, *et al.* The Multiflow trace scheduling compiler. The Journal of Supercomputing 7, 1/2 (May 1993), 51-142.

46. S. A. Mahlke, *et al.* Sentinel scheduling: a model for compiler-controlled speculative execution. ACM Transactions on Computer Systems 11, 4 (November 1993), 376-408.

47. S. A. Mahlke, *et al.* Characterizing the impact of predicated execution on branch prediction. Proc. 27th International Symposium on Microarchitecture (San Jose, California, November 1994), 217-227.

48. S. A. Mahlke, *et al.* Effective compiler support for predicated execution using the hyperblock. Proc. 25th Annual International Symposium on Microarchitecture (1992), 45-54.

49. P. Mateti and N. Deo. On algorithms for enumerating all circuits of a graph. SIAM Journal of Computing 5, 1 (1976), 90-99.

50. F. H. McMahon. The Livermore Fortran kernels: a computer test of the numerical performance range. Technical Report UCRL-53745. Lawrence Livermore National Laboratory. Livermore, California, 1986.

51. S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992).

52. T. Muller. Employing finite automata for resource scheduling. Proc. 26th International Symposium on Microarchitecture (Austin, Texas, December 1993), 12-20.

53. Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. Proc. 20th ACM Symposium on Principles of Programming Languages (Charleston, South Carolina, January 1993), 29-42.

54. J. C. H. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58. Hewlett Packard Laboratories, May 1991.

55. T. A. Proebsting and C. W. Fraser. Detecting pipeline hazards quickly. Proc. 21st Annual ACM Symposium on Principles of Programming Languages (Portland, Oregon, January 1994), 280-286.

56. M. Rajagopalan and V. H. Allan. Efficient scheduling of fine-grain parallelism in loops. Proc. 26th International Symposium on Microarchitecture (Austin, Texas, December 1993), 2-11.

57. S. Ramakrishnan. Software pipelining in PA-RISC compilers. Hewlett-Packard Journal (July 1992), 39-45.

58. C. V. Ramamoorthy, K. M. Chandy and M. J. Gonzalez. Optimal scheduling strategies in a multiprocessor system. IEEE Transactions on Computers C-21, 2 (February 1972), 137-146.

59. B. R. Rau. Data flow and dependence analysis for instruction level parallelism, in Fourth International Workshop on Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua (Editor). (Springer-Verlag, 1992), 236-250.

60. B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. Proc. 27th Annual International Symposium on Microarchitecture (San Jose, California, November 1994), 63-74.

61. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. Proc. Fourteenth Annual Workshop on Microprogramming (October 1981), 183-198.

62. B. R. Rau, M. Lee, P. Tirumalai and M. S. Schlansker. Register allocation for software pipelined loops. Proc. SIGPLAN'92 Conference on Programming Language Design and Implementation (San Francisco, June 17-19 1992).

63. B. R. Rau, M. S. Schlansker and P. P. Tirumalai. Code generation schemas for modulo scheduled loops. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992), 158-169.

64. B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. Computer 22, 1 (January 1989), 12-35.

65. C. R. Reeves. Modern Heuristic Techniques for Combinatorial Problems. (Halsted Press: an imprint of John Wiley & Sons, Inc., New York, New York, 1993).

66. M. Schlansker and V. Kathail. Acceleration of first and higher order recurrences on processors with instruction level parallelism. Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing (Portland, Oregon, August 1993).

67. M. S. Schlansker, V. Kathail and S. Anik. Height reduction of control recurrences for ILP processors. Proc. 27th Annual International Symposium on Microarchitecture (San Jose, California, November 1994), 32-39.

68. B. Su and J. Wang. GURPR*: a new global software pipelining algorithm. Proc. 24th Annual International Symposium on Microarchitecture (Albuquerque, New Mexico, November 1991), 212-216.

69. J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. Communications of the ACM 13 (1970), 722-726.

70. P. Tirumalai, M. Lee and M. S. Schlansker. Parallelization of loops with exits on pipelined architectures. Proc. Supercomputing '90 (November 1990), 200-212.

71. M. Tokoro, T. Takizuka, E. Tamura and I. Yamaura. A technique of global optimization of microprograms. Proc. 11th Annual Workshop on Microprogramming (Asilomar, California, November 1978), 41-50.

72. M. Tokoro, E. Tamura, K. Takase and K. Tamaru. An approach to microprogram optimization considering resource occupancy and instruction formats. Proc. 10th Annual Workshop on Microprogramming (Niagara Falls, New York, November 1977), 92-108.

73. J. Uniejewski. SPEC Benchmark Suite: Designed for Today's Advanced Systems. SPEC Newsletter 1, 1 (Fall 1989).

74. J. Wang, A. Krall, M. A. Ertl and C. Eisenbeis. Software pipelining with register allocation and spill. Proc. 27th International Symposium on Microarchitecture (San Jose, California, November 1994), 95-99.

75. N. J. Warter, J. W. Bockhaus, G. E. Haab and K. Subramanian. Enhanced modulo scheduling for loops with conditional branches. Proc. The 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992), 170-179.

76. N. J. Warter, D. M. Lavery and W. W. Hwu. The benefit of predicated execution for software pipelining. Proc. 26th Annual Hawaii International Conference on System Sciences (Hawaii, 1993).

77. N. J. Warter, S. A. Mahlke, W. W. Hwu and B. R. Rau. Reverse if-conversion. Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation (Albuquerque, New Mexico, June 1993), 290-299.

78. H. Zima and B. Chapman. Supercompilers for Parallel and Vector Computers. (Addison-Wesley, Reading, Massachussetts, 1990).