AN IMPLEMENTATION OF GURPR*: A SOFTWARE PIPELINING ALGORITHM

BY

JOHN WILLIAM BOCKHAUS

B.S., University of Illinois, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank Professor Wen-mei Hwu for all of the time he has taken to advise me in both academics and research, from writing recommendation letters and suggesting coursework to helping me become involved with this project. I would also like to thank Nancy Warter for her guidance throughout my research. Finally, I want to thank my fiancée, Lynn Morstadt, for all of the support she has given in motivating me to complete this project.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

Recently, there has been a trend in uniprocessor design to increase the performance of the microprocessor by exploiting operation-level parallelism. Various architectures, including superscalar and VLIW machines, have been designed specifically for this purpose. Both of these architectures have the capability of issuing multiple operations each cycle. The problem is that the available operation-level parallelism within each basic block of a program has been shown to be close to 2, hardly enough to justify these new architectures [1],[2]. To increase the available operation-level parallelism, many new scheduling techniques have been developed: trace scheduling [3], superblock scheduling [4], and percolation scheduling [5], to name a few. All of these global compaction techniques are used in the compiler to schedule operations beyond basic blocks and, thus, create more operation-level parallelism.

Since loop execution time dominates total execution time, special consideration must be given to loops. To speed up loops, these scheduling techniques first unroll the loop a number of times and then apply the compaction algorithm. The problem is that

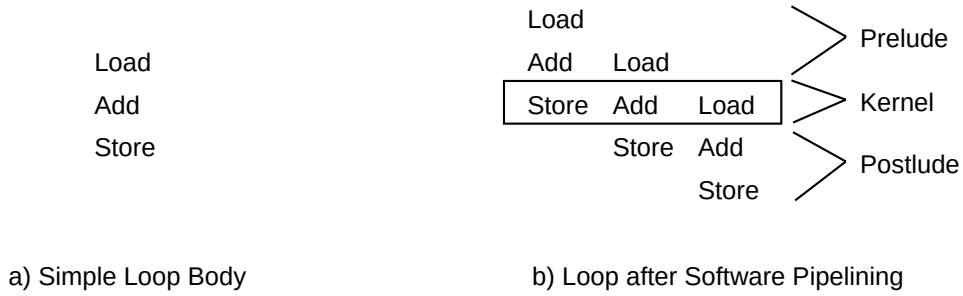


Figure 1.1: Simple Example of Software Pipelining

this method still imposes sequentiality between each group of k iterations, where k is the number of times the loop has been unrolled. Thus, better performance is achieved for larger k, but it comes at the cost of greater code expansion. Another technique for optimizing loops, software pipelining, claims to achieve the effect of unlimited loop unrolling and compaction with a fraction of the code expansion that loop unrolling would require to achieve the same performance.

The main idea of software pipelining is to overlap successive loop iterations to increase the available parallelism. It is analogous to hardware pipelining, where an operation is initiated every cycle even if it takes longer than one cycle to execute. However, with software pipelining, an *iteration* is started every II cycles, where II is the initiation interval. The software pipelined loop then consists of a prelude, kernel, and postlude. The prelude starts the first p iterations. After the first $p * II$ cycles, a steady state is reached, where one iteration is completed every II cycles. The postlude completes execution of the last p iterations. A simple example is shown in Figure 1.1. For loops with large trip counts, most of the execution time is spent executing the kernel. Thus, the goal of software pipelining techniques is to find the smallest possible kernel, or equivalently, II.

The original software pipelining idea arose from hand microcoding techniques and was first used with scientific programs in overlapping vector computations [6]. As software pipelining is applied to nonnumerical programs, the loops that are encountered are more complex, containing conditional branches and recurrences. Current algorithms must handle these complexities, as well as addressing issues such as resource constraints and register allocation. To see how these issues can be resolved, this thesis describes the implementation of a specific software pipelining algorithm, GURPR* [7], which is built into the IMPACT C compiler [8].

The remainder of this thesis is organized into four chapters. Chapter 2 provides an overview of several software pipelining algorithms, including the advantages and disadvantages of each and how each handles the fundamental issues mentioned above. In Chapter 3, the implementation details of GURPR* are discussed, including the specific features necessary for a working compiler. In addition, the limitations of this algorithm are presented, along with modifications to the original algorithm which are designed to improve its performance. Chapter 4 presents the experimental results of the original GURPR* algorithm along with three modified versions and compares them with the results of one other method, Enhanced Modulo Scheduling [9]. Chapter 5 provides a conclusion.

2. RELATED WORK

The idea of software pipelining has generated many viable algorithms. Each is slightly different in the way it handles conditional branches, recurrence relations, and resource constraints. Several of the techniques also require additional hardware support. Four different techniques, GURPR*, Modulo Scheduling, Perfect Pipelining, and Enhanced Pipeline Scheduling, are surveyed below.[1]

2.1 GURPR*

The first software pipelining techniques were applicable to loops consisting of a single basic block. One of these techniques, URCR (UnRolling, Compaction, and Rerolling) came about as an enhancement of trace scheduling [10]. In short, the algorithm unrolled the loop twice, compacted the operations, and searched for the shortest interval which contained all operations of the loop. This became the new loop body. As more complex

[1] Although the methods discussed in this thesis can be used for both VLIW and superscalar processors, VLIW terminology is used to clarify the discussion. Thus, an instruction refers to a very long instruction word which contains multiple operations.

loops were software pipelined, the algorithm was enhanced to handle specific problems. To handle recurrence relations, the URPR (UnRolling, Pipelining, and Rerolling) algorithm was presented [11]. Another version, GURPR (Global URPR), extended the algorithm for conditional branches [12]. Finally, GURPR* modified the conditional branch handling capabilities of GURPR to achieve better time efficiency [7].

The first step in the GURPR* algorithm is to compact the loop body. Next, the initiation interval (II) is computed as the maximum interbody dependence distance (D) [7]. The loop body is then pipelined, with successive iterations starting every II cycles. Since the operations must be kept in the same order as the original, compacted loop, and since the initiation interval, D, respects recurrences, dependences do not need to be considered during the pipelining phase; they are guaranteed to be satisfied. If a resource conflict is encountered during pipelining, the operation is delayed one cycle. If there is still a conflict, an empty cycle is inserted between the two conflicting cycles. After pipelining, the schedule is searched to find the shortest interval which contains all of the operations in the loop. This interval is the new initiation interval. Any redundant operations are deleted so that exactly one iteration is completed in the steady state. Finally, the prelude and postlude must be generated.

The advantages of this algorithm are that it requires no additional hardware support, dependences do not have to be considered during pipelining, and resource constraints are handled explicitly. Also, the time complexity of this algorithm is less than a similar algorithm, Perfect Pipelining, since GURPR* forces a steady state to form rather than

waiting for one to occur. The disadvantage is that potential parallelism is sacrificed in several ways. The scheduling of redundant operations in the new loop body prevents an optimal steady state since resources are required for each operation even though redundant operations will be deleted later. In addition, the unnecessary constraints of the precompacted loop and the lack of dependence analysis during pipelining require the algorithm to make worst-case assumptions about dependences.

2.2 Modulo Scheduling

The modulo scheduling technique originated as a scheduling technique for the polycyclic architecture. Rau and Glaeser proved that modulo scheduling for loops without conditional branches or recurrences will yield an optimal schedule [13]. The modulo scheduling algorithm determines the minimum initiation interval from the available resources and the recurrence relations and schedules all of the operations from one iteration into this II by delaying them (modulo II) to satisfy the resource constraints and dependences. To handle recurrence relations, the operations involved in the recurrence are scheduled first. If multiple paths of execution are present inside the loop due to conditional branches, the loop body is transformed into straight-line code by one of several methods. The various modulo scheduling techniques are differentiated by their approaches to conditional branches.

One such approach is hierarchical reduction, which reduces the entire conditional construct to a single path representing the union of the resource usages of each path [14].

The disadvantage of this method is that each path in the conditional construct is list scheduled and cannot change during modulo scheduling.

Another method for converting conditional constructs to straight-line code is if-conversion, which removes conditional branches by computing a condition (predicate) for the execution of each operation. An additional hardware feature, predicated execution, is required to conditionally execute each operation, based on its predicate. The disadvantage of this technique is that it requires the resource usage of the conditional construct to be the sum of the resource usages of each path, rather than using the union operator as with hierarchical reduction.

A technique that combines the best of the previous two methods is called Enhanced Modulo Scheduling [9]. Enhanced Modulo Scheduling uses if-conversion to transform the loop into straight-line code but then converts it back into multiple paths after software pipelining, thus removing the need for predicated execution hardware support. Since the technique converts the code back into multiple paths, the resource usage of a conditional construct can be the union of the resource usages of the two paths.

2.3 Perfect Pipelining

The Perfect Pipelining Algorithm is based on the fact that a loop which is repeatedly unwound and compacted eventually falls into a repeating pattern. The algorithm uses the core transformations of percolation scheduling as its compaction operators, namely, Unify, Delete, Move-operation, and Move-conditional jump [5]. The operations of loop

body n move as far up the current schedule (of loop bodies 1 to $n-1$) as possible, limited by several factors. An operation will not move up any farther in the schedule if the move would violate intraiteration or interiteration dependences nor will it move up if the move would violate the resource constraints. Conditional branches are also handled during compaction by migrating all operations of loop body n up each path in the current schedule.

An advantage of this technique is that it does not force the steady state to contain only one copy of each operation, thus allowing a greater possibility of achieving a time-optimal loop. The disadvantages are that it relies on a pattern to occur; the worst case running time of the algorithm is exponential. Finally, additional hardware support is needed in the form of a multiway jump mechanism.

2.4 Enhanced Pipeline Scheduling

Enhanced Pipeline Scheduling (EPS) extends a general machine model such that each instruction can be represented by a binary tree with its operations along the edges of the tree [15],[16]. A single binary tree, termed a node, contains all operations that can be executed at that cycle without violating dependences. A set of conditions defines which path through the tree is executed. Thus, conditional execution of operations is needed as an architectural feature, as well as multiway branching. Figure 2.1 illustrates an example node.

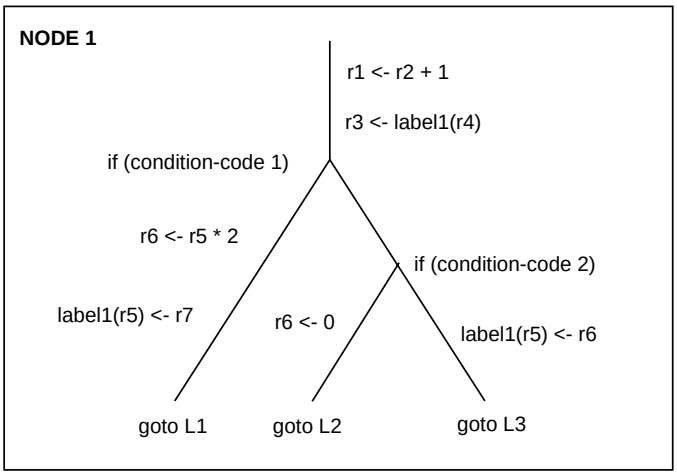


Figure 2.1: A Single Instruction (Node) in the EPS Machine Model.

The algorithm first compacts the operations in the loop into nodes, $(n_1, \ldots, n_m)$, using percolation scheduling. Next, the software pipeline is generated by attempting to start a new iteration every cycle. The first node in the software pipelined loop is n_1, which starts iteration 1 at cycle 1. The second node is created by appending the node n_1 to each branch of the node n_2. This new node executes an instruction from iteration 1 (n_2) and starts iteration 2 (n_1), all in cycle 2. The third node is n_3 with the previously created node ($n_2 n_1$) appended to each of its branches. This continues for the remaining nodes in the original loop. However, if there are resource conflicts or data dependence conflicts when trying to append a node n_p to a branch b in another node, the node n_p must be delayed and inserted into the node to which branch b jumps. If a loop contains recurrences, it needs to be unrolled so that dependences go from one iteration to the next.

The advantage of this technique is that it produces a flexible pipeline–it does not compromise the length of the shortest path. If a longer path is taken, the other iterations

which are currently executing must wait for the longer path to finish before they can continue.

3. IMPLEMENTATION OF GURPR*

3.1 The GURPR* Algorithm

An overview of the algorithm was given in the last chapter, but the complete step-by-step approach with some added comments is presented below [7]:

1. Compact the loop with a global compaction algorithm such as trace scheduling.
2. Create the parallel program flow graph (PPFG) from the compacted loop body. The PPFG is essentially a flow graph of the basic blocks in the loop body.
3. Construct the global interbody data dependence graph (GIBDDG) and determine the interbody distance, D.
   - D is the largest interiteration dependence distance.
4. Pipeline the loop bodies while maintaining the order of execution within each loop body, as determined during global compaction.
   - II is initially set equal to D.
   - Pipelining continues until the first operation of a new loop body starts after the last operation of the first pipelined loop body.
5. Determine the shortest interval in the pipeline that contains all operations from the loop. The length of this interval becomes the new II.
6. Delete any redundant operations within this interval, so that only one iteration will be executed every II cycles. This interval is the steady state.

7. "Unfold" the operations in these II cycles to create a new loop body. The length of this new loop body is L_{unfold}.

8. From this result, the software pipelined loop can be created by overlapping $\frac{L_{unfold}}{II}$ copies of this new loop body.

3.2 Implementation Details

3.2.1 Global compaction

The main idea behind the global compaction step is to create some initial operation-level parallelism and to shorten the loop body so that fewer loop bodies have to be overlapped to find a steady state. If fewer loop bodies are overlapped, the code expansion will be reduced. Also, with all operations compacted into a shorter length loop body, there is, theoretically, a better chance of finding a shorter steady state during the pipelining phase. The GURPR algorithm suggests using trace scheduling for the global compaction [12],[3]. The GURPR* algorithm suggests a different technique based on minimizing the global interbody dependence distance [17].

In an effort to determine the best global compaction technique to use, several techniques were applied to some sample loops. The results of percolation scheduling [18] and superblock scheduling [4], a variation of trace scheduling, were compared to the loops whose basic blocks had been list scheduled. The findings seem to preclude using trace scheduling or any of its variations. The fundamental idea in trace scheduling is to make the most frequently executed path through the program as efficient as possible, at the expense of all other paths. This method is not suitable for any fixed-II software pipelining

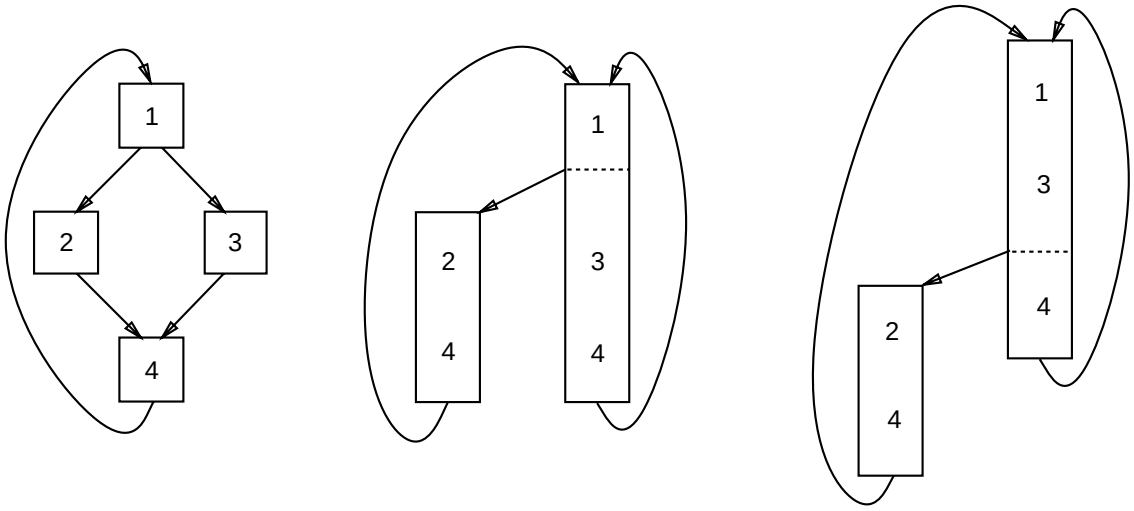


Figure 3.1: Superblock Scheduling.

technique[1] because the longest path through the loop determines the shortest possible steady state. In fact, the more aggressive techniques such as speculative execution may result in an even worse software pipeline because they add more operations which normally would not be executed to the least-frequently executed path. Figure 3.1(a) shows a simple loop with four basic blocks in its original form. Figure 3.1(b) shows the loop after superblock formation, with the dotted line representing the branch out of the superblock. Figure 3.1(c) shows how code might be moved above that branch such that the path off the primary superblock becomes much longer, since it executes so many operations that normally would not be executed on that path.

[1]With fixed-II techniques, each iteration in the software pipeline executes II cycles, no matter which path is taken through the loop. GURPR* and modulo scheduling are both fixed-II techniques.

The second method, percolation scheduling, has some of the same problems. Any operations moved from one path of a conditional construct to the head of that construct may create a longer path through the loop. For instance, in Figure 3.1(a), if operations are moved from basic block 2 and basic block 3 to basic block 1, more total operations will be executed no matter which path is taken.

In nonsoftware pipelined code, these global compaction techniques are necessary to create more operation-level parallelism for the underlying hardware. However, creating the maximum amount of parallelism before applying software pipelining is not critical because software pipelining itself creates the necessary operation-level parallelism. Thus, the global compaction algorithm used in conjunction with software pipelining should use a special set of rules to determine when to move an operation. The two most important rules seem to be:

1. If moving an operation does not increase the length of the longest path through the loop, the operation should be moved.
2. If moving an operation decreases the maximum interbody dependence distance, the operation should be moved.

Since these rules could conflict, a set of cost functions may be needed as well. Still, after preliminary experiments, the advantage of this new global compaction algorithm over a simple list scheduling algorithm for each basic block seems to be minimal.

One optimization that is made during global compaction is loop induction reversal [19]. This moves a loop induction variable that is incremented at the end of the loop to the beginning of the loop. As a result, all recurrences involving this variable are transformed into a single cycle dependence.

A restriction must also be added. When multiple paths merge back into one, the length of each of the shorter paths, $s_1, \ldots, s_m$, must be increased to equal the length of the longest path, l. Essentially, the unconditional branch operation in each of the merging basic blocks is delayed until a cycle equal to or greater than the last operation of the longest path. This guarantees that in generation of the software pipelined loop, if an operation from a different loop body needs to be inserted between any cycle s_m and cycle l, that operation can be inserted into both paths, thus preserving the semantics of that loop body. At the same time, a merge attribute is associated with the unconditional branches in the basic blocks which are merging. The value of the merge attribute is the path to which that basic block should merge. This annotation is used in generating the software pipelined loop and is explained more in Section 3.2.10.

An example loop and its compacted schedule are shown in Figures 3.2 and 3.3, assuming an issue rate of 2, uniform resources with the exception of one branch per cycle, and a latency of one cycle for each operation except load, multiply, and store, which have a latency of two cycles. The compacted schedule shows which operations would execute at which cycle. Each basic block has been list scheduled. Operations 8 and 9 can be scheduled together since they are on different paths in the loop and only one of them (or neither of them) will be executed each iteration. Since operation 5 merges two paths, it is delayed until the other path is done executing.

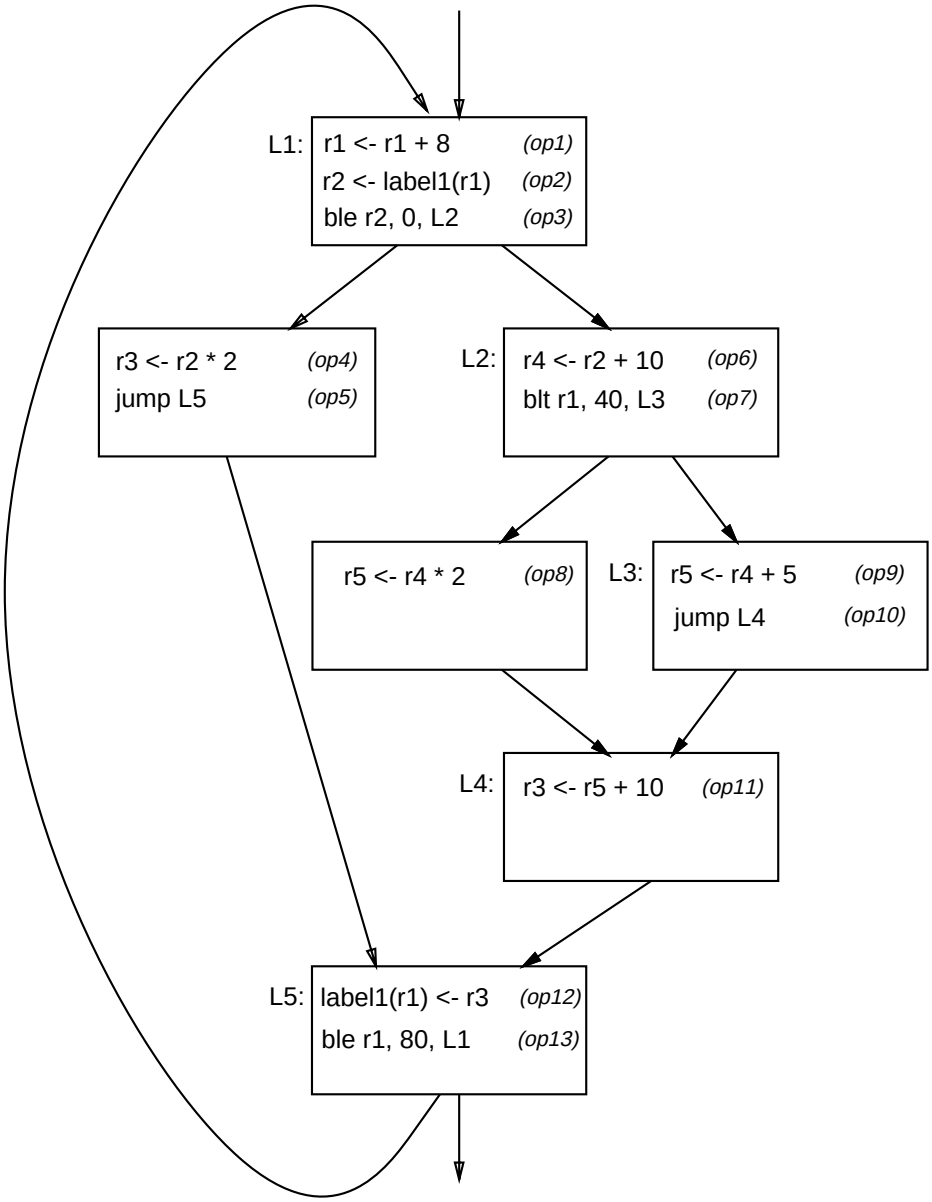


Figure 3.2: A Simple Loop.

Cycle 1:	1
Cycle 2:	2
Cycle 3:	
Cycle 4:	3
Cycle 5:	4, 6, 7
Cycle 6:	8, 9, 10
Cycle 7:	
Cycle 8:	5, 11
Cycle 9:	12, 13

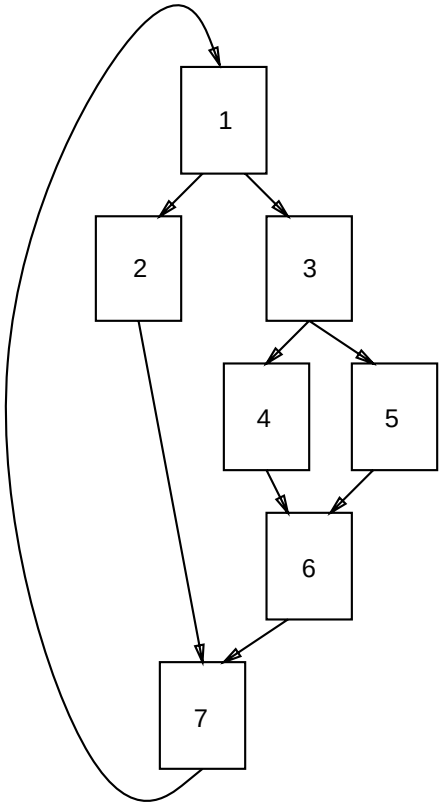
Figure 3.3: The Compacted Schedule of the Loop.

3.2.2 The parallel program flow graph

The parallel program flow graph (PPFG) is the flow graph of the basic blocks in the loop. This step of the algorithm has been modified so that it generates each possible path through the loop. This information is valuable in determining whether there is a control path between two operations when calculating their total resource usage. For example, since basic blocks 4 and 5 are on different paths, their operations can be scheduled in the same cycle without summing the resource usage; the total resources used will be the union of the resource usage for each path. The PPFG for the loop in Figure 3.2 is shown in Figure 3.4.

3.2.3 The global interbody data dependence graph

The global interbody data dependence graph is examined to find the maximum interbody dependence distance. Each arc in the original data dependence graph (DDG) is labeled with a (*distance*, *latency*) pair. The distance is the number of iterations the dependence spans. The latency is the minimum number of cycles the operation needs to



(a) Control Flow Graph

Path Number

1	2	3
1	1	1
2	3	3
7	4	5
	6	6
	7	7

(b) Modified PPFG

Figure 3.4: The Parallel Program Flow Graph.

complete. Any intrabody (loop-independent) dependences will have a distance of zero and will not be shown on the global interbody DDG. From the remaining dependence arcs, the maximum interbody dependence distance can be calculated as $D = \max_{i \in E}\{\lceil \frac{latency_i}{distance_i} \rceil\}$, where E is the set of all recurrences in the loop. For the pipelining algorithm, II is set to D. In the example, the only recurrence is the single cycle dependence where *r1* is incremented; thus, II is initially set to 1.

3.2.4 Pipelining the loop body

The pipelining phase starts out with the globally compacted loop body. The object is to pipeline multiple copies of this loop body to create a much shorter interval of cycles than the length of the original loop body, but still include all operations from the loop. Since this interval will contain operations from many different loop bodies, it will be "unfolded" to produce a *new* loop body, from which the software pipeline can be generated.

The pipelining algorithm overlaps loop bodies, each starting II cycles after the previous loop body, until the starting cycle of the next loop body is greater than the cycle time of the last operation in the first loop body. Initially, II is set to D. If a resource conflict occurs for an operation in the current loop body, that operation and the remaining operations in the current loop body are delayed by one cycle. If a resource conflict still occurs, an empty cycle is inserted between the two conflicting cycles.

Without resource constraints, $II = D$ for the entire pipelining process and $\lceil \frac{L_{gc}}{D} \rceil$ loop bodies will be overlapped, where L_{gc} is the length of the globally compacted loop body. However, when a resource conflict causes an operation to be delayed, II is increased, in case this operation is involved in a recurrence. The reason II must be increased is that operations which are part of a recurrence must stay the same relative distance from each other between different loop bodies. Once one is delayed, all others must be delayed. Since dependences are not checked during pipelining, any operation could be part of a recurrence. Thus, the worst case must be assumed, and II must increase for any operation that is delayed. Since II either increases or stays the same for each subsequent loop body, the number of overlapped loop bodies will be less than $\lceil \frac{L_{gc}}{D} \rceil$. Of course, as II increases, the loop bodies which have already been pipelined with a different II do not have to be changed.

Another consequence of the lack of dependence analysis during pipelining is that all operations that were scheduled for the same cycle in the globally compacted loop body must also be scheduled in the same cycle in the pipelined loop body. This handles the possibility of two operations in the same cycle being anti-dependent on each other.

Finally, to find the best possible steady state, the loopback branch is not inserted into the pipeline. It will be added after a steady state has been found. With these considerations, the pipelining routine is presented in Figure 3.5. The result of applying this routine to the example loop is shown in Figures 3.6, 3.7, and 3.8.

```
Pipeline()
{
  II = D;               /* the starting initiation interval */
  start_cycle = 1;      /* the starting cycle for the first loop body */
  added_cycles = 0;     /* the number of inserted cycles */
  last_cycle = latest issue time of the loop body;
  while (start_cycle ≤ (last_cycle + added_cycles)) {
      Overlap another copy of the loop beginning at start_cycle
      pipeline_cycle = start_cycle;
      for (each cycle in the original, compacted loop body) {
          Schedule all operations from that cycle in pipeline_cycle of the pipeline
            (except for the loopback branch operation)
          if (All these operations cannot be scheduled in this cycle) {
              pipeline_cycle++;
              II++;
              if (All operations cannot be scheduled in this cycle either) {
                  Insert a new cycle between the conflicting cycles
                    (all operations are guaranteed to fit in an empty cycle)
                  added_cycles++;
              }
          }
          pipeline_cycle++;
      }
      start_cycle += II;
  }
}
```

Figure 3.5: The Pipelining Routine.

Cycle 1:	1	
Cycle 2:	2	1’
Cycle 3:		2’
Cycle 4:	3	
Cycle 5:	4, 6, 7	
Cycle 6:		3’
Cycle 7:	8, 9, 10	
Cycle 8:		4’, 6’, 7’
Cycle 9:	5, 11	
Cycle 10:		8’, 9’, 10’
Cycle 11:	12	
Cycle 12:		5’, 11’
Cycle 13:		12’

Figure 3.6: Pipelined Loop Bodies 1 and 2.

Cycle 1:	1		
Cycle 2:	2	1’	
Cycle 3:		2’	
Cycle 4:	3		
Cycle 5:	4, 6, 7		
Cycle 6:		3’	1’’
Cycle 7:	8, 9, 10		
Cycle 8:			2’’
Cycle 9:		4’, 6’, 7’	
Cycle 10:	5, 11		
Cycle 11:			3’’
Cycle 12:		8’, 9’, 10’	
Cycle 13:			4’’, 6’’, 7’’
Cycle 14:	12		
Cycle 15:			8’’, 9’’, 10’’
Cycle 16:		5’, 11’	
Cycle 17:		12’	5’’, 11’’
Cycle 18:			12’’

Figure 3.7: Pipelined Loop Bodies 1, 2, and 3.

Cycle 1:	1			
Cycle 2:	2	1’		
Cycle 3:		2’		
Cycle 4:	3			
Cycle 5:	4, 6, 7			
Cycle 6:		3’	1’’	
Cycle 7:	8, 9, 10			
Cycle 8:			2’’	
Cycle 9:		4’, 6’, 7’		
Cycle 10:	5, 11			
Cycle 11:			3’’	
Cycle 12:		8’, 9’, 10’		
Cycle 13:			4’’, 6’’, 7’’	
Cycle 14:	12			1’’’
Cycle 15:			8’’, 9’’, 10’’	
Cycle 16:		5’, 11’		2’’’
Cycle 17:		12’	5’’, 11’’	
Cycle 18:			12’’	3’’’
Cycle 19:				4’’’, 6’’’, 7’’’
Cycle 20:				8’’’, 9’’’, 10’’’
Cycle 21:				
Cycle 22:				5’’’, 11’’’
Cycle 23:				12’’’

Figure 3.8: Pipelined Loop Bodies 1, 2, 3, and 4 with the Shortest Interval.

3.2.5 Searching for a steady state

After all copies of the loop body have been pipelined, a steady state for the software pipeline must be found. The steady state is the shortest interval of cycles that still contains all operations from the loop. The length of the shortest interval is the new II. Most likely, the interval will contain multiple copies of some operations. Any redundant operations must be deleted so that one iteration will be completed every II cycles in the steady state.

Since the loopback branch must be the last operation in the steady state, it was not inserted into the pipeline. This allows consideration of *any* interval in the pipeline as a possible steady state. For each interval that is considered, the loopback branch must be placed in the last cycle of that interval. If placing the loopback in that cycle violates any resource constraints or dependence relations, cycles must be added to the end of the interval until no conflicts occur. These extra required cycles are then taken into account when determining the shortest interval.

One potential problem not mentioned in the GURPR* algorithm is that dependences could be violated in the new loop body. In Figure 3.8, one possibility for the shortest interval is cycles 11 to 16. Assuming the loopback branch could fit in cycle 16, after deleting redundant operations and forming the new loop body, operation 3 would be executed one cycle after operation 2. However, there is a dependence latency of 2 from operation 2 to operation 3; this dependence is violated. To satisfy the dependence, an extra cycle would have to be added to the beginning or the end of the interval. This

problem can only occur at the boundaries of the steady state, between two different loop bodies. All operations from the same loop body are guaranteed to be scheduled correctly. The solution to this problem is to verify that all dependences are satisfied for each interval that is considered. If the dependences are not satisfied, the number of cycles that would have to be added to the end of the interval is taken into account when determining the shortest interval.

The algorithm to verify the dependences analyzes all operations in the given interval for each overlapped loop body. For each operation, the algorithm determines the dependences which are "live" at the last cycle of this interval. A dependence is live if its latency plus the cycle time of the given operation is greater than the last cycle of the interval. For each dependence that is live, the number of violation cycles, v, is calculated:

$$v = dependence\_latency + cycle\_of\_operation - last\_cycle\_in\_interval - 1$$

The destination operation of this dependence arc must not appear within the first v cycles of the previous loop body. If it does, the number of cycles which must be added to this interval to satisfy this dependence is calculated. The algorithm is shown in Figure 3.9.

The deletion of redundant operations is now a simple task. If n loop bodies have been overlapped, operations are selected from the steady state from loop body n to loop body 1. Any operation which has not been encountered yet is recorded and remains in the schedule. If an operation has already been encountered, this new occurrence of the operation is deleted from the schedule.

```
Verify_Dependences()
{
  conflict_cycles = 0;
  for (i = each loopbody, from n to 2) {
     for (j = each cycle in the interval, from start_cycle to end_cycle) {
        for (all operations from cycle j that are in loopbody i) {
           for (all dependences where
                        (dependence_latency + j - 1 > end_cycle)) {
              violation_cycles = dependence_latency + j - end_cycle - 1;
              cycle = start_cycle;
              loop_body = i - 1;
              while (violation_cycles > 0) {
                 search for destination operation of the dependence in cycle
                  from loop_body
                 if (found)             /* there is a conflict */
                    break;
                 cycle++;
                 violation_cycles--;
                 if (cycle > end_cycle) {
                    cycle = start_cycle
                    loop_body--;
                 }
              }
              new_conflict = violation_cycles / (i - loop_body);
              if (violation_cycles mod (i - loop_body) != 0)
                 new_conflict++;
              if (new_conflict > conflict_cycles)
                 conflict_cycles = new_conflict;
           }
        }
     }
  }
  return(conflict_cycles);
}
```

Figure 3.9: Verifying the Dependences for the Steady State.

Cycle 14:	12		1'''
Cycle 15:		8'', 9'', 10''	
Cycle 16:	5', 11'		2'''
Cycle 17:	12'	5'', 11''	
Cycle 18:		12''	3'''
Cycle 19:			4''', 6''', 7'''

(a) Shortest Interval with All Operations

Cycle 14:		1'''
Cycle 15:	8'', 9'', 10''	
Cycle 16:		2'''
Cycle 17:	5'', 11''	
Cycle 18:	12''	3'''
Cycle 19:		4''', 6''', 7'''
Cycle 20:	13''	

(b) Steady State

Figure 3.10: Determining the Steady State.

For the example, the shortest interval which contains all operations is shown in Figures 3.8 and 3.10. The steady state is shown in Figure 3.10.

3.2.6 The unfolded PPFG

A new loop body can be created by unfolding the steady state. Each interval of II cycles in the new loop body contains the operations from a different overlapped loop body. If n is the number of overlapped copies of the loop body present in the steady state, the new loop body has a length of $(n * II)$ cycles. The first II cycles in the new loop body contain the operations within the steady state which are from the nth overlapped

Cycle 1:	1
Cycle 2:	
Cycle 3:	2
Cycle 4:	
Cycle 5:	3
Cycle 6:	4, 6, 7
Cycle 7:	
Cycle 8:	
Cycle 9:	8, 9, 10
Cycle 10:	
Cycle 11:	5, 11
Cycle 12:	12
Cycle 13:	
Cycle 14:	13

Figure 3.11: The New Loop Body.

loop body. The next II cycles contain the operations from the $(n-1)$th loop body and so on until the last II cycles contain the operations from the first loop body. Figure 3.11 shows the new loop body created from the steady state in Figure 3.10(b), where $n = 2$ and $II = 7$. Cycles 1 to 7 contain the operations from loop body 2, and cycles 8 to 14 contain the operations from loop body 1.

Using this new loop body, an iteration can be started every II cycles with the guarantee that all dependences and resource constraints are satisfied.

3.2.7 Modulo variable expansion

Before overlapping the new loop body to form the software pipelined loop, we must determine if the steady state needs to be unrolled to avoid overlapping register lifetimes. Since one loop iteration can span multiple II's, the lifetime of a register can overlap

itself. To guarantee that a value in a register is not overwritten, the steady state must be unrolled enough times to satisfy the longest register lifetime. Then the overlapping registers are renamed. This optimization is called modulo variable expansion [14].

The lifetime, T, of a register is determined by the earliest time the value in that register is defined ($first\_define$) and the latest time the value in that register is used before the value in that register is redefined ($latest\_use$). Thus, $T = (latest\_use - first\_define + 1)$. The lifetime ($T_i$) of each register ($r_i$) is found from the new loop body. Then each register lifetime is calculated modulo II, $q_i = \lceil \frac{T_i}{II} \rceil$, to determine the minimum number of copies of r_i needed to avoid register conflicts. The minimum degree of unrolling, u, is equal to the maximum of all of the modulo lifetimes.

So that the register renaming scheme will work correctly when the flow of control loops back to the beginning of the kernel, the actual number of copies of r_i needed is k_i, where k_i is the minimum integer, such that $k_i \geq q_i$ and $u \bmod k_i = 0$ [20].

Normally, to create the software pipelined loop n copies of the new loop body will be overlapped, where n is the number of stages in one iteration and each stage is II cycles long. This produces a prelude and postlude which are each $(n-1)$ stages long and a kernel that is one stage long. To achieve the effect of unrolling the kernel, simply overlap $(u-1)$ more copies of the loop body in the final software pipelined loop such that $(n+u-1)$ total loop bodies are overlapped.

For the registers that need to be renamed, successive copies of the loop body will use different copies of that register, given in the *register renaming* array. The array

Table 3.1: Modulo Variable Expansion.

Original Register	Modulo Lifetime	Renamed Register 0	1
r1	$ML_{r1} = \lceil \frac{14-1+1}{7} \rceil = 2$	r1	r6
r2	$ML_{r2} = \lceil \frac{6-3+1}{7} \rceil = 1$	r2	N/A
r3	$ML_{r3} = \lceil \frac{12-6+1}{7} \rceil = 1$	r3	N/A
r4	$ML_{r4} = \lceil \frac{9-6+1}{7} \rceil = 1$	r4	N/A
r5	$ML_{r5} = \lceil \frac{11-9+1}{7} \rceil = 1$	r5	N/A

initially has one register specifier for each register r_i in the loop–its initial value. For each additional copy of r_i needed, a new register specifier is appended to r_i's row in the array. When generating the loop and overlapping the new loop body, all registers (r_i) will be renamed by indexing into the *register renaming* array at the row corresponding to r_i and the $((loop_body - 1) \bmod k_i)$th column, where *loop_body* is the number of the current loop body that is being overlapped, ranging from 1 to $(n + u - 1)$. The exception to this rule is if the register is used before defined and *loop_body* is greater than zero. In that case, the *register renaming* array must be indexed at column $((loop\_body - 2) \bmod k_i)$. The calculation of the modulo lifetimes and the resulting *register renaming* array are shown in Table 3.1.

At the end of the loop, if any registers are live, their values are copied to the corresponding original registers so that they will be correct for the rest of the program.

3.2.8 Reducing the length of the new loop body

For every loop body, $(b_1 \ldots b_k)$, that appears in the steady state, the new loop body, B, produced from "unfolding" the steady state, contains II cycles. If the number of loop bodies, k, in the steady state can be reduced, the length of the new loop body, L_{unfold}, can be reduced. Since the new loop body, B, is used to generate the software pipeline, less code expansion will result if L_{unfold} can be reduced. The register lifetimes are also calculated from this new loop body, B. The longer L_{unfold} is, the longer the register lifetimes will be, and the more the kernel will have to be unrolled. Thus, it is very important to keep the length of the new loop body, L_{unfold}, as short as possible.

Figure 3.10 identifies an optimization that has been made to reduce the length of the new loop body. After deleting redundant operations, there are no operations from loop body 1 or loop body 2 in the steady state. Thus, the loopback branch is inserted as an operation of loop body 3 rather than loop body 1, and the new loop body is formed by "unfolding" two loop bodies rather than four, thereby reducing the length of the new loop body. In general, if no operations are used from one of the overlapped loop bodies, those II cycles may not be necessary in the new loop body. However, before these cycles are removed, the dependences must be verified to ensure that no violations would occur if these cycles were deleted. This optimization does not decrease II; yet, as we have seen, it will reduce the code expansion and the number of registers needed.

3.2.9 Modifying the loopback

During the execution of one iteration in the software pipelined loop, $(n-1)$ more iterations are started. Thus, the loopback test, which is executed at the last cycle of iteration i and normally tests whether to execute iteration $(i+1)$, must instead test whether iteration $(i+n)$ should be executed. To achieve this effect while keeping the correct value for the induction variable throughout the iteration, the loopback operation is modified to test for $(n-1)$ fewer iterations. For this technique to work, the loop induction variable must be incremented by a value that is constant for each iteration, and the loop bound must be loop invariant. The new loop bound can be determined as such:

$$new\_loop\_bound = old\_loop\_bound - ((n-1) * increment\_value)$$

In the example, $n = 2$ and $increment\_value = 8$. Thus, the loopback operation will be changed to: $ble\ r2,\ 72,\ L1$.

3.2.10 Regeneration of the loop

After the pipelining phase, a new II is determined as the length of the steady state. This new II is used to generate the software pipelined loop and remains constant throughout that process. To create the software pipelined loop, $(n+u-1)$ loop bodies are overlapped, each starting II cycles after the previous one. For a single basic block loop, the regeneration process is simple. The algorithm starts with a single basic block loop and a set, J, of loop bodies containing only one element, j_1. The element j_i corresponds

to the ith loop body. Every II cycles, another loop body, j_{i+1}, is added to the current set of loop bodies, while $i \leq n + u - 1$. The algorithm generates the software pipelined loop cycle by cycle, inserting operations into cycle c of the software pipelined loop from cycle $(c - (II * (k - 1)))$ of loop body j_k, for all k in the set of loop bodies. For example, if cycle 10 of a software pipelined loop is being generated and $II = 4$, three loop bodies will be executing. Cycle 10 will contain all operations of cycle $(10 - (4 * (1 - 1))) = 10$ from loop body 1, all operations of cycle $(10 - (4 * (2 - 1))) = 6$ from loop body 2, and all operations of cycle $(10 - (4 * (3 - 1))) = 2$ from loop body 3.

After $((n-1)*II)$ cycles, a new basic block must be created; the loopback will branch to this basic block. The loopback operation is inserted into the software pipelined loop from only one of the loop bodies, the uth loop body. This ensures that it will be placed in the last cycle of the kernel, the $((n + u - 1) * II)$th cycle. After the loopback is inserted, a new basic block must be created to hold the remaining operations in the software pipelined loop.

After $((n + i - 1) * II)$ cycles, the last operation from loop body i has been inserted into the software pipelined loop. This loop body, j_i, is removed from the set J. The algorithm continues until there are no more loop bodies in the set J. Figure 3.12 shows the result of regeneration for a single basic block loop.

If the loop contains conditional constructs, the regeneration is more complicated. In this case, the regeneration algorithm maintains a set of leaf node basic blocks and a *path* array for each leaf. The *path* array for a certain leaf defines which operations from each

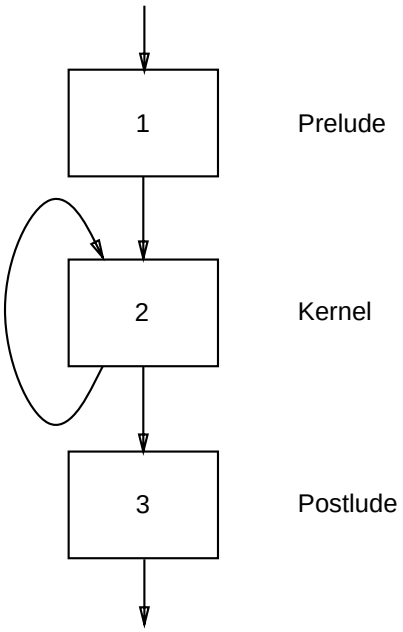
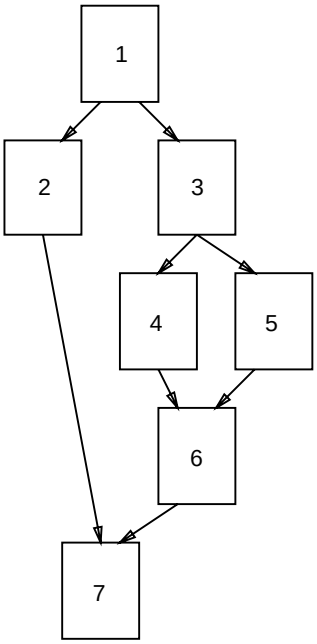


Figure 3.12: Regeneration of a Simple Loop.

loop body, j_i, are inserted into this leaf. Thus, if the value of the ith element of a leaf's *path* array is k, the operations from path k of loop body i are inserted into this leaf. Figures 3.13(a) and 3.13(b) show a loop and the three paths through the loop.

Figure 3.13(c) shows the start of the regeneration process for the loop in Figure 3.13(a). Each node is a basic block. The *path* array for each node is shown inside that node. The algorithm starts with the leaf node set containing only the root basic block with a *path* array of $(1, 0, 0, \ldots 0)$. The 1 in position one indicates that in this node, path 1 is being executed from loop body one. The zeroes indicate that no other loop body is valid. The algorithm continues as with the single basic block loop, except that each leaf must be checked to see which operations are inserted from which loop body. At cycle c of the software pipelined loop, each leaf will contain different operations. For example, if a

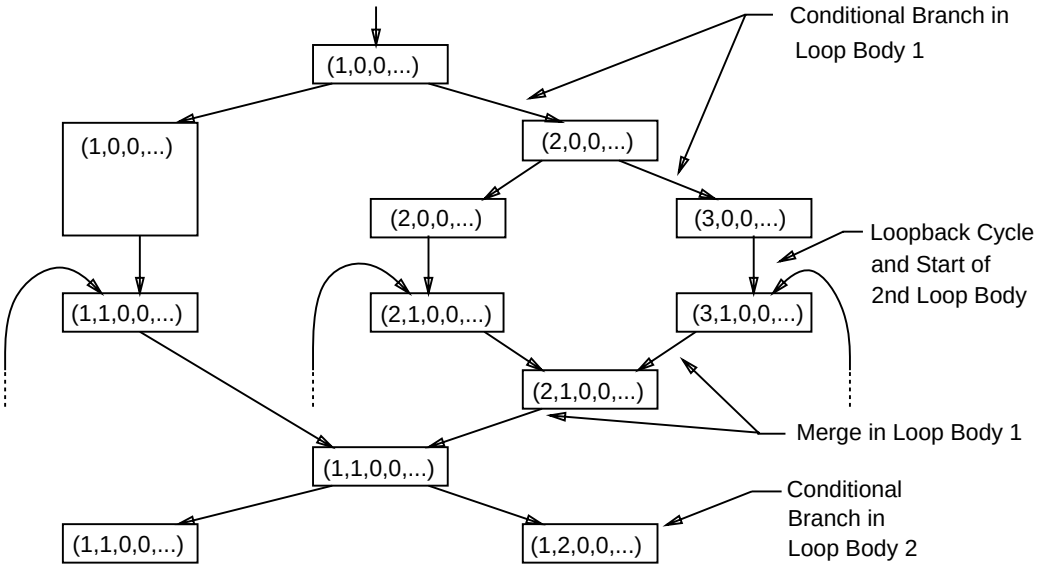


(a) Loop Body

Path Number

1	2	3
1	1	1
2	3	3
7	4	5
	6	6
	7	7

(b) Paths in the Loop



(c) Regeneration of the Software Pipelined Loop

Figure 3.13: Regeneration of a More Complex Loop.

certain node has a *path* array of $(3, 1, 0, 0 \ldots 0)$, cycle c of the software pipelined loop for that node will contain the operations from cycle c, path 3 of loop body 1, and operations from cycle $(c - II)$, path 1 of loop body 2.

When the ith loop body is initiated, a 1 is inserted into the *path* array for each leaf at the ith element to signify that path 1 is being executed. When the last operation of the ith loop body is reached, each leaf's *path* array has a 0 inserted at the ith element to signify that loop body i is no longer valid. If a conditional branch from the ith loop body is inserted into a leaf l, two new basic blocks are created. The *path* array is copied from the leaf l to each of the new basic blocks, with one change. The ith element of one of the new basic blocks is incremented to account for another path in the loop.

Merges take place when an unconditional branch is encountered with a merge attribute, which specifies to which path to merge. For example, if basic block 5 in the original loop has an unconditional branch to basic block 6, the value of its merge attribute will be 2 since basic block 6 is on path 2. During generation of the software pipelined loop, when this unconditional branch is encountered in loop body i, for each leaf, l, into which this operation is inserted, the regeneration algorithm searches for any other leaf which satisfies two conditions: 1) a 2 in the ith position of that leaf's *path* array, and 2) all other positions in that leaf's *path* array are identical to the *path* array of leaf l. In this case, the two (or more) leaves can be merged into a new basic block. The new *path* array for this basic block is the *path* array from any of the merging leaves.

The regeneration algorithm given above will handle conditional constructs such as *if-then-else* statements. However, if arbitrary control flow occurs inside the loop, a more powerful algorithm would have to be used. One possibility is to use the predication scheme and regeneration algorithm given in [9].

Figure 3.14 shows the software pipelined loop of the example loop in Figure 3.2. Each node is a basic block, and each row in a node is a VLIW instruction. For a VLIW processor without interlocking, the empty operation slots are filled with NO-OPs. For a VLIW processor with interlocking, instructions consisting only of empty slots are deleted. For partially full instructions, the empty operations are filled with NO-OPs. For a superscalar processor, all empty slots can be ignored.

3.2.11 Remainder scheduling

After generation of the software pipelined loop, extra code must be inserted to make sure that the correct number of iterations of the loop are executed. This is necessary since there is only one exit from the loop, at the end of the kernel. Without this restriction, each early exit from the software pipelined loop requires a special postlude, which increases considerably the code generation complexity and code expansion. With only one exit from the software pipelined loop, the loop must execute $((n-1)+k*u)$ times, where n is the number of stages of II cycles in one loop body after the unfolding phase, k is an integer greater than or equal to one, and u is the unrolling factor. Of course, if the loop trip count is not known in advance, as with *while* loops, early exits must be allowed [21].

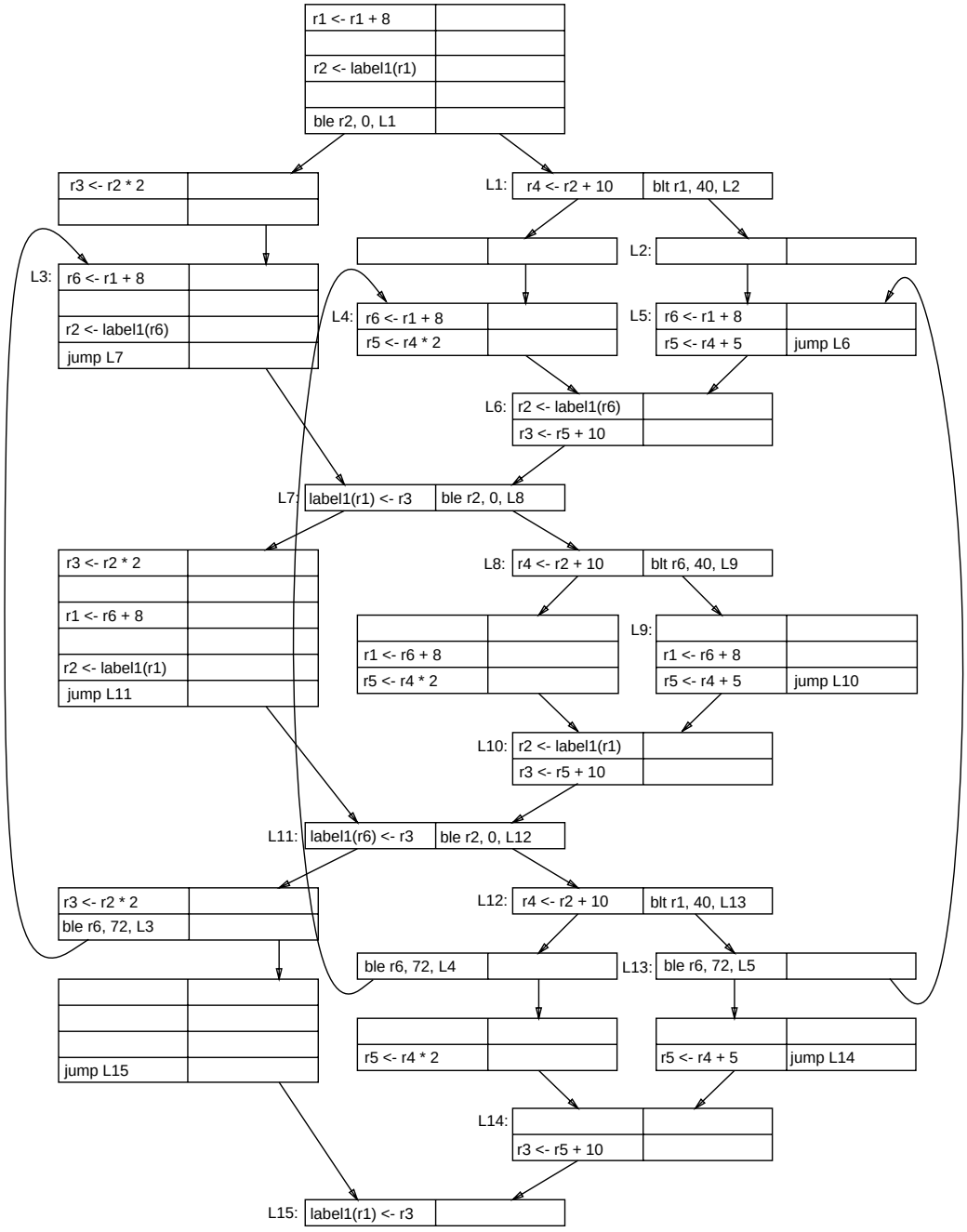


Figure 3.14: The Software Pipelined Loop after Regeneration.

```
  ⋮
loop:
     if (max < (n - 1) + u)
          goto remainder_loop;
     remainder = (max - (n - 1)) mod u;
     execute (max - remainder) iterations with the software pipelined loop
     if (remainder == 0)
          goto done;
remainder_loop:
     execute the remaining iterations with the nonsoftware pipelined loop
done:
  ⋮
```

Figure 3.15: Remainder Scheduling.

If we assume a simpler case, where the loop trip count is known, we can use a remainder scheduling technique to reduce the code expansion and the code generation complexity. With this technique, code is generated for both the original loop and the software pipelined loop. If the trip count is less than $((n-1)+u)$, only the original, nonsoftware pipelined loop is executed. If the trip count is greater than or equal to $((n-1)+u)$, the software pipelined loop executes $(trip\_count - remainder)$ iterations and the original, nonsoftware pipelined loop executes *remainder* iterations, where *remainder* is $((trip\_count - (n-1)) \bmod u)$. The necessary code is seen in Figure 3.15. Here *max* is the loop trip count. If *max* is a constant, it can be determined at compile time whether both of the loops are actually needed.

3.3 Improvements to GURPR*

3.3.1 Limitations of the algorithm

The major problem with the GURPR* algorithm is its handling of resource constraints. During the pipelining stage, as each loop body is inserted, the resource usage of the operations currently in each cycle determines whether an operation from this new loop body can be scheduled at that cycle. However, the resource usage for that cycle is calculated using the worst-case assumptions: that each operation currently scheduled at that cycle will be present in the steady state. In reality, many redundant operations occur in the interval chosen to be the steady state. Each redundant operation, which must be deleted, frees some resource that could have been used by a different operation. After deletion of all redundant operations, the available resources in the steady state are far from being fully utilized. By artificially constraining the resources, potential parallelism is sacrificed.

Another problem is the lack of dependence analysis during the pipelining stage. The worst-case assumptions must be followed in this situation as well, namely, that whenever an operation is delayed by a cycle, the initiation interval must be increased by one for all subsequent loop bodies, in case that operation is involved in a recurrence relation. In addition, operations compacted into one cycle by the global compaction algorithm cannot be split up, in case any one of them is anti-dependent on any of the others. Thus, if there are two slots available at cycle p of the pipeline, but the current cycle of the

new loop body contains three operations, all three operations must be delayed until cycle $(p+1)$. If the operations could be split up, a more compact schedule would result.

The following sections examine several new algorithms that are designed to improve the pipelining step of the GURPR* algorithm. They are intended to compensate for some of the deficiencies in the original algorithm.

3.3.2 Pipelining without insertion of cycles

The original algorithm states that whenever an operation cannot be scheduled in two consecutive cycles due to resource conflicts, an empty cycle is inserted between the two conflicting cycles. The justification for this rule is that II should be kept as small as possible during pipelining. If II is kept small, the resulting steady state should be shorter. However, this method tends to create many new cycles for each loop body that is overlapped. A large percentage of these new cycles are never filled with operations, leaving unutilized resources.

A simple change in the algorithm was made to test how beneficial the insertion of new cycles is. The new algorithm does not insert any cycles. Rather, it delays the operations until they can be scheduled. Of course, this may introduce large gaps between operations from the same loop body.

The result of applying this new pipelining algorithm to the sample loop of Figure 3.2 is shown in Figure 3.16. Note that now only five cycles are required for the steady state, rather than seven, as with the original GURPR* pipelining algorithm. Since no cycles

Cycle 1:	1	
Cycle 2:	2	1’
Cycle 3:		2’
Cycle 4:	3	
Cycle 5:	4, 6, 7	
Cycle 6:	8, 9, 10	
Cycle 7:		3’
Cycle 8:	5, 11	
Cycle 9:	12	
Cycle 10:		4’, 6’, 7’
Cycle 11:		8’, 9’, 10’
Cycle 12:		
Cycle 13:		5’, 11’
Cycle 14:		12’

(a) Pipelined Loop Bodies 1 and 2

Cycle 1:	1		
Cycle 2:	2	1’	
Cycle 3:		2’	
Cycle 4:	3		
Cycle 5:	4, 6, 7		
Cycle 6:	8, 9, 10		
Cycle 7:		3’	1’’
Cycle 8:	5, 11		2’’
Cycle 9:	12		
Cycle 10:		4’, 6’, 7’	
Cycle 11:		8’, 9’, 10’	
Cycle 12:			3’’
Cycle 13:		5’, 11’	
Cycle 14:		12’	
Cycle 15:			4’’, 6’’, 7’’
Cycle 16:			8’’, 9’’, 10’’
Cycle 17:			
Cycle 18:			5’’, 11’’
Cycle 19:			12’’

(b) Pipelined Loop Bodies 1, 2, and 3

Cycle 5:	4, 6, 7		
Cycle 6:	8, 9, 10		
Cycle 7:		3’	1’’
Cycle 8:	5, 11		2’’
Cycle 9:	12, 13		

(c) Steady State

Figure 3.16: Pipelining with No Insertion of Cycles.

are inserted, the operations in the first loop body remain close together, but still have enough available resources to include other operations in the interval, thus producing a slightly shorter steady state. However, a problem can occur. If most of the resources in a certain interval are already used, large gaps can occur in some of the loop bodies. This problem is seen mostly at higher issue rates when several loop bodies have been pipelined already and each interval is made up of multiple copies of identical operations. When different operations from subsequent loop bodies try to fit into one of these intervals, all of the resources are used, so they are delayed. If this situation occurs, the resulting steady state is usually longer.

3.3.3 Pipelining with early deletion of redundant operations

As mentioned above, the artificial resource constraints imposed by the GURPR* algorithm greatly reduce the potential parallelism. A solution to this problem is to delete an operation once it has been determined that the operation, at its current cycle, will not be in the resulting steady state. Several modifications to the GURPR* algorithm are necessary.

Since redundant operations are being deleted as the pipelining occurs, a search for the shortest interval which contains all operations from the loop is no longer applicable. At any one time, the pipeline will contain only one copy of each operation. Thus, an interval is chosen into which all operations will be channeled. This interval ends at the

last cycle of the first overlapped loop body, so that the starting interval contains the entire globally compacted loop body–one instance of each operation from the loop.

Having chosen the interval, we add several rules to the pipelining process. As the operations from a loop body are being inserted into the pipeline:

- If an identical operation occurs at an earlier cycle, delete that operation.
- If an identical operation occurs at the same cycle or a later cycle, do not insert the current operation.
- Do not insert any operations past the last cycle of the first overlapped loop body. Once that cycle is reached, we are done overlapping the current loop body.

These rules are designed to channel all operations into the shortest possible interval. By freeing the resources of redundant operations during the pipelining process, these resources can be used for operations of subsequent loop bodies, thus reducing the interval which contains all of the operations. Since we are no longer searching for the shortest interval, the loopback operation can be included in the pipelining process.

The results of applying this pipelining algorithm to the sample loop are shown in Figure 3.17. Normally, by deleting redundant operations, resources will be available for operations in later loop bodies so that cycles will not have to be inserted. However, in this example, after pipelining loop bodies 1, 2, and 3, the last three cycles are completely full. The final result is a longer steady state than for the previous two pipelining algorithms. This result is unusual; on average, this pipelining algorithm performs better than the previous two.

Cycle 1:	1	
Cycle 2:	2	1’
Cycle 3:		2’
Cycle 4:	3	
Cycle 5:	4, 6, 7	
Cycle 6:		3’
Cycle 7:	8, 9, 10	
Cycle 8:		4’, 6’, 7’
Cycle 9:	5, 11	
Cycle 10:		8’, 9’, 10’
Cycle 11:	12, 13	

(a) Pipelined Loop Bodies 1 and 2

Cycle 1:			
Cycle 2:		1’	
Cycle 3:		2’	
Cycle 4:			
Cycle 5:			
Cycle 6:		3’	1’’
Cycle 7:			2’’
Cycle 8:		4’, 6’, 7’	
Cycle 9:	5, 11		
Cycle 10:			3’’
Cycle 11:		8’, 9’, 10’	
Cycle 12:			4’’, 6’’, 7’’
Cycle 13:	12, 13		

(b) Pipelined Loop Bodies 1, 2, and 3

Cycle 1:				
Cycle 2:				
Cycle 3:				
Cycle 4:				
Cycle 5:				
Cycle 6:			1’’	
Cycle 7:			2’’	
Cycle 8:				
Cycle 9:	5, 11			
Cycle 10:			3’’	
Cycle 11:		8’, 9’, 10’		
Cycle 12:			4’’, 6’’, 7’’	
Cycle 13:				1’’’
Cycle 14:	12, 13			

(c) Pipelined Loop Bodies 1, 2, 3, and 4 with the Steady State

Figure 3.17: Pipelining with Early Deletion of Redundant Operations.

3.3.4 Pipelining with dependence analysis

This algorithm adds dependence analysis to the algorithm in the last section, Early Deletion of Redundant Operations. With this change, we no longer need to adhere to the worst-case assumptions of the GURPR* algorithm. II can now remain constant throughout the pipelining process. With the original GURPR* algorithm, any inserted cycles cause every operation in the next loop body to be delayed to satisfy the worst-case dependences. Now, any delays for an operation are based on the dependences of that operation. An operation will be delayed only if it is necessary to satisfy a cross-iteration dependence.

Each operation is scheduled based on its dependences. Thus, the operations originally compacted into one cycle can be split up into different cycles. If an operation, o, has an intraiteration dependence, it can be scheduled as long as the source operation, s, of the dependence arc has been scheduled. If operation s has not been scheduled yet, operation o must wait until operation s has been scheduled. Since we process the operations cycle by cycle from the globally compacted loop body, the dependences and delays are already enforced within each loop body. The only way an operation, o, would have to wait is if it is anti-dependent on another operation, s, in the same cycle. If an operation, o, has an interiteration dependence, it is ready to be scheduled since the source operation, s, of the dependence arc is in a previous "iteration," or loop body, and thus must have been scheduled already. In this case, the operation, o, is scheduled in the first cycle

which satisfies both this dependence and the resource constraints. The flexibility of this algorithm should produce a much shorter steady state.

Another change must be made to the algorithm for cases where D is small. The original GURPR* algorithm sets II to D, but II increases for every cycle that is inserted. Thus, even if $D = 1$, II increases quickly and the number of overlapped loop bodies will be much lower than $\lceil \frac{L_{gc}}{D} \rceil$, where L_{gc} is the length of the globally compacted loop body. However, since II is constant for this new pipelining algorithm, the number of pipelined loop bodies will actually be $\lceil \frac{L_{gc}}{D} \rceil$. If D is small, or if there are no recurrences ($II = 1$), the pipeline will overlap a large number of loop bodies. As a result, unfolding the steady state will produce a very long new loop body, which could cause exponential code growth.

To see why this is a problem, consider two cases for a compacted loop body of 40 cycles, as shown in Table 3.2. Case one sets II to one. Since II is constant throughout the pipelining process, 40 different loop bodies will be overlapped to find a steady state. Assuming that the minimum steady state is found to be 10 cycles, the new loop body will be $(40\ loop\_bodies * 10\ cycles) = 400$ cycles long. Generation of the software pipelined loop uses the length of the steady state as II. Thus, a new loop body will be overlapped every 10 cycles. If there is a conditional branch at cycle 2 which does not merge back into one path until cycle 398, the number of control paths is doubled every II cycles, until a merge occurs at cycle 398. For example, at cycle 2, a total of 2 control paths will exist. At cycle $(2 + II) = 12$, 4 control paths will exist. At cycle $(12 + II) = 22$, 8 control paths will exist. The maximum number of control paths will be 2^{40}, at the point where 40 loop

Table 3.2: Two Cases of Code Growth.

	II = 1	II = 10
Length of the Loop Body	40 cycles	40 cycles
Number of Overlapped Loop Bodies	40	4
Possible Steady State	10 cycles	12 cycles
Length of New Loop Body	400 cycles	48 cycles
Maximum Number of Paths	2^{40}	2^4

bodies have been overlapped. Obviously, the code growth produced by this method is unmanageable. If, on the other hand, II is set to the minimum possible interval based on resource constraints, many fewer loop bodies are overlapped. Although the resulting steady state probably will not be as short, the code growth is now manageable. In the example, we start out with $II = 10$. Now only four loop bodies must be overlapped to find a steady state. If the steady state is now 12 cycles long, the new loop body will be 48 cycles long. If we assume a conditional branch at cycle 2 and a merge at cycle 46, the maximum number of paths is only 2^4.

We can calculate the minimum II due to resource constraints by determining the most heavily utilized resource along any execution path. If an execution path p uses a resource r for c_{pr} cycles, and there are n_r copies of this resource, then the minimum II due to resource constraints, RII, is

$$RII = \max_{p \in P}(\max_{r \in R}\lceil \frac{c_{pr}}{n_r} \rceil)$$

where P is the set of all execution paths and R is the set of all resources. Thus, for this algorithm, II is set to $\max(D, RII)$.

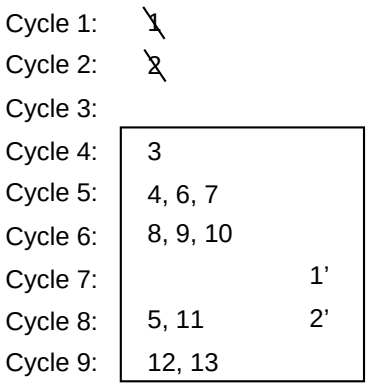


Figure 3.18: Pipelining with Dependence Analysis.

The results of applying this algorithm to the sample loop are shown in Figure 3.18. The length of the steady state, six cycles, is better than the results from two of the three other pipelining algorithms. On average, this algorithm is the best of the four pipelining algorithms.

3.3.5 One step further

The only problem with the algorithm in the last section, Pipelining with Dependence Analysis, is that all of the operations in the loop do not always fall into the shortest possible interval. Thus, why not force the operations to fall into the minimum interval, using dependence analysis to place them into the schedule? This change would actually create a modulo scheduling algorithm [13]. In the next chapter, the original GURPR* algorithm and the enhanced GURPR* algorithms presented in the previous sections will be compared to the Enhanced Modulo Scheduling (EMS) technique [9].

4. EXPERIMENTAL RESULTS

This chapter reports the results obtained from the GURPR* algorithm and each of its modified versions. In addition, results from the Enhanced Modulo Scheduling (EMS) technique are presented for comparison. There are two reasons why EMS was chosen to be compared against GURPR*. Unlike many software pipelining algorithms, neither EMS nor GURPR* requires any extra hardware support, such as multiway branching or conditional execution. Thus, the results from the techniques can be compared directly. Also, the three modifications to GURPR* discussed in Chapter 3 gradually were evolving into a variation of modulo scheduling. As mentioned in Section 3.3.5, one more improvement to GURPR* would have created a modulo scheduling technique. For that reason, we expect EMS to be the best of the five techniques.

4.1 The Compiler

The GURPR* algorithm and the three modified algorithms presented in this thesis have been built into the IMPACT C compiler [8]. The Enhanced Modulo Scheduling

Table 4.1: Operation Latencies.

fn	*cycles*
integer alu	1
integer mul/div	*
load	2
store	1
branch	1
Float load	3
Float/Double alu	3
Float mul	3
Double mul	4
Float/Double div	*

technique, to which we compare the GURPR* techniques, has also been built into this compiler. Each of these software pipelining algorithms is applied to the appropriate loops after performance of classical code optimizations [22] and after translation into the target machine assembly code but before register allocation. In this implementation, we apply software pipelining to inner loops that do not have function calls or early exits.

4.2 Machine Model

The machine model for these experiments is a VLIW processor with no interlocking. There are uniform resource constraints with the exception that only one branch can be issued per cycle. The target operation set is based on the Intel i860 processor with operation latencies shown in Table 4.1. The integer multiply and divide and the floating-point divide are implemented using approximation algorithms [23]. For the branch operation, we assume that the compare and branch are performed in one cycle. Thus, there are no branch delay slots.

Table 4.2: Characteristics of the 69 Sample Loops.

Loop Characteristics	
Simple Loops (A Single Basic Block)	62.32%
Loops with Simple Conditional Constructs	36.23%
Loops with Nested Conditional Constructs	1.45%

The base processor for these experiments is a RISC processor with an infinite register file and ideal cache. Each experiment is done for instruction widths (or issue rates) of 2, 4, and 8. The basis for comparison in each experiment is a basic block schedule.

4.3 Benchmarks

A set of 69 loops was collected from the Perfect and SPEC benchmarks. The characteristics of these loops are shown in Table 4.2.

4.4 Results

4.4.1 Performance

For loops with large trip counts, most of the execution time is spent executing the kernel. Thus, a good performance measure is the length of the kernel, which is II cycles long. To find the speedup of each software pipelining method, we assume that the time spent in the prelude and postlude is insignificant compared to the time spent in the kernel. Thus, the speedup is given by $\frac{II}{L_{orig}}$, where L_{orig} is the length of the original, nonsoftware pipelined loop body after simple basic block scheduling. Figure 4.1 shows the speedup of each of the techniques for issue 2, issue 4, and issue 8, where each point is the harmonic

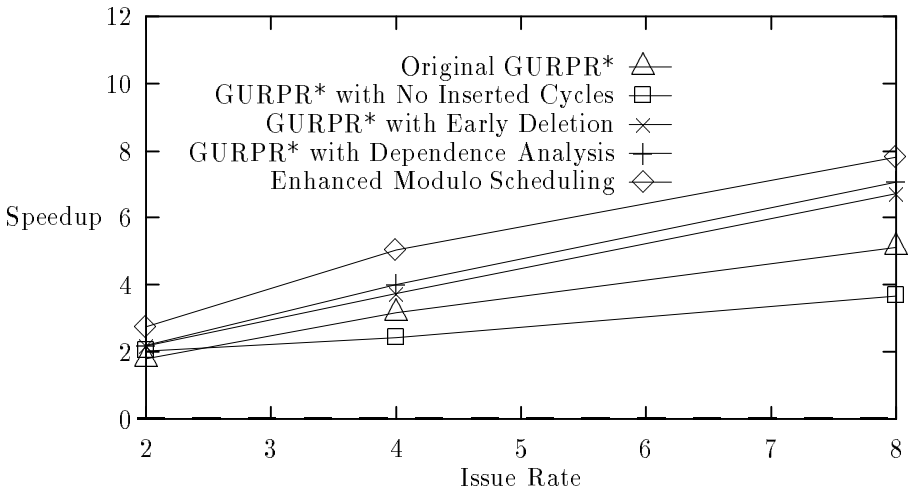


Figure 4.1: Speedup of GURPR* Techniques and Enhanced Modulo Scheduling.

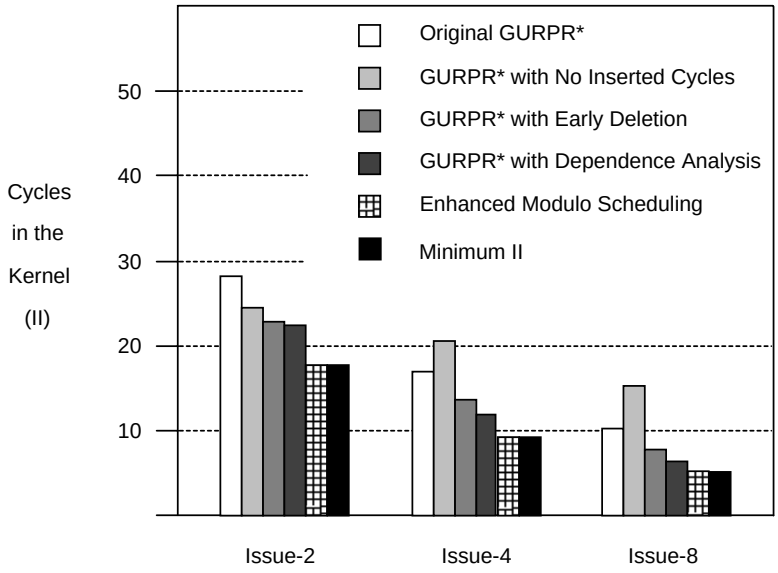


Figure 4.2: Comparison between the Five Methods and the Minimum II.

mean of the speedup of the individual loops. Figure 4.2 compares the arithmetic mean of the resulting II from the same five techniques with the minimum possible II. This minimum possible II is the same as described in Section 3.3.4. It is limited by resource constraints, RII, and the interbody dependence distance, D. The minimum II is then given by $\max(D, RII)$. In some of the loops, especially for issue 8, the restriction of one branch per cycle will be the limiting factor in achieving a smaller II. Enhanced Modulo Scheduling achieves the minimum II for nearly all of the loops. Each successive change to GURPR* improves the performance of the algorithm, with the exception of Pipelining without Inserted Cycles. Pipelining without Inserted Cycles improves upon the original GURPR* algorithm for issue 2, but lags behind all other techniques for issue 4 and issue 8. Pipelining with Early Deletion of Redundant Operations improves the performance by removing the artificial resource constraints and enabling more parallelism to arise. Pipelining with Dependence Analysis improves the performance even more by removing the worst-case assumptions of the algorithm and using dependences to schedule each operation. Still, none of the versions of the GURPR* algorithm match the effectiveness of Enhanced Modulo Scheduling.

4.4.2 Code expansion

The code expansion due to software pipelining arises for several reasons: 1) the software pipelined loop contains more than one copy of the original loop body, and 2) when the loop bodies are overlapped, an operation from one loop body may have to be placed

in multiple control paths. The code expansion is calculated by dividing the number of operations in the software pipelined loop by the number of operations in the original, nonsoftware pipelined loop. Figure 4.3 shows the arithmetic mean of the code expansion of each technique. These numbers do not include the operations required for remainder scheduling, such as the extra control operations or the extra nonsoftware pipelined loop, if necessary.

Normally, the algorithms which have the highest performance will also have the highest code expansion. With a shorter II, more loop bodies must be overlapped to find all operations from the loop, thus producing more code expansion and more register usage, as discussed in Section 3.2.8. However, even though the original GURPR* algorithm and the modification, Pipelining with Early Deletion of Redundant Operations, do not have the highest performance, they do have the highest code expansion because they overlap the highest number of loop bodies in the pipelining phase. The least code expansion is from the GURPR* modification Pipelining with Dependence Analysis. In this case, the initiation interval is fixed at the minimum achievable II throughout the pipelining process, and fewer iterations need to be overlapped to find a steady state. The reason this technique has less code expansion than EMS is because EMS achieves a smaller II and must overlap more loop bodies in the steady state.

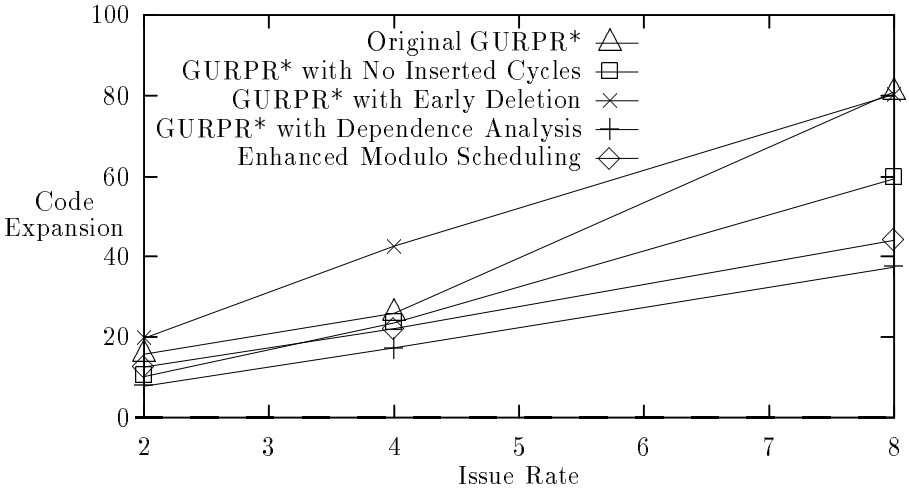


Figure 4.3: Code Expansion of GURPR* Techniques and Enhanced Modulo Scheduling.

4.4.3 The unrolling factor

The unrolling factor is the number of times the kernel must be unrolled due to overlapping register lifetimes. This is a measure of the relative number of registers needed. Figures 4.4, 4.5, and 4.6 show the unrolling factor for issue 2, issue 4, and issue 8. The numbers on the x-axis represent the upper bound of a given range (e.g., 4 refers to the range 0-4). At higher issue rates, more of the loop is packed into a smaller number of cycles; thus, more register lifetimes will overlap. The techniques which overlap more loop bodies in the steady state also require a larger unrolling factor, for the reasons given in the last section.

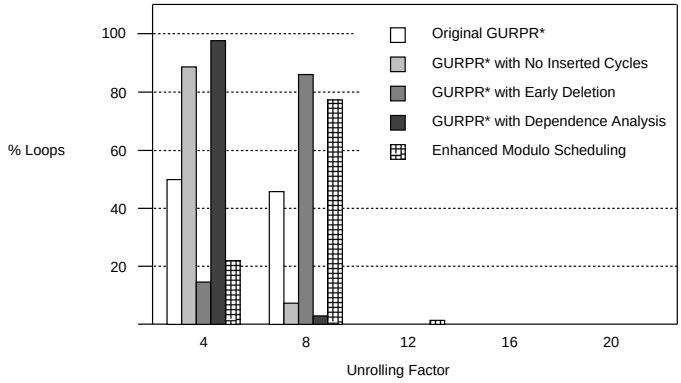


Figure 4.4: The Unrolling Factor for an Issue Rate of 2.

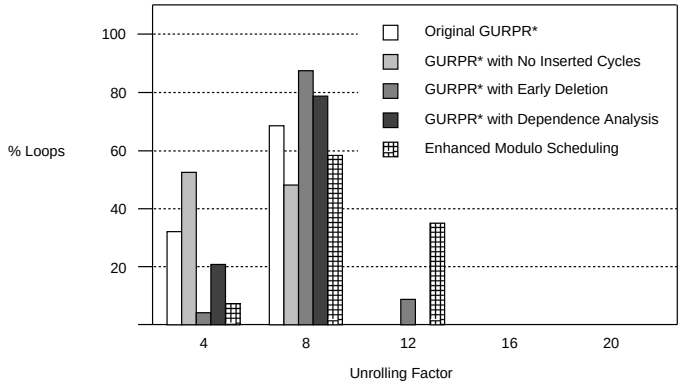


Figure 4.5: The Unrolling Factor for an Issue Rate of 4.

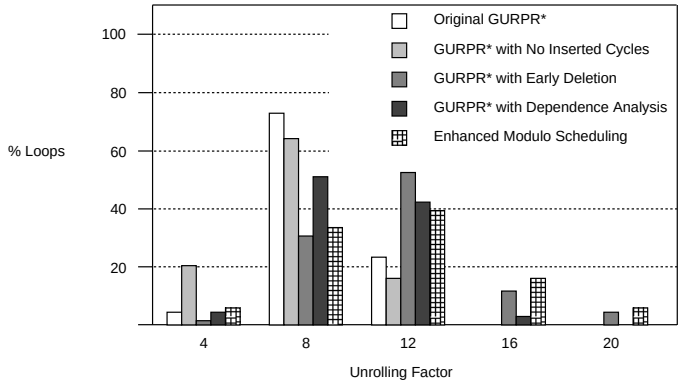


Figure 4.6: The Unrolling Factor for an Issue Rate of 8.

4.4.4 Minimum loop trip count

The minimum loop trip count is the minimum number of iterations that can be executed by the software pipelined loop. As was mentioned in Section 3.2.11 on remainder scheduling, if the trip count is less than this minimum, only the original, nonsoftware pipelined loop is executed. This minimum trip count is equal to the number of stages in the prelude and the kernel, where each stage is II cycles long. This measure reveals the magnitude of a loop trip count, such that software pipelining will improve the performance of the loop. Figures 4.7, 4.8, and 4.9 show the distributions of the minimum loop trip counts for issue 2, issue 4, and issue 8. As the issue rate increases, II decreases, and a single iteration spans more stages. Thus, the minimum loop trip count increases. As in the previous two sections, the techniques which overlap more loop bodies in the steady state also require a larger minimum loop trip count for the reason given above–a single iteration spans more stages.

4.4.5 Software pipeline startup delay

The software pipeline startup delay is equal to the number of cycles in the prelude, and shows how many cycles must be executed before the loop reaches the steady state. This delay is also equal to the number of cycles required to complete the unfinished iterations after the kernel is done executing. Figures 4.10, 4.11, and 4.12 show the pipeline startup delays. As the issue rate increases, the number of stages in the prelude increases, but II decreases. Thus, the pipeline startup delay does not change significantly for different

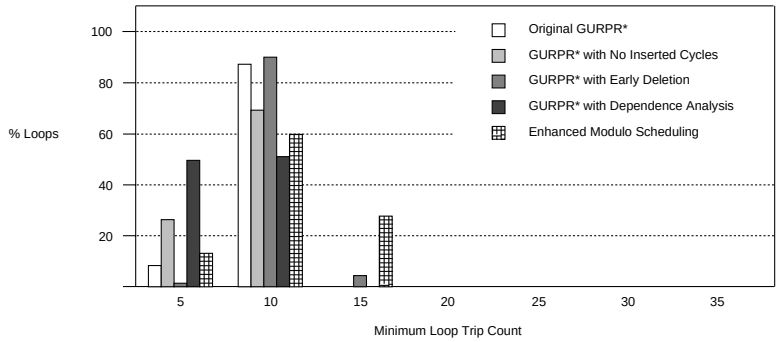


Figure 4.7: The Minimum Loop Trip Count for an Issue Rate of 2.

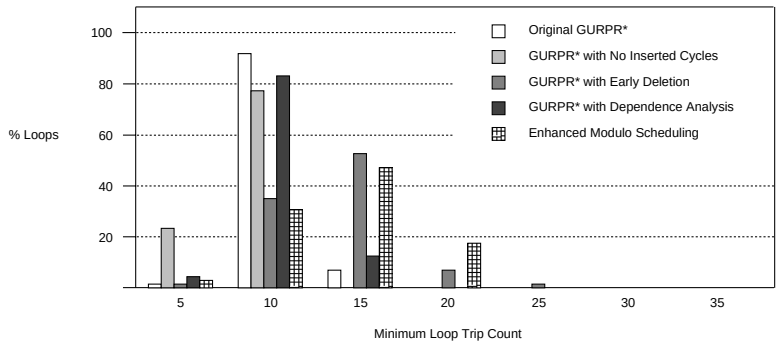


Figure 4.8: The Minimum Loop Trip Count for an Issue Rate of 4.

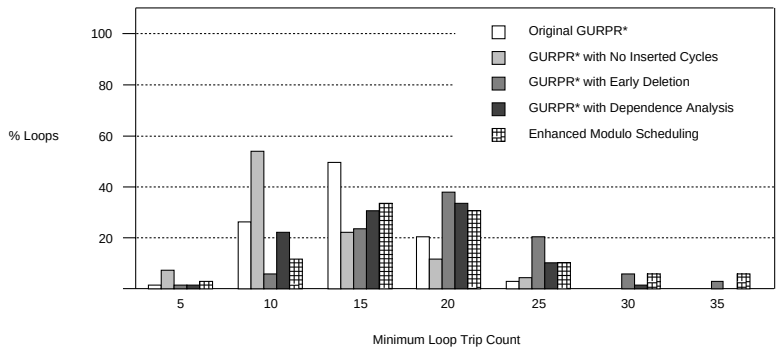


Figure 4.9: The Minimum Loop Trip Count for an Issue Rate of 8.

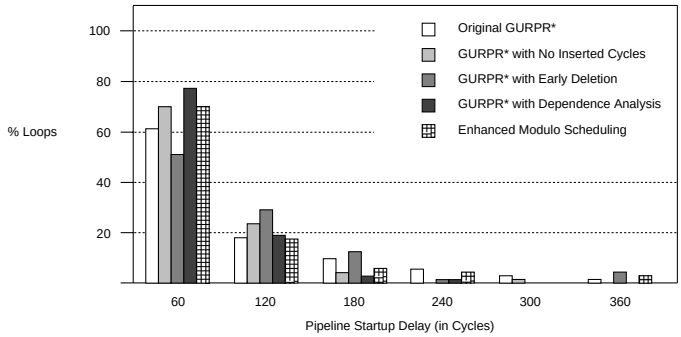


Figure 4.10: The Software Pipeline Startup Delay for an Issue Rate of 2.

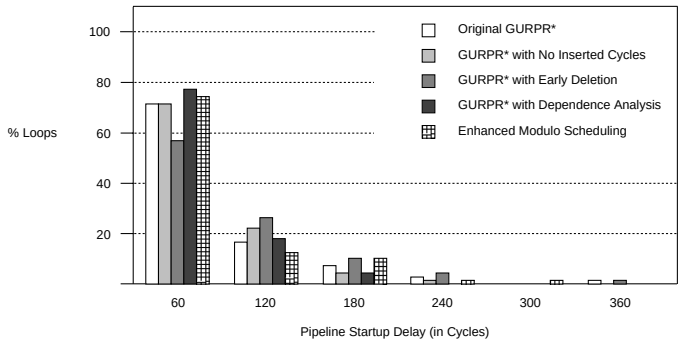


Figure 4.11: The Software Pipeline Startup Delay for an Issue Rate of 4.

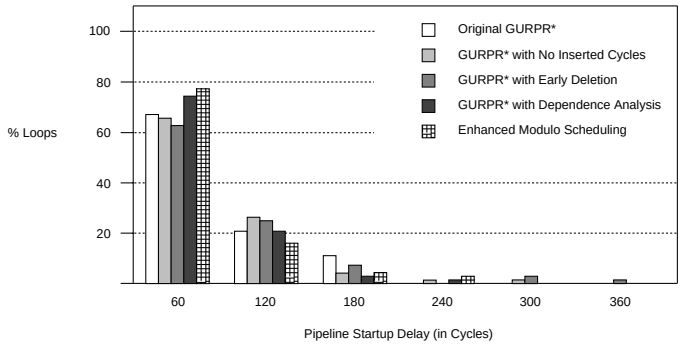


Figure 4.12: The Software Pipeline Startup Delay for an Issue Rate of 8.

issue rates. Among the different techniques, the GURPR* modification, Pipelining with Dependence Analysis, has the shortest startup delay.

4.5 Discussion

In analyzing the results, the last two modifications to the original GURPR* algorithm, Early Deletion of Redundant Operations and Pipelining with Dependence Analysis, improve the performance of the original algorithm by removing its weaknesses. Early Deletion of Redundant Operations removes the artificial resource constraints of the original algorithm, and Pipelining with Dependence Analysis adds dependence analysis to the technique Early Deletion of Redundant Operations. It is apparent that both removing the artificial resource constraints and adding dependence analysis are important to increasing the performance of a software pipelining algorithm. Thus, the performance of Pipelining with Dependence Analysis is closest to the performance of Enhanced Modulo Scheduling.

Another major disadvantage of the GURPR* technique is that it overlaps many loop bodies in the steady state, causing more code expansion, more required registers, and a greater minimum loop trip count. The technique Pipelining with Dependence Analysis removes this disadvantage and produces better results than any of the other techniques in terms of code expansion and the unrolling factor. Overall, that technique, Pipelining with Dependence Analysis, produces better results than any of the GURPR* algorithms.

5. CONCLUSIONS

This thesis has provided the implementation details of the GURPR* software pipelining algorithm along with three major modifications which point out and correct the weaknesses of the original GURPR* algorithm. Each of these algorithms has been applied to a suite of sample loops to obtain performance measures and other relevant statistics.

The main goal of Chapter 3 was to explain in full detail the complete implementation of GURPR*. Each phase of the algorithm was discussed with any assumptions that were made. That chapter also provided some insight into several issues that are not addressed in most software pipelining algorithms but are necessary for their implementation. Some of the issues included: how to globally compact the operations in the loop using a new algorithm specifically for software pipelining, how to handle overlapping register lifetimes using modulo variable expansion, and how to minimize the code generation complexity by using remainder scheduling.

Chapter 4 presented the results of the GURPR* algorithm, the three modified versions of GURPR*, and Enhanced Modulo Scheduling. These results indicate that the original

GURPR* algorithm can be improved by removing its artificial resource constraints and its worst-case assumptions about dependences. The first modification to GURPR*, No Insertion of Cycles, performs worse, on average, than the original GURPR* algorithm. This result justifies the insertion of cycles in the pipelining routine to keep identical operations in different iterations as close as possible. The second modification, Early Deletion of Redundant Operations, performs better than the original GURPR* algorithm by removing the artificial resource constraints. This result indicates that the benefits of searching for the shortest interval containing all operations from the loop are outweighed by allowing the maximum amount of parallelism to arise. The third modification to GURPR*, Pipelining with Dependence Analysis, removes both the artificial resource constraints and the worst-case assumptions about dependences. As expected, it comes closest to matching the performance of Enhanced Modulo Scheduling. The results also show other measures of interest when deciding whether to implement a software pipelining algorithm. The amount of code expansion, the unrolling factor (related to register usage), the minimum loop trip count, and the pipeline startup delay are all important issues.

More work needs to be done to analyze the performance of various software pipelining algorithms. This thesis has provided a starting point in understanding both the issues surrounding software pipelining and the details necessary for its implementation.

REFERENCES

[1] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.

[2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.

[3] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[4] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three superblock scheduling models for superscalar and superpipelined processors," Tech. Rep. CRHC-91-25, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, October 1991.

[5] A. Aiken and A. Nicolau, "Perfect pipelining: A new loop parallelization technique," Tech. Rep. 87-873, Department of Computer Science, Cornell University, 1987.

[6] A. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," *IEEE Computer*, September 1981.

[7] B. Su and J. Wang, "GURPR*: A new global software pipelining algorithm," in *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchitecture*, pp. 212–216, November 1991.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[9] N. Warter, J. Bockhaus, G. Haab, and K. Subramanian, "Enhanced modulo scheduling for loops with conditional branches," Tech. Rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1992.

[10] B. Su, S. Ding, and L. Jin, "An improvement of trace scheduling for global microcode compaction," in *Proceedings of the 17th Annual Workshop on Microprogramming and Microarchitecture*, pp. 78–85, December 1984.

[11] B. Su, S. Ding, and J. Xia, "URPR – an extension of URCR for software pipelining," in *Proceedings of the 19th Annual Workshop on Microprogramming and Microarchitecture*, pp. 104–108, October 1986.

[12] B. Su, S. Ding, J. Wang, and J. Xia, "GURPR: A method for global software pipelining," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 97–105, December 1987.

[13] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[14] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[15] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 69–79, December 1987.

[16] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[17] B. Su and J. Wang, "Loop-carried dependence and the general URPR software pipelining approach," in *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, vol. 2, January 1991.

[18] F. Gasperoni, "Compilation techniques for VLIW architectures," Tech. Rep. 435, New York University, March 1989.

[19] N. J. Warter, D. M. Lavery, and W. W. Hwu, "The benefit of predicated execution for software pipelining," Tech. Rep. CRHC-92-6, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, June 1992.

[20] M. Lam, *A Systolic Array Optimizing Compiler*, Ph.D. dissertation, Carnegie Mellon University, Pittsburg, PA, 1987.

[21] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," *Supercomputing*, November 1990.

[22] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley, 1986.

[23] Intel, *i860 64-Bit Microprocessor*, Santa Clara, CA, 1989.