code generation, modulo scheduling, software pipelining, instruction scheduling, register allocation, instruction-level parallelism, multiple operation issue, VLIW processors, Very Long Instruction Word processors, superscalar processors, pipelining

Software pipelining is an important instruction scheduling technique for efficiently overlapping successive iterations of loops and executing them in parallel. Modulo scheduling is one approach for generating such schedules. This paper addresses an issue which has received little attention thus far, but which is non-trivial in its complexity: the task of generating correct, high-performance code once the modulo schedule has been generated, taking into account the nature of the loop that is being scheduled. This issue is studied both with and without hardware features that are specifically aimed at supporting modulo scheduling.
1 Introduction

1.1 Software pipelining

Software pipelining is a loop scheduling technique which yields highly optimized loop schedules. Algorithms for achieving software pipelining fall into two broad classes:

- modulo scheduling, in which all iterations of the loop have a common schedule [1] and,
- algorithms in which the loop is continuously unrolled and scheduled until a situation is reached allowing the schedule to wrap back on itself without draining the pipelines [2].

Although, to the best of our knowledge, there have been no published measurements on this issue, it is the authors' belief that the second class of software pipelining algorithms can cause unacceptably large code size expansion. Consequently, our interest is in modulo scheduling. In general, this is an NP-complete problem and subsequent work has focused on various heuristic strategies for performing modulo scheduling [3-7] and the as yet unpublished heuristics in the Cydra 5 compiler [8]. Modulo scheduling of loops with early exits is described by Tirumalai, et al. [9]. Modulo scheduling is applicable to RISC, CISC, superscalar, superpipelined, and VLIW processors, and is useful whenever a processor implementation has instruction-level parallelism either by virtue of having pipelined operations or by allowing multiple operations to be issued per cycle.

This paper describes code generation alternatives for modulo scheduled loops, both on processors such as the Cydra 5 [10] which have hardware support for modulo scheduling as well as on other instruction-level parallel processors which do not. The focus of the paper is on precisely specifying the alternatives and far less on evaluating their relative merit. Hardware support for modulo scheduling includes rotating register files (register files which support compiler-managed register renaming, also known as the MultiConnect in the Cydra 5), predicated execution and the Iteration Control Register (ICR) file (a Boolean register file that holds the predicates), certain loop control opcodes [8, 10, 9] and support for speculative code motion [11]. The processor model assumes the ability to initiate multiple operations in a single cycle where each operation may have latency greater than one cycle. For brevity, we shall only discuss code generation for VLIW processors in which each instruction contains multiple operations, where each operation is equivalent to a RISC instruction. Nevertheless, everything discussed in this paper is applicable to RISC and superscalar processors as well; a left-to-right, top-to-bottom scan of the VLIW code would yield the corresponding RISC code. Attention must be paid to one detail, which is to ensure that in all cases the branch instruction is the appropriate number of instructions from the end of the basic block depending on the architecturally specified delay of the branch.

The examples discussed in this paper assume a processor with seven pipelined functional units as detailed in Table 1. The mnemonics for the various operations that are relevant to the examples are shown along with the unit on which those operations execute as well as their latencies.
Table 1: Description of the sample processor used in this paper.

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Operations Performed</th>
<th>Mnemonic</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>IALU 1</td>
<td>integer add</td>
<td>iadd</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>integer subtract</td>
<td>isub</td>
<td>1</td>
</tr>
<tr>
<td>IALU 2</td>
<td>integer add</td>
<td>iadd</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>integer subtract</td>
<td>isub</td>
<td>1</td>
</tr>
<tr>
<td>Memory Port 1</td>
<td>load</td>
<td>load</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>store</td>
<td>stor</td>
<td>1</td>
</tr>
<tr>
<td>Memory Port 2</td>
<td>load</td>
<td>load</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>store</td>
<td>stor</td>
<td>1</td>
</tr>
<tr>
<td>Floating Point Multiplier</td>
<td>floating multiply</td>
<td>fmul</td>
<td>4</td>
</tr>
<tr>
<td>Floating Point Adder</td>
<td>floating add</td>
<td>fadd</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>floating subtract</td>
<td>fsub</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>floating ≥</td>
<td>fge</td>
<td>2</td>
</tr>
<tr>
<td>Instruction Unit</td>
<td>branch operations</td>
<td>brtop</td>
<td>2</td>
</tr>
</tbody>
</table>

In the rest of this section, we provide a brief overview of modulo scheduling and discuss the problem that arises from the fact that after modulo scheduling, the successive lifetimes of a loop-variant variable are live concurrently. This section sets up the need for more sophisticated code schemas than have heretofore been discussed in the literature. In Section 2 we define and describe certain hardware capabilities to support modulo scheduling, some of which were present in the Cydra 5. The motivation for them is supplied in Section 3 which discusses the code schemas that are used for DO-loops and WHILE-loops depending on whether or not these hardware features are provided. Section 4 presents a few measurements on these code schemas performed on over 1350 loops.

1.2 An Overview of Modulo Scheduling

It is generally understood that there is inadequate instruction-level parallelism (ILP) between the operations within a single basic block and that higher levels of parallelism can only result from exploiting the ILP between successive basic blocks [12-17]. In the case of innermost loops, the successive basic blocks are the successive iterations of the loop. One method that has been used to exploit such inter-block parallelism has been to unroll the body of the loop some number of times and to overlap the execution of the multiple copies of the loop body [18]. Although this does yield an improvement in performance, the back-edge of the unrolled loop acts as a barrier to parallelism. Software pipelining, in general, and modulo scheduling, specifically, are scheduling techniques which attempt to achieve the performance benefits of completely unrolling the loop without actually doing so. The net result is that the interval between the initiation of successive iterations of the loop is less than the time that it takes to execute a single iteration. This overlapped, parallel execution of the loop iterations can yield a significant increases in performance.

The number of instruction issue cycles between the initiation of successive iterations in a modulo schedule is termed the **initiation interval** (II). This is also the number of (VLIW) instructions in the body of the modulo scheduled code if kernel unrolling (see Section 1.3 below) has not been employed. The objective of modulo scheduling is to engineer a common schedule for all iterations.
such that when successive iterations are initiated II cycles apart, no resource usage conflict arises between operations of either the same or distinct iterations. This requirement is met by constraining the schedule for a single iteration to be such that the same resource is never used more than once at the same time modulo the II. It is from this constraint that the name modulo scheduling is derived.

Lower bounds on II can be established through a simple analysis of the data dependence graph for the loop body. One bound is derived from the resource usage requirements of the graph while the other is derived from latency calculations around circuits defining recurrences within the data dependence graph for the loop body. The actual II must be greater than or equal to the maximum of these bounds.

The resource bound, \( \text{ResMII} \), provides a bound calculated by totaling, for each resource, the usage requirements within the program graph. Assume that functional units can be divided into classes. Each class contains a number of identical function units so that operations of a given type can be executed only within a single class and can be arbitrarily assigned to any member of that class. If \( N_i \) represents the number of operations of a loop body which are executed within class \( i \), and \( U_i \) represents the number of identical units within class \( i \), then II \( \geq \) ResMII where:

\[
\text{ResMII} = \max_i \left( \left\lfloor \frac{N_i}{U_i} \right\rfloor \right).
\]

The recurrence bound, \( \text{RecMII} \), is calculated using a unified control and data dependence graph. The vertices in this graph are the operations. The edges represent dependencies of all types between pairs of operations: control dependencies as well as data dependencies of all flavors (flow, anti- and output dependencies). The edges are labeled with the amount of time by which the start of the two operations at either end of the edge must be separated. In general, this is influenced by the type of the dependence edge. For each elementary circuit\(^1\) in the graph, the total latency of operations along that circuit divided by the total number of iterations spanned by the circuit is computed. Assume that a program has a set of elementary circuits \( C \). For any single circuit \( c \in C \), let \( n = l(c) \) be the number of operations in the circuit which define a closed path \( \text{op}_0, \text{op}_1, ..., \text{op}_{n-1} \). Let \( \text{lat}(c,j), 0 \leq j \leq l(c)-1 \), represent the delay on the edge between operations \( \text{op}_j \) and \( \text{op}_{j+1} \) in circuit \( c \). Operation \( \text{op}_{j+1} \) (where \( j+1 \) is calculated modulo \( n \)) must execute at least \( \text{lat}(c,j) \) cycles after \( \text{op}_j \). Let \( \omega(c,j) \) represent the number of iterations spanned by the dependence edge between \( \text{op}_j \) and \( \text{op}_{j+1} \). Thus, if \( \text{op}_j \) and \( \text{op}_{j+1} \) are within the same iteration, then \( \omega(c,j) = 0 \), if \( \text{op}_j \) and \( \text{op}_{j+1} \) are from adjacent iterations, then \( \omega(c,j) = 1 \), etc. The recurrence bound, \( \text{RecMII} \), is defined as

\[
\text{RecMII} = \max_{c \in C} \left( \left\lfloor \frac{l(c)-1}{\sum_{j=0}^{l(c)-1} \omega(c,j)} \right\rfloor \right)
\]

\(^1\) 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.
Any legal II must be equal to or greater than MAX(ResMII, RecMII). Figure 1 displays the FORTRAN DO-loop which we shall use as an example, and the corresponding intermediate representation using virtual registers. The left-hand column of Figure 1b lists the operations within the body of the loop and names used to refer to individual operations. These names are used in Figure 2a to specify at what time and on which functional unit each operation is scheduled after modulo scheduling is completed with an II of 3.

```
DO 10 I = 1,N
  Q = U(I) * Y(I)
  Y(I) = X(I) + Q
  X(I) = Q - V(I) * X(I)
10 CONTINUE
```

Figure 1: (a) A sample FORTRAN DO-loop. (b) Intermediate representation of body of loop.

The schedule for an iteration can be divided into stages consisting of II cycles each. The number of stages in one iteration is termed the stage count (SC). Each stage of the schedule in Figure 2a is demarcated by heavy lines. Figure 2b shows the record of execution of seven iterations of the modulo scheduled loop. The prefix before each operation's name indicates the iteration to which that operation belongs. The fifth and last stage of the first iteration occurs in cycles 12 through 14 and is concurrent with the first stage of the sixth iteration. Thereafter, in each succeeding stage, one iteration completes for every one that starts until the last iteration has started (in cycles 18 through 20). This is the steady-state part of the execution of the modulo scheduled loop. From cycle 23 onward, one iteration completes every II cycles until the execution of the loop completes at time 32. Note that although a single iteration takes 15 cycles to execute, each additional iteration takes only an additional 3 cycles. This is the motivation for performing modulo scheduling.

In generating code for a modulo schedule, one can take advantage of the fact that exactly the same pattern of operations is executed in each stage of the steady state portion of the modulo schedule's execution. This behavior can be achieved by looping on a piece of code that corresponds to one stage of the steady state portion of the record of execution. This code is termed the kernel. The record of execution leading up to the steady state is implemented with a piece of code called the prologue. A third piece of code, the epilogue, implements the record of execution following the steady state. Figure 3a shows the operations of one iteration of the loop partitioned into stages as specified by the schedule in Figure 2a, and Figure 3b shows the code for the kernel. Instruction i of the kernel includes all operations that are scheduled at time i modulo the II. Also, shown in Figure 3b is the stage of the schedule in Figure 3a from which each operation comes. Operations in the kernel code which are from distinct stages are from distinct iterations of the original loop. Note that since the branch operation, B1, determines whether or not another iteration is to be executed, and since its latency is 2 cycles, it must be scheduled in the second last instruction of the kernel.
<table>
<thead>
<tr>
<th>Time</th>
<th>Modulo 3</th>
<th>IALU 1</th>
<th>IALU 2</th>
<th>Memory Port 1</th>
<th>Memory Port 2</th>
<th>Multiplier</th>
<th>Adder</th>
<th>Instruction Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>P1</td>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>P3</td>
<td></td>
<td>R1</td>
<td>R2</td>
<td></td>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>P4</td>
<td></td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>M1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>M2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td></td>
<td></td>
<td>W1</td>
<td></td>
<td></td>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td></td>
<td></td>
<td>W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Time</th>
<th>Modulo 3</th>
<th>IALU 1</th>
<th>IALU 2</th>
<th>Memory Port 1</th>
<th>Memory Port 2</th>
<th>Multiplier</th>
<th>Adder</th>
<th>Instruction Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1:P1</td>
<td>1:P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1:P3</td>
<td></td>
<td>1:R1</td>
<td>1:R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>1:P4</td>
<td></td>
<td>1:R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2:P1</td>
<td>2:P2</td>
<td></td>
<td>1:R4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2:P3</td>
<td></td>
<td>2:R1</td>
<td>2:R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
<td>2:P4</td>
<td></td>
<td>2:R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>3:P1</td>
<td>3:P2</td>
<td></td>
<td>2:R4</td>
<td>1:M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3:P3</td>
<td></td>
<td>3:R1</td>
<td>3:R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td>3:P4</td>
<td></td>
<td>3:R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>4:P1</td>
<td>4:P2</td>
<td></td>
<td>3:R4</td>
<td>2:M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4:P3</td>
<td></td>
<td>4:R1</td>
<td>4:R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td></td>
<td>4:P4</td>
<td></td>
<td>4:R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>5:P1</td>
<td>5:P2</td>
<td></td>
<td>1:W1</td>
<td>4:R4</td>
<td>3:M1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>5:P3</td>
<td></td>
<td>5:R1</td>
<td>5:R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td></td>
<td>5:P4</td>
<td></td>
<td>5:R3</td>
<td>1:W2</td>
<td>3:M2</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>6:P1</td>
<td>6:P2</td>
<td></td>
<td>2:W1</td>
<td>5:R4</td>
<td>4:M1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>6:P3</td>
<td></td>
<td>6:R1</td>
<td>6:R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td></td>
<td>6:P4</td>
<td></td>
<td>6:R3</td>
<td>2:W2</td>
<td>4:M2</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>7:P1</td>
<td>7:P2</td>
<td></td>
<td>3:W1</td>
<td>6:R4</td>
<td>5:M1</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>7:P3</td>
<td></td>
<td>7:R1</td>
<td>7:R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td></td>
<td>7:P4</td>
<td></td>
<td>3:R2</td>
<td>4:A1</td>
<td>7:B1</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>4:W1</td>
<td></td>
<td>4:W1</td>
<td>7:R4</td>
<td>6:M1</td>
<td>4:S1</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>5:R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>4:W2</td>
<td>6:M2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>5:W1</td>
<td></td>
<td>5:W1</td>
<td>7:M1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>6:M1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>5:W2</td>
<td>7:M2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>6:W1</td>
<td></td>
<td>6:W1</td>
<td>7:M2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>7:A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>6:W2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>7:W1</td>
<td></td>
<td></td>
<td>7:S1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>7:W2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)

Figure 2: (a) Modulo schedule for the example of Figure 1. (b) Record of execution for 7 iterations of the modulo scheduled DO-loop (assuming register renaming).
Figure 3: (a) Code for one iteration of the loop after modulo scheduling. (b) Kernel code after modulo scheduling. (c) The code schema for the modulo scheduled loop.
Figure 3c uses this example DO-loop to demonstrate the abstracted representation of code that we shall use in this paper. Each square represents one stage's worth of code from a single iteration. The letter label in the square indicates the corresponding stage, with A corresponding to stage 0, B to stage 1, and so on. Thus, the set of rectangles in the leftmost column correspond to all the operations in the first iteration (Figure 3a). Each row of squares represents II VLIW instructions. The row of squares that includes the last stage of the first iteration corresponds to the kernel code (Figure 3b). The triangle of squares above the kernel represents the prologue code and the triangle of squares below the kernel represents the epilogue code.

1.3 Overlapped Lifetimes

The code in Figure 3 is incorrect as shown. Consider the operation $t03 = iadd(t03, #4)$ which at time 0 computes a new address value into virtual register t03 (Figure 3a). The lifetime of this value extends to time 12 when it is used for the last time. However, 3 cycles later the same operation is executed again on behalf of the next iteration and will overwrite the previous value in t03 while it is still live yielding an incorrect result. One approach to fixing this problem is to provide some form of register renaming so that successive definitions of t03 actually use distinct registers. We shall define such a scheme in Section 2. It is important to note that conventional hardware renaming schemes are inadequate. Since, successive definitions of t03 are encountered before the uses of the prior definitions, it is impossible even to write correct code for the modulo scheduled loop with the conventional model of register storage.

When no hardware support is available, modulo scheduling is made possible by modulo variable expansion, (MVE), i.e., unrolling the kernel and renaming at compile time the multiple (static) definitions that now exist of each virtual register [5]. The unrolling and renaming prevents successive lifetimes, corresponding to the same loop-variant physical register, from overlapping in time. The minimum degree of unroll, $K_{\text{min}}$, is determined by the longest lifetime among all loop-variants $i$. Assume that each loop variant $i$ has parameters $\text{end}_i$ and $\text{start}_i$ marking the beginning and end of the lifetime. Because iterations are initiated every II cycles, $K_{\text{min}}$ can be calculated as

$$K_{\text{min}} = \max_i \left( \left\lceil \frac{\text{end}_i - \text{start}_i}{\text{II}} \right\rceil \right).$$

In our example, the longest lifetime is 12 cycles corresponding to the definition of t03. For an II of 3, this requires that $K_{\text{min}} = 4$. Every fourth definition of t03 can reuse the same physical register since its previous contents are no longer live. The structure of the code after kernel unrolling is shown in Figure 4. The labels for the squares now include a numerical suffix which specifies which code version is being used. By looking down the columns one can see that there are $K_{\text{min}}$ distinct versions of code for an iteration and that the successive iterations cycle through these four versions. Each version makes uses of different sets of physical registers to avoid over-writing live values. Because of this, the steady state portion of the record of execution repeats only every $K_{\text{min}}$ stages which is why the kernel must be unrolled $K_{\text{min}}$ times.
It may appear that modulo scheduled code can be generated in conformance with the code schemas of Figures 3c or 4. We shall see in Section 3 that this is not the case and that, in fact, considerably more complex schemas are needed if performance is not to be compromised\(^2\). The problem is that with the code schemas of Figures 3c and 4, it is only possible to execute \(i+4\) iterations, where \(i \geq 0\). (Four iterations can be executed by branching from the last stage of the prologue to the first stage of the epilogue. Fewer iterations cannot be executed with the code schema in their current form.) These code schemas have to be augmented if an arbitrary number of iterations is to be executable.

### 1.4 Pre-conditioning of Modulo Scheduled DO-Loops

A solution, that is often employed, is to pre-condition the modulo scheduled loop so that only the appropriate number of iterations remain to be executed at the time the prologue is entered. In general, the code schemas of Figures 3c and 4 can execute only certain numbers of iterations, \(N\), where

\[
N = K*i + (SC-1)
\]

\(^2\) The authors are indirectly aware of at least one computer manufacturer whose attempts to implement modulo scheduling, without having understood this issue, resulted in a compiler which generated incorrect code.
and where $K$ is the degree of unroll, $SC$ is the number of stages in one iteration and $i \geq 0$. When the desired number of iterations, $L$, is not of this form, a conventional, non-software pipelined version of the loop is first executed until the number of remaining iterations is of the above form. At this point, the modulo scheduled code schema is entered with an appropriate trip count. The number of iterations, $M$, in the pre-conditioning loop is given by

$$M = \begin{cases} L, & \text{if } L < SC - 1 \\ [L - (SC - 1)] \mod K, & \text{otherwise.} \end{cases}$$

$$N = L - M$$

These $M$ iterations are executed relatively slowly and the remaining $N$ iterations are executed with the full, modulo scheduled level of performance. Assume that the time taken to execute one iteration of the non-software pipelined, pre-conditioning loop is $SL$ cycles. Then

$$T_{PC} = M \times SL + (N + SC - 1) \times \Pi$$

$$T_{Ideal} = (L + SC - 1) \times \Pi$$

where $T_{PC}$ is the execution time for the pre-conditioned loop and $T_{Ideal}$ is the ideal execution time for the software pipelined loop. The first term in the formula for $T_{PC}$ is the time spent in the pre-conditioning loop and the second term is the time spent in the software pipelined loop. The speedups in the two cases, relative to a non-software pipelined version of the loop are given by

$$S_{PC} = \frac{L \times SL}{M \times SL + (N + SC - 1) \times \Pi}$$

$$S_{Ideal} = \frac{L \times SL}{(L + SC - 1) \times \Pi}$$

The effectiveness of pre-conditioned code is highly dependent upon the nature of the processor architecture. In order to better illustrate this point, we define four processors: P1, P2, P3, P4 (Table 2). Processor P3 is exactly the sample processor of Table 1. Processors P1 and P2 are versions of the sample processor having identical latency but reduced numbers of functional units, while processor P4 is P3 with increased latencies.

A schedule was generated for the example program of Figure 1 for each of the processors in order to help illustrate the relationship between the amount of processor parallelism and the four parameters which determine pre-conditioned code performance. The four parameters are: the initiation interval ($\Pi$), the number of stages ($SC$), the minimum degree of kernel unroll ($K_{min}$) and the schedule length of a single non-overlapped loop iteration ($SL$). Note that $K \geq K_{min}$. In this discussion we assume that $K = K_{min}$. Parameters resulting from schedules for the four processors are shown in Table 3.
Table 2. Definition of Processors P1, P2, P3, P4

<table>
<thead>
<tr>
<th>Processor</th>
<th>Description</th>
<th>Latencies</th>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A single functional unit which executes all operations. Latencies are as in Table 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>Three functional units. An IALU unit executes all integer operations and branches. The memory unit performs loads and stores. The floating point unit executes all floating point operations. Latencies are as in Table 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>The sample processor of Table 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>The sample processor with all latencies doubled.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Results of Scheduling Sample Processors

<table>
<thead>
<tr>
<th>Machine</th>
<th>II</th>
<th>SC</th>
<th>K_{min}</th>
<th>SL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>15</td>
<td>2</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>P3 (sample processor)</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>27</td>
</tr>
</tbody>
</table>

In all four schedules, II was equal to ResMII because each schedule saturates a resource. For the schedule for processor P1, fifteen total operations were scheduled onto a single functional unit. For P2, six memory operations were scheduled onto a single memory unit. For both processors P3 and P4, six memory operations were scheduled onto two memory units. Thus, the ResMII and a resulting II can be justified and as we increase the number of functional units within the processor (P1, P2, P3), the II decreases.

SC represents the length of the software pipeline schedule of a single iteration divided by the II and rounded up to the nearest integer. If we were to assume that the schedule length for a single iteration were held constant, than the effect of reducing II is to increase the number of stages. We can see that this increase in SC indeed occurs as one goes from processor P1 to P2 to P3. As II decreases through the values 15, 6 and 3, SC increases through the values 2, 3 and 5, respectively. The parameter K_{min} can be viewed similarly. If we were to assume that the longest lifetime is constant among the different schedules and is then divided by II to yield K_{min} we will see a similar progression of decreasing II and increasing K_{min}. SC and K_{min} are not strictly inversely proportional to II. As we add functional units, the schedule length and the lifetime lengths do not stay absolutely constant. Processor P4 demonstrates the effects of increasing the latency which is to generally increase schedule lengths and, correspondingly, to increase SC, K_{min}, and SL.
The effects of varying these parameters on the speedups, $T_{PC}$ and $T_{Ideal}$, achieved by the pre-conditioned loop and the ideal case, respectively, are shown in Figure 5. Note that for a machine with little parallelism such as P1 (Figure 5a), pre-conditioning is quite satisfactory because the time lost in non-overlapped pre-conditioning code is small. One reason for this is the small $K_{\text{min}}$ which results in a small value for $M$. Secondly, the small difference between $SL$ and $II$ decreases the benefits of software pipelined execution over non-overlapped execution.

Although pre-conditioning is an acceptable solution for processors with little instruction-level parallelism, in processors with as much parallelism as P3 or P4 (Figures 5c and 5d), the loss in
performance due to pre-conditioning is very significant. In particular, only very large trip counts can guarantee that the loop achieves close to asymptotic performance. For these loops, M is large and so is the difference between SL and II. Furthermore, pre-conditioning is not an option with WHILE-loops. Better alternatives are needed with VLIW processors and aggressively superscalar processors or if general-purpose computation involving WHILE-loops is to be supported. These are the subject of Section 3.

1.5 Modulo Scheduling of WHILE-Loops

In DO-loops, it is possible to decrement and test the count of the remaining iterations in time to either start the next iteration with an initiation interval of II or exit the kernel. This is not always the case in the broader class of loops which we shall refer to in this paper as "WHILE-loops". This is the class of single entry loops with a single control flow back-edge, one or more exits and for which it is not known, at the time that the loop is entered, what the trip count will be. Whether another iteration is to be executed is known somewhere in the middle of the current iteration and the extent of software pipelining is limited by the fact that the next iteration cannot be initiated until this point in time.

Figure 6: A modulo scheduled WHILE-loop (a) without and (b) with speculative execution
Consider the situation if in Figure 3c it is not known until stage C whether another iteration is to be executed. The earliest time that the next iteration could be initiated would be at the end of stage C. The resulting modulo schedule would have an II that is three times as large (Figure 6a). The limiting dependence is the control dependence between a loop-exiting branch operation in one iteration and all of the operations in the next iteration. Assuming hardware support for speculative code motion [11], this control dependence can be relaxed to yield a smaller II and a better modulo schedule [9].

In Figure 6b, the operations in stages A and B of a given iteration, instead of being control dependent on the branch operation from stage C of the previous iteration, have instead been made dependent on the corresponding branch operations from three and two iterations ago, respectively, i.e., they are executed speculatively. This is clear in Figure 6b since stages A and B are executed before or in parallel with stage C of the previous iteration. The remaining stages are scheduled non-speculatively after stage C of the previous iteration. The net result is a schedule that yields the same performance as would be obtained for a DO-loop.

The speculative execution of stages A and B implies that at every instant, after the second stage of the first iteration, we have two iterations that have been initiated speculatively. When the kernel is exited, we can stop executing, and leave unfinished, the two speculative iterations that are in progress at that point. In Figure 6b, this aborted computation is the rightmost two columns of the epilogue which are shown shaded. The code for this is eliminated from the epilogue. In general, if \( \theta \) stages of each iteration are executed speculatively, the rightmost \( \theta \) columns of the epilogue are eliminated and the epilogue length reduces by \( \theta \) stages.

2 Architectural Support for Modulo Scheduling

In this section, we shall describe various architectural features that support the use of fast, compact code for modulo scheduled DO-loops and WHILE-loops. The motivation for their existence as well as the manner in which they are intended to be used is deferred to Section 3.

2.1 Rotating Register Files

A rotating register file is addressed by adding the instruction's register specification field to the contents of the Iteration Control Pointer (ICP) modulo the number of registers in the rotating register file (Figure 7). Special loop control operations, that are described below, decrement the ICP each time a new stage starts. As a result of decrementing the ICP, a new absolute register now corresponds to the register specifier \( i \), and the register that was previously specified as register \( i \) would have to be specified as register \( i+1 \). This allows the lifetime of a value generated in one iteration to overlap the lifetimes of corresponding values generated in previous and subsequent iterations without code replication.

The rotating register file is quite similar in concept to vector registers. Instead of moving the pointer every cycle, it is decremented once per kernel iteration, and instead of having multiple vector registers, they are all pooled into one register file. The use and allocation of the rotating registers is described elsewhere [19]. One version of rotating registers first appeared in the scratchpad register files of the FPS AP-120B and FPS-164 [20].
2.2 Predicated Execution

The Iteration Control Register (ICR) is a rotating register file that stores Boolean values called predicates. An operation is conditionally executed based on the value of the predicate associated with it. For example, the operation \( a = \text{op}(b, c) \) if \( p \) executes if the predicate in the ICR register \( p \) is true (one), and is nullified if the predicate is false (zero). Predicated execution permits the generation of more compact code by conditionally disabling the execution of operations during prologue and epilogue execution. The need to unroll a prologue and epilogue is eliminated, thereby supporting the generation of kernel-only code as described in Section 3.5.

In addition to using predicated execution to support the combining of prologue, kernel, and epilogue code, predicates are also used to enable modulo scheduling of loops containing conditional branches \([8, 10]\). Predicates permit the IF-conversion of the loop body \([21]\), thereby eliminating all branches from the loop body. The resulting branch-free loop body is modulo scheduled. This was the primary motivation for providing predicated execution in the Cydra 5. More recently, limited forms of predicated execution have been incorporated or proposed in other machines \([22, 23]\). In the absence of predicated execution, other techniques must be used which require either multiple versions of code corresponding to the various combinations of branch conditions \([22, 2, 24]\) or restrictions on the extent of overlap between successive iterations \([5]\). Predicated execution is conceptually similar to, but more general than, the use of mode bits in the vector mask register of a vector processor.

2.3 Speculative Execution

Speculative execution consists of executing an operation before it is clear that it should, in fact, be executed. One way of achieving speculative execution is by speculative code motion, i.e., by moving an operation up above the branch that could have directed flow of control away from this operation \([18]\). The main challenge is to report exceptions correctly in the face of speculative execution, i.e., if and only if the exception would have been reported in the non-speculative execution of the program. The hardware support assumed involves having two versions of every operation that can be speculatively executed (one normal opcode and one speculative opcode), and an additional bit in every register to serve as a tag indicating that the register contains an exception.
tag rather than normal data. A detailed description of this hardware support and its use is described elsewhere [11].

2.4 Loop Control Operations

A generalized branch operation (shown in Figure 8a) is used to effect branch control for all loop schemas presented within this report. A macro scheme defines branch semantics in terms of variables which we shall term branch definition parameters. Given an assignment to each of seven branch definition parameters, a specific member of the branch control family is instantiated. A reference to a variable within angle brackets (<>) indicates a reference to a definition (<> :=) which is provided within the 8b. Each variable beginning with # is substituted textually to complete the definition of the operation. A specific branch operation is defined by assigning true or false to the seven branch definition parameters:

- #counted_loop
- #rot_hdw - true indicates use of rotating register hardware
- #pred_hdw - true indicates use of predicate hardware
- #cont_dir - controls continue branch sense (taken vs fall through)
- #ramp_dir - controls ramp down branch sense
- #stop_dir - controls stop branch sense
- #theta - controls first θ stages of loop (disable the decrementing of ESC & LC)

The branch operation either falls through or branches to a single target as provided within a branch target specification. The sense of the branch dictates whether the branch is either taken or it falls through. The generalized branch flowchart reaches one of three terminal branches each with programmable sense. Control over branch sense supports the flexible construction of loop schema. The parameters #cont_dir, #ramp_dir, and #stop_dir independently control the sense of branching in three situations. At the end of each stage of software pipeline execution, a loop must either: 1) continue to initiate new loop iterations (continue), 2) halt the issue of new loop iterations but continue the execution of iterations in process (ramp down) or, 3) halt all work because execution is complete (stop). The three variables #cont_dir, #ramp_dir, and #stop_dir are assigned value true if the operation is to take a branch in the corresponding situation and are assigned false if the operation is to fall through. This parameterization is used to accommodate distinct branch requirements for: forward and backward branches, and differing strategy with respect to flow of control within the prologue, kernel, and epilogue.
<initiate_iter>(true)  <lc_decr>

T

F

<esc_tests>

F

T

Figure 8a The generalized branch operation

Figure 8b The generalized branch operation macro definitions
The `brtop` operation (Figure 8c) has been used as the loop closing operation for DO-loop software pipelines with full hardware support for both rotation and predicates [8]. The `brtop` operation is a member of the family of generalized branch operations where branch definition parameters are selected to indicate that the branch control operation is for a DO-loop, with full hardware support for rotation and predicates, `brtop` branches to continue, `brtop` branches to ramp down, and `brtop` falls through to stop. We use the following assignments of true or false to branch definition parameters to generate a `brtop`.

```plaintext
#counted_loop=true
#rot_hdw=true
#pred_hdw=true
#conc_dir=true
#ramp_dir=true
#stop_dir=false
#theta=false.
```

The `brtop` operation shown in Figure 8c results when the flowchart of Figure 8a is interpreted with these assignments. This can be explained as follows. With the above assignments to branch definition parameters, the following macro substitutions result:

```plaintext
<pred> = "ICR(ICP)"
<new_iter> = "LC > 0"
<initiate_iter>(true) = "if(ESC>0) then ICP=ICP-1; ICR(ICP)=true"
<initiate_iter>(false) = "if(ESC>0) then ICP=ICP-1; ICR(ICP)=false"
<lc_decr> = "LC=LC-1"
<esc_decr> = "ESC=ESC-1"
<branch>(#cont_dir) = "branch"
<branch>(#ramp_dir) = "fall_through"
<branch>(#stop_dir) = "fall_through"
```

The `brtop` operation shown in Figure 8c results when the flowchart of Figure 8a is interpreted with these assignments. This can be explained as follows. With the above assignments to branch definition parameters, the following macro substitutions result:
With these macro substitutions, the general branch of Figure 8a is equivalent to the brtop of Figure 8c. Note that the test "<pred> ∧ <new_iter>" within the generalized operation branch substitutes to "ICR(ICP) ∧ LC>0". We have an extra term "ICR(ICP)" relative to the brtop operation shown in 8c. If initially, ICR(ICP) is true and LC > 0, this test can be simplified to "LC>0" as in Figure 8c. This is true because as we execute consecutive branch operations ICR(ICP) remains true while LC > 0. When LC = 0, ICR(ICP) is conjoined with false and does not affect the conjunction. The term ICR(ICP) is useful within the WHILE-loop.

The brtop will be used within schema 4c described below. The brtop operation is scheduled to execute in the last cycle of a stage within the loop body. The ICP is decremented every loop iteration so that each iteration can reference a different set of registers. The loop counter (LC) which counts the remaining loop iterations is decremented until it reaches zero. Thereafter, the epilogue stage counter (ESC) which counts epilogue stages is decremented until it reaches zero. At this point, the brtop branch is not taken and the loop is exited. The ESC supports the execution of the extra stages of execution required to drain the software pipeline. Here, these counters are initialized with: LC = (number of iterations -1) and, ESC = (number of stages -1).

The brtop operation assigns a Boolean value to the predicate register ICR(ICP) which controls the conditional execution of the next loop iteration. Initially, ICR predicates are cleared except for ICR(ICP) which is set true insuring that the first iteration of the loop is executed. As the LC is decremented, the assignment to ICR(ICP) sets to true the controlling predicate for the next loop iteration. After LC has reached zero, the assignment to predicate ICR(ICP) sets to false the controlling predicate for all subsequent loop iterations, thereby discontinuing the initiation of new iterations. Intuitively, the brtop operation is supposed to find LC = 0 after the last iteration has been initiated, and is supposed to find ESC = LC = 0 when it is time to exit the kernel. The initial value of ESC determines how many additional times the kernel should be executed after LC has become 0.

The wtop operation is used [9] in order to control branching within WHILE-loops. This will be used within schema 4c described below for the WHILE-loop (#counted_loop=false) case. We instantiate this operation through the following assignment to branch definition parameters: #counted_loop=false, #rot_hdw=true, #pred_hdw=true, #cont_dir=true, #ramp_dir=true, #stop_dir=false. In the case of WHILE-loops, the number of iterations is not known at the time the loop is entered and, a loop counter cannot be used as in brtop. Instead of the loop counter, the wtop operation uses two inputs, a Boolean (exit_condition) and a predicate, and produces an output predicate. The output is true only if both the exit_condition is false and the input predicate is true. This ensures that an iteration completes only if the previous iteration completed and the relevant Boolean expression determining the exit condition for the WHILE-loop was evaluated to false. The ESC is used just as in brtop to allow the last few iterations to complete before the branch out of the loop is taken. The ICP is decremented as in brtop so that each iteration references a different set of registers. The wtop tests a predicate and exit_condition and sets a predicate. If rotation is available, wtop references ICR(ICP) however if rotation is unavailable, wtop references explicit source (pred_source) and target (pred_target) predicates.

Table 4 below is used to define the use of branch operations within the five regions of eight software pipeline schemas which are defined in section 3. Each of the software pipeline schemas is separated into no more than five regions: the first θ stages of the prologue, the rest of the prologue,
the first K-1 stages of the unrolled kernel, the final stage of the unrolled kernel, and the epilogue
stages. In all cases, identical branch operations are used throughout each region. The five regions
do not exist within all of the schemas. A "" indicates that the corresponding region does not exist
and no branch operation is required.

Table 4 defines values for five branch definition selectors: #cont_dir, #ramp_dir, #stop_dir,
#pred_hdw, #rot_hdw. Each entry in the table is a five character string where upper case indicates
that a corresponding variable is set true, and lower case indicates that a variable is set false.
Characters used within the strings are listed alongside the corresponding branch definition
parameter below Table 4. Additionally, the branch definition parameter #counted_loop is set to true
for DO-loops and to false for WHILE-loops.

In the discussion of code schemas, we shall be considering situations when either predicated
execution or rotating registers, or both, are absent. As will be defined later, schemas 1 and 1s use
neither predicates nor rotation, schema 2 and 2s use rotation only, schema 3 and 3c use predicates
only, and schema 4 and 4c use both. Corresponding to this we see that "pr" indicating use of
neither predicates nor rotation appears in the schema 1 column of entries. Similarly, the other
columns specify assignments to pred_hdw and rot_hdw. The t (#theta) variable is true only within
the first K-1 stages of the prologue.

We will explain the entries controlling the setting of each branch definition parameter within Table
4. Some of the settings are don’t cares (can be either lower or upper case) and these cases will not
be described. The settings of these branch definition parameters are closely coupled to the schema
definition which will be provided in section three. The c (#cont_dir) variable is used to cause
branches within the final stage of the unrolled kernel to branch back to the top of the loop as the
loop continues to execute. We see that "C" appears in upper case only within the final stage of the
unrolled kernel. The d (ramp_dir) variable is used to take a branch out of a loop as we terminate
the execution of new iterations but prior to ESC reaching zero. We see that "D" appears in upper case
only within the rest of the prologue and within the first K-1 stages of the unrolled kernel for
schema having explicit epilogues. Within these regions we take a branch to ramp down the
computation by branching to the epilogue. Note that schema 3c has no explicit epilogues and does
not branch to ramp down (lower case d) within the first K-1 stages of the unrolled kernel. The s
(stop_dir) variable is used to take a branch out of the loop when all computation has ceased. This
will be particularly important in schema 3c in the first K-1 stages of the unrolled kernel where "S"
appears in upper case causing a loop with no further computation to exit by branching.

In general, the full capability of the branch operation is not required within specific schemas
presented. In some situations, the branch operation which is actually required degenerates to a
"branch and count" or a "noop". When rotating registers are absent, the generalized branch
operation loses that portion of its semantics having to do with the decrementing of the ICP. Also,
the implicit references to the predicate register pointed to by the ICP are replaced by explicit
references to a predicate register. When predicated execution is absent, the semantics relating to
interrogating or setting a predicate register are eliminated. The process of simplifying branch
operations to support each schema is relatively simple and will not be presented.
3 Code Generation Schemas for Modulo Scheduled Loops

When generating code for modulo schedules, two fundamental problems must be overcome. First, a means must be identified to prevent successive lifetimes, corresponding to successive definitions of the same loop-variant virtual register in successive iterations, from being assigned to the same physical register. One way to accomplish this is to use different versions of the code for successive iterations, with each version making use of different registers as a result of modulo variable expansion. The alternative is to use a single version of the code and to provide a rotating register file that dynamically renames the instruction-specified sources and targets, thereby achieving the same objective. Second, a means must be identified to allow subsets of the steady state software pipeline, the kernel, to be executed. This is required in order to handle the first few and last few iterations of the modulo scheduled loop and to handle the case of a smaller number of loop iterations than that corresponding to a single pass through the prologue, kernel and epilogue. It is possible to generate code for modulo scheduled loops for each assumption regarding the choice of code generation technique and available hardware support.

All of the code schemas described below have two things in common. First, it is assumed that there is a branch preceding the code schema that checks that the trip count of a DO-loop is at least one; if not, the entire code schema is branched around. Second, whenever a code schema has more than one control flow path out of it and into the code that follows the modulo scheduled loop, it is to be understood that there exists code on each of these paths which copies the scalar live-out values (if any) into the registers in which the subsequent code expects to find them.
Code generation schemas for modulo scheduled WHILE-loops are similar to those for DO-loops. Nevertheless, there are differences that result from the fact that the trip count cannot be predetermined prior to loop entry. Here, we shall consider only the schemas for code generation, not the details of how to modulo schedule WHILE-loops, which is discussed elsewhere [9]. The WHILE-loops referred to in this section correspond to the do-while construct of the C language with an arbitrary number of exits from the loop. One important distinction from DO-loops is that pre-conditioning is not an option with WHILE-loops.

We shall avoid detailed discussions of the WHILE-loop schemas since in all cases they closely parallel those for DO-loops, but with the following differences.

- The variable #counted_loop is set false so that the branch tests for an exit condition.
- The rightmost \( \theta \) columns of every epilogue (which are shown shaded in the figures) are deleted since these correspond to the unnecessary completion of speculatively initiated iterations. As a result, the length of each complete epilogue decreases by \( \theta \) stages and, the initial value of \( \text{ESC} = \text{SC} - \theta - 1 \).

We describe a number of code generation schema below to indicate a broad variety of approaches to generating code for software pipelines. This is not complete in the sense that a number of variations of these approaches are known to exist, but we feel that the selected schema are particularly instructive.

### 3.1 Code Schema 1: With Only Speculative Support

We first consider, for the DO-loop example of Figure 1, a code generation schema (Figure 9a) which requires no special hardware. Recall that for this example, \( \text{SC} = 5 \) and \( \text{K}_{\min} = 4 \). As before, each square is labeled with a letter identifying the stage and a number identifying the code version (register assignment choice) used. All stages of a single iteration (same column) correspond to the same code version. A single stage of the modulo scheduled code (all the squares in a single row) consists of one stage each (and a different one) from successive iterations. A loop-control branch is executed at the end of every stage of the modulo scheduled code. Arrows indicate taken branches which, typically, signify transfer of control to an epilogue which completes unfinished portions of the iterations that were in execution when the exit branch was taken.

We can divide the code generated with this schema into a prolog, a kernel, multiple partial epilogues and multiple complete epilogues. Since rotating registers are absent, the code schema must include all the code shown in Figure 4 plus additional code to permit an arbitrary number of iterations. The unique prologue is depicted by the topmost triangle of rows with left hand column A1 \( \ldots \) D1. The kernel is the full width parallelogram consisting of \( K_{\min} = 4 \) rows with left hand squares labeled E1, E2, E3, E4. Register lifetime overlap requirements necessitate the kernel be unrolled to yield four copies. The last stage of the kernel contains a \text{btop} operation which, when taken, closes the loop or, when not taken, enters the complete epilogue depicted by the triangle with righthand column B4 \( \ldots \) D4. The rest of the stages in the prologue and kernel contain \text{brtheta}, \text{brpro}, and \text{bruk} operations as indicated in Table 5 which, when taken, lead to various versions of complete or partial epilogues. Complete epilogues are reached by exiting the loop at the end of any of the four kernel stages, or by exiting from the the final prologue stage. Partial epilogues are reached by exiting from any of the earlier (first three) prologue stages.
Figure 9: (a) Code schema 1 (without predicated execution or rotating register files). (b) Code schema 1 after removal of redundant code.
The code schema of Figure 9a can be seen to be redundant. The epilogues reached by branching out of the final prologue stage and by falling out of the final kernel stage are identical and can be merged into a single epilogue. Each of the partial epilogues reached by branching out of one of the earlier stages of the prologue has a final portion which is identical to the final portion of one of the complete epilogues. This final portion of the partial epilogues can be eliminated and replaced by an unconditional branch to the appropriate stage of the appropriate complete epilogue. The resulting code schema, with this redundancy eliminated, is shown in Figure 9b.

Figure 9 also shows the code generation schema for a WHILE-loop in the absence of hardware support. (The shaded squares and the dashed lines should be viewed as absent for the WHILE-loop schema.) This example loop has $SC = 5$, $K_{\text{min}} = 4$, $\theta = 2$. Therefore, the number of epilogue stages $= SC - \theta - 1 = 2$. All of the standard differences listed above, between DO-loop schemas and WHILE-loop schemas, apply. Other aspects of this schema are the same as that for DO-loops. As with DO-loops, normal conditional branches are employed and the ESC is unnecessary because neither rotating registers nor predicates are used.

![Figure 10: Code schema 2 (without predicated execution but with rotating register files)](image)

### 3.2 Code Schema 2: With Rotating Registers Only

The code schema of Figure 10 illustrates the simplification obtained by using rotating register files. Here, rather than achieving renaming through code replication, the register renaming hardware renames registers at the end of each stage. This is effected by a brpro operation at the end of each stage, except at the end of the kernel where a brtop is used. This eliminates the need for multiple versions of the code to differentiate allocation, thereby eliminating the need to replicate the kernel and epilogue code. The code schema of Figure 10 consists of a single complete prologue, a single non-unrolled kernel, a single complete epilogue, and multiple partial epilogues. The partial
epilogues are still required to permit the execution of less than SC-1 iterations. With this code schema, too, the LC is initially be set to one less than the desired trip count and the ESC is initially set to SC-θ-1.

The primary difference in the code schema for the WHILE-loop from that for the DO-loop is that during the first θ stages, there is no branch out of the prologue because the first exit condition has not yet been evaluated. Instead, a brth operation is placed at the end of the first θ stages. The branch definition parameter #counted_loop takes is assigned value false redefining the meaning of the brth, brpro, bruk, and brtop operations used within the schema..

3.3 Code Schemas 1s and 2s: Aggressive Speculation

Aggressive speculative code motion can be used to minimize the length of the epilogue in both DO-loops and WHILE-loops. In particular, if the loop exit branch can be scheduled in the last stage of an iteration, then θ would be equal to SC-1 and the length of the epilogue would be zero. Since θ rows and θ columns of every partial or complete epilogue are deleted from schema 1, all of the epilogues would disappear. θ = SC-1 corresponds to all but the operations in the last stage being executed speculatively. We shall refer to this as code schema 1s. In certain cases, there may be too many operations (such as stores) to fit in the last stage without compromising the II. In such cases, θ would have to be less than SC-1 and some of the epilogues would be present albeit with reduced length. Branch operations used within this schema are identical to those of Schema 1 except that all prologue stages use brth operations, i.e., brpro with #theta=true. Aggressive speculation can readily be applied when rotating register hardware is used in schema 2. Here, however, only a single epilogue is reduced in length.

3.4 Code Schemas 1pc and 2pc: Pre-conditioned Code (for DO-Loops only)

In the absence of predicates, the multiple epilogues of code schemas 1 and 2 can be eliminated, yielding the code schemas in Figure 3c or Figure 4, by pre-conditioning the loop. The pre-conditioned versions of code schemas 1 and 2 will be referred to as code schemas 1pc and 2pc, respectively. Branch closing operations for 1pc and 2pc are identical to those used in schema 1 and 2 however, the preconditioned loop count guarantees that the kernel will always be exited by a fall through the brtop operation leading to a unique epilogue. Recall from Section 1.4 that the number of iterations, M, in the pre-conditioning loop is given by

\[
M = \begin{cases} 
L, & \text{if } L < SC - 1 \\
[L - (SC-1)] \mod K, & \text{otherwise.}
\end{cases}
\]

\[
N = L - M
\]

where L is the desired number of iterations, K is the degree of unroll and SC is the number of stages in one iteration. The remaining N iterations are executed in the modulo scheduled code schema. The LC must be initialized prior to entering the modulo scheduled loop with the value \([N-(SC-1)] \div K. The branch operation at the end of the kernel must decrement the loop counter by 1
each time it is executed (which is every $K \times II$ cycles). No other branch operations are needed in either the prologue or the kernel. Alternatively, the LC may be initialized to $[N-(SC-1)]$ and the branch at the bottom of the kernel must decrement the LC by $K$ each time. (Of course, both alternatives are identical when $K = 1$.)

Figure 11: Code schema 3 (with predicated execution but without rotating register files).

3.5 Code Schema 3: With Predicated Execution Only

Whereas the presence of rotating registers eliminates the need for multiple copies of the kernel and multiple complete epilogues, the presence of predicated execution eliminates the need for the partial epilogues. In Figure 9a one can see that each partial epilogue is identical to the rightmost few columns of the complete epilogue that is just above it. Therefore, the complete epilogue may be executed in place of the partial one if the undesired computation can be disabled using the predicated execution capability. A detailed explanation of how this is accomplished is deferred to Section 3.7. The net result is the code schema of Figure 11 in which the brpro operations branch directly to a complete epilogue rather than first passing through a partial one. Consequently, the partial epilogues may be eliminated. With this code schema, too, the LC must initially be set to one less than the desired trip count and the ESC is initially set to SC-0-1.

The use of predicated execution introduces additional lifetimes, those of the predicates, which must be considered in calculating $K_{min}$. Without rotating register files, this often leads to $K_{min}$ increasing since the predicate computed by the loop-control operation is live for all SC stages of the following iteration. In the example of Figure 1, SC is equal to five; consequently, $K_{min}$ increases from 4 to 5. The increase in code size of the unrolled kernel due to the increase in $K_{min}$ partially offsets the reduction due to the elimination of the partial epilogues.
### 3.6 Code Schema 4: With Rotating Registers and Predicated Execution

The code schema of Figure 12 makes use of both predicates and rotating registers. Starting with code schema 2 (Figure 10), we see once again that each partial epilogue is a subset of that portion of the complete epilogue that precedes the target of the unconditional branch at the end of the partial epilogue. So, rather than executing the partial epilogue one could, instead, execute the complete epilogue with the appropriate number of the leftmost columns disabled by predicates. Once again, the details of how this is achieved are left to Section 3.5, but the result is the code schema shown in Figure 12. All the brpro and bruk operations have as their target the beginning of the (single) epilogue. With this code schema, too, the LC must initially be set to one less than the desired trip count and the ESC is initially set to SC-θ-1.

![Code Schema 4 Diagram](image)

**Figure 12:** Code schema 4 (with predicated execution and rotating register files).

### 3.7 Code Schemas 3c and 4c: Kernel-Only Code

With hardware support in the form of rotating registers and predicated execution, it is not necessary to have explicit code even for a single prologue and epilogue; a single copy of the kernel is sufficient to execute the entire modulo scheduled loop. This is termed **kernel-only code**. Consider the kernel-only code schema depicted in Figure 13a. Every stage of the code schema in Figure 12 is a subset of this kernel-only schema. The prologue and epilogue can be swept out by executing the kernel with the appropriate operations disabled by predicated execution. Since this is a compact version of code schema 4, we shall refer to the schema in Figure 13a as code schema 4c.
Figure 13: (a) Kernel-only code schema - 4c with predicated execution and rotating register files. (b) Kernel code for the kernel-only code schema. (c) Operation of the brtop instruction while executing kernel-only code for 7 iterations of a loop with 5 stages.

The code corresponding to the kernel-only schema is shown in Figure 13b. All operations from the i-th stage are logically grouped by attaching them to the same predicate, specifically, the contents of the ICR register specified by the predicate specifier i relative to the ICP. This is represented in Figure 13b by appending "if pi" to every operation from the i-th stage. This permits...
all operations from a particular stage (of one iteration) to be disabled or enabled independently of
the operations from some other stage (of some other iteration). At every point in time, predicate \( p_0 \)
is the ICR register that is currently pointed to by the ICP. This predicate is set to 1 by the brtop
operation during the prologue and kernel phases (i.e., while the value of the loop counter is greater
than 0) and is set to 0 during the epilogue phase. Because brtop decrements the ICP, a different
physical predicate register is written into every II cycles and, a given predicate value must be
referred to by different predicate specifiers in different stages.

Figure 13c demonstrates the manner in which this is actually effected with the joint use of rotating
registers, predicated execution and the brtop operation. The example assumes that 7 iterations of a
loop of 5 stages is desired. The loop counter, LC, is initialized to 6, one less than the number of
iterations desired. The epilogue stage counter, ESC, is initialized to 4, one less than the number of
stages. Lastly, \( p_0 \) (the ICR location that is currently pointed to by the ICP) is set to 1 and \( p_1 \)
through \( p_4 \) are set to 0. At this point, the kernel-only code is entered. Since only \( p_0 \) is true, only
the operations from the first stage, labeled A, are executed and the rest of the operations are
disabled. At the end of the first trip through the kernel, since the LC is greater than 0, the brtop
operation loops back to the top of the kernel and decrements the LC by 1. It also decrements the
ICP by 1 and, since the ICR is a rotating register file, the true predicate that used to be \( p_0 \) is now
\( p_1 \). Also, because the LC was greater than 0, the new \( p_0 \) is set to 1. During the next trip through
the kernel code, the operations corresponding to the first two stages, A and B, execute since both
\( p_0 \) and \( p_1 \) are true.

This process is repeated with the operations in the i-th stage being executed when the
corresponding predicate, \( p_i \), is 1. Eventually, the brtop operation finds that the LC is 0, but loops
back because the ESC is greater than 0. However, it now decrements the ESC, decrements the ICP
and inserts a 0 in the new \( p_0 \). As a result, the next time around, operations from stage A are not
executed. Finally, when both the LC and ESC are 0, the brtop operation falls through to the code
following the loop. In the process, seven iterations each consisting of five stages have been swept
out by the combined operation of the brtop operation, rotating registers and predicated execution.

The unrolled kernel-only code schema of Figure 14a is the kernel-only equivalent for code schema
3, i.e., when predicated execution is present but there are no rotating register files. We shall refer
to this as code schema 3c. Unlike the previous case, all operations from the same code version, i,
are predicated on the same predicate, \( p_i \), regardless of the stage from which the operation comes.
The operation of this code schema is quite similar to the previous case (Figure 14b). A brtop
operation is used in the last stage of the kernel, and bruk operations are used in the earlier stages.
However, the brtop and bruk operations must be redefined to permit the explicit specification of the
destination predicate instead of it always being \( p_0 \). On each trip through the kernel, the first
through fourth bruk operations set predicates \( p_1 \) through \( p_4 \), respectively, to either true or false.
The brtop operation sets \( p_5 \) to either true or false.

Predicated execution and the loop-control branches are used in much the same way in code
schemas 3 and 4 as they are in schemas 3c and 4c, respectively, even though explicit prologues
and epilogues are provided. When executing the prologue or when the epilogue is executed due to
the brtop operation falling through, the predicates are redundant since only those stages are present
for whom the predicate is true. However, the predicates are required, when an epilogue is entered
from the prologue via a bruk operation, so as to disable those stages that are not part of the partial
epilogue that needs to be executed.
Figure 14: (a) Unrolled kernel-only code schema -3c with predicated execution but without rotating register files. (b) Operation of the brtop instruction while executing unrolled kernel-only code for 7 iterations of a loop with 5 stages.

Two primary benefits result from the fact that schemas 3 and 4 provide explicit prologues and epilogues unlike their kernel-only counterparts. First, the schedules of the prologue and epilogues can be customized and optimized to take advantage of the reduced requirements for resources within the loop startup and loop shutdown phases. Second, code which originates from outside the innermost loop may be percolated into and scheduled in parallel with prologue and epilogue code. This can result in better performance than with kernel-only code, an effect that is more noticeable when the trip count of the innermost loop is small.

As with DO-loops, it is possible to generate kernel-only code and unrolled kernel-only code for WHILE-loops, but only for the portion after the first \( \theta \) stages. The first \( \theta \) stages of the WHILE-loop when the Boolean expression for the first iteration is being computed constitutes the minimal length prologue permissible. The kernel can then generate all the remaining stages of the loop iterations. The ESC must be initialized to SC-\( \theta \)-1.
Figure 15: Distribution of stage count over loop examples.

Figure 16: Distribution of $K_{\text{min}}$ over loop examples.

Figure 17: Distribution of II over loop examples.
4 Measurements on DO-Loops

The actual impact on code size for the various schemas was evaluated by measuring 1358 FORTRAN loops from the SPEC [25] and Perfect Club [26] benchmarks. This was achieved by instrumenting the Cydra 5 FORTRAN 77 compiler to emit information regarding the II, SC, etc., for each loop. Only DO-loops were measured because the Cydra 5 compiler does not recognize loops other than DO-loops. The operation latencies assumed by the compiler were different from those listed in Table 1 in that loads take 13 cycles, floating point and integer multiplications take 2 cycles and floating point and integer additions take 1 cycle.

4.1 Profile of the Sample Loops

The plots of Figures 15, 16, and 17 show the statistical distribution of SC, K\text{min}, and II across the sample loops. The cumulative distribution and density functions for the stage count are shown in Figure 16. The most frequently occurring value of SC was two, with about 35% of the loops falling into this bin. 75% of the loops had an SC of seven or less. Stage counts as high as seventeen were encountered. Large values of SC cause heavy code replication for schemas which control pipeline fill and drain without the use of predicates.

The cumulative distribution and density functions for K\text{min} are shown in Figure 16. The cumulative distribution is shown both with and without the presence of predicate lifetimes. In general, the impact of predicate lifetimes is to skew the cumulative distribution plot to the right by selectively enlarging K\text{min}. However, this effect is almost negligible across this set of loops, most likely because the array address lifetimes are already close to SC in length. About 40% of loops had K\text{min} equal two, while 75% of loops had a K\text{min} of seven or less. K\text{min} as high as eighteen was encountered. Large K\text{min} causes heavy replication for schemas which overlap lifetimes without making use of rotating registers.

The cumulative distribution and density functions for II are shown in Figure 17. About 30% of the loops had an II of 1. This results from a large number of simple vectorizable loops with a small number of operations in the body. About 75% of the loops had an II of sixteen or less. The distribution has been plotted up to an II of forty covering about 95% of the sample loops. One loop, an II of 130 was reached indicating that some very complex loops were encountered.

4.2 Code Size Measurements

Table 4 displays the measured code size, averaged over all 1358 sample loops, for the eight code generation schemas. Schemas are marked +/-pred and +/-rot in order to indicate the use (or non-use) of predicates and rotating register files. The marking "pke" (prologue-kernel-epilogue) indicates that explicit prologue and epilogue code are present, "ko" stands for kernel-only code, and "pre" indicates that loop preconditioning has been performed. Two columns of results are presented, the first reports the average number of operations required by the schema, and the second reports the ratio of the average code size to that of the most compact schema (schema 4c).

Schema 1 (Fig. 7b) requires replicated epilogues and unrolled kernels which, together, contribute to the large increase in code size. This approach requires code size about fourteen times that of schema 4c. The preconditioning which is applied in schema 1pc eliminates the replicated epilogues, thereby saving almost half the code for schema 1.
Table 4: Average code size for the eight code generation schemas.

<table>
<thead>
<tr>
<th>Code Schema</th>
<th>Code Schema Attributes</th>
<th>Average Number of Operations Generated per Loop</th>
<th>Code Size Relative to Schema 4c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-pred -rot pke</td>
<td>255.49</td>
<td>14.03</td>
</tr>
<tr>
<td>1pc</td>
<td>-pred -rot pre</td>
<td>141.26</td>
<td>7.76</td>
</tr>
<tr>
<td>2</td>
<td>-pred +rot pke</td>
<td>85.88</td>
<td>4.72</td>
</tr>
<tr>
<td>2pc</td>
<td>-pred +rot pre</td>
<td>88.64</td>
<td>4.87</td>
</tr>
<tr>
<td>3</td>
<td>+pred -rot pke</td>
<td>258.23</td>
<td>14.18</td>
</tr>
<tr>
<td>3c</td>
<td>+pred -rot ko</td>
<td>76.61</td>
<td>4.21</td>
</tr>
<tr>
<td>4</td>
<td>+pred +rot pke</td>
<td>70.43</td>
<td>3.87</td>
</tr>
<tr>
<td>4c</td>
<td>+pred +rot ko</td>
<td>18.21</td>
<td>1.00</td>
</tr>
</tbody>
</table>

When rotating register files are used as in schema 2 (Fig. 8), code replication to support overlapped lifetimes is no longer required. The only replication performed is to support the filling and draining of the software pipeline. Schema 2 is less than five times as large as schema 4c on the average. Pre-conditioning is applied in schema 2c. Schema 2 already uses rotating registers and thus, epilogues are not replicated. In schema 2c the additional preconditioning does not combine epilogues and results in no overall code savings.

Schema 3 (Fig. 9) also requires replicated epilogues and about fourteen times as much code as kernel-only code. Schema 3c (Fig. 12a) represents an interesting alternative in that while it uses predicates, it makes no use of rotating register files. $K_{min}$ is an important factor in determining the code size of schema 3c because unrolling must be performed to support lifetime overlap in the absence of rotating register files. Schema 3c results in about a four times kernel-only code increase.

When full use is made of both predicates and rotating register files, the single prologue, single stage kernel, single epilogue code schema 4 (Fig. 10) results. No code is required to compensate for early exits. The compact or kernel-only form of this is code schema 4c (Fig. 11). This kernel-only code is fourteen times as compact as non-preconditioned code which makes no use of either rotating registers or predicates. Kernel-only code is almost eight times more compact than preconditioned code which makes no use of either rotating registers or predicates.

The increase in code size for certain schemas over that for schema 4c can have two consequences. Less importantly, the static code size of the program is increased, but it should be noted that it is only the innermost loop code that suffers this dilation; the rest of the program is unaffected. Consequently, the percentage increase of the static size of the program as a whole will be less than that presented in Table 4. Potentially, more important is the effect of code dilation on instruction cache performance. The larger amount of space occupied by the loop schemas will tend to drive other code out of the cache and increase the miss rate. However, the code size data presented above does not bear a direct relationship to overall system performance. For instance, if a particular loop is always executed with the same trip count, only one version of the epilogue will be resident in the cache. The impact of the loop code size increase also depends on the average loop trip count. If the
trip count is very high and the loop fits in the instruction cache, then the cost of the code expansion may be minimal. If the trip count is very small, instruction cache penalties are likely to be higher.

5 Conclusions

Pre-conditioning a modulo scheduled loop, though acceptable on processors with little instruction-level parallelism, leads to significant performance degradation on processors which either are capable of issuing many operations per cycle or are deeply pipelined. In such cases, other code schemas must be employed. The generation of a high performance and correct modulo scheduled code schema is affected by a number of issues: whether or not the loop is a DO-loop and the nature of the hardware support provided. In this paper we have detailed the code generation schemas, both for DO-loops and WHILE-loops, for certain combinations of assumptions regarding hardware support. Four of these schemas (1, 2, 3, and 4) do not compromise performance. Six other schemas (1pc, 2pc, 1s, 2s, 3c and 4c) trade varying amounts of performance for more compact code.

Hardware support for speculative code motion is valuable with all of the modulo scheduled WHILE-loop schemas and, in the case of DO-loops, for Schemas 1s and 2s as well. Predicated execution and rotating register files are needed for Schemas 4 and 4c with both types of loops. Predicated execution is also valuable when modulo scheduling either type of loop if control flow is present in the loop body. Rotating registers are needed for Schemas 2, 2pc, 2s, 4, and 4c with both types of loops.

References


