A Comparative Study of Modulo Scheduling Techniques

Josep M. Codina
Computer Architecture Department
Universitat Politècnica de Catalunya-Barcelona
jmcodina@ac.upc.es

Josep Llosa
Computer Architecture Department
Universitat Politècnica de Catalunya-Barcelona
josepll@ac.upc.es

Antonio González*
Computer Architecture Department
Universitat Politècnica de Catalunya-Barcelona
antonio@ac.upc.es

ABSTRACT
Modulo Scheduling is an instruction scheduling technique that is used by many current compilers. Different approaches have been proposed in the past but there is not a quantitative comparison among them, using the same compiling platform, benchmarks and architectures.

This paper presents a performance comparison of the most relevant Modulo Scheduling techniques, based on a detailed quantitative evaluation of them. The results point out which are the most effective techniques for different architectures, which is useful for compiler designers when choosing the most appropriate technique for a particular processor architecture.

Categories and Subject Descriptors
C.1.1 [Processor Architectures]: Single Data Stream Architectures—Pipeline processors, RISC/CISC, VLIW architectures; D.3.4 [Programming Languages]: Processors—Code generation, Compilers.

General Terms
Algorithms, Measurement, Performance, Experimentation, Languages.

Keywords
Modulo scheduling, instruction scheduling, comparative study, quantitative evaluation, instruction level parallelism, architectures.

1. INTRODUCTION
Software pipelining [9] is a family of instruction scheduling techniques that exploit instruction level parallelism (ILP) in loops by overlapping the execution of successive iterations. There are two main approaches to software pipelining: Move-then-scheduling [11, 13, 18, 12] techniques (also known as code motion techniques) obtain a software pipelined loop by moving operations, one by one, across the back-edge of the loop; and Schedule-then-move techniques, which instead of transforming an existing schedule into a better one, directly construct the final schedule from scratch. Additionally, there are two families of Schedule-then-move techniques: Unroll based scheduling techniques [1, 2, 22, 3, 24, 4, 8], which simultaneously unroll and schedule the loop; and Modulo Scheduling techniques [25] that directly construct a schedule for an iteration in such a manner that when the same schedule is repeated at constant intervals there are no resources or dependence conflicts. Modulo scheduling is the most popular approach to implementing software pipelining.

There are two alternative approaches to constructing modulo schedules: techniques that search in all the solution space looking for an optimal schedule, and heuristic-based techniques. The former can be implemented using a mathematical description of the scheduling objectives and constraints, and can be stated as an Integer Linear Programming problem [23, 16, 14]. Alternatively, the former approach can also be implemented by enumerating all possible solutions and choosing the most effective one [6]. These optimal approaches require an extremely large computation time since finding the optimal schedule is an NP-complete problem, and may not be practical for production compilers. On the other hand, heuristic-based techniques do not guarantee optimal solutions but have much lower computing requirements.

Heuristic-based modulo scheduling techniques have been implemented in many production compilers and there are many proposals in the literature that rely on different heuristics. However, compiler writers have a difficult task when choosing a modulo scheduling technique because there is not an assessment of all of them using the same metrics, compiling platform, benchmarks and architectures. This work tries to cover this gap by presenting a detailed quantitative assessment of the most relevant modulo scheduling techniques:

- Iterative Modulo Scheduling (IMS) [27].
- Swing Modulo Scheduling (SMS) [29].
- Slack Modulo Scheduling (Slack) [17].
- Integrated Register-Sensitive Iterative Software pipelining (IRIS) [10].
Find $M_{II}$
And set $II = M_{II}$

Look for a schedule

If found?
Yes

Increase the $II$

No

Figure 1: Basic scheme of Modulo Scheduling techniques.

For all of them we also evaluate the benefits of the Stage Scheduling [15], which is an instruction scheduling technique that reduces the register requirements of schedules produced by any modulo scheduling approach.

The results show that in terms of exploited parallelism, all the above techniques are very effective for all considered architectures, producing results that are very close to the optimum. A comparison of the different techniques shows that the exploited parallelism is quite similar for architectures with low and medium complexity, whereas IMS is slightly better than the others for very complex architectures. On the other hand, SMS achieves always the best performance regarding register pressure for all considered architectures. Slack is very close to SMS in register pressure, whereas IMS and IRIS have a significantly lower performance, even when Stage Scheduling is applied. Finally, SMS is the approach that spends less computing time for performing the scheduling.

The rest of the paper is organized as follows. Section 2 presents an overview of the most relevant Modulo Scheduling techniques. The benchmarks, compiling platform, architectures, and performance metrics used in this work are described in Section 3. Section 4 presents the comparison of the different Modulo Scheduling techniques. Finally, Section 5 summarizes the main conclusions of this work.

2. MODULO SCHEDULING TECHNIQUES

Heuristic-based Modulo Scheduling is a family of Software Pipelining techniques that produce effective schedules with a relatively small compilation time. Alternative modulo scheduling techniques differ in the heuristics they use. In this section, we present a brief overview of the most relevant Modulo Scheduling techniques, which are later evaluated. For more details on these techniques, the interested reader is referred to the cited papers.

2.1 A Basic Modulo Scheduling Scheme

Figure 1 shows the basic scheme of Modulo Scheduling techniques. Based on this scheme, we can define alternative techniques by using different approaches in each step. Modulo Scheduling techniques take as input the loop to be scheduled, represented by its data dependence graph, and a description of the architecture, and produce a schedule for this loop. The main steps involved in computing a schedule are the following:

- **Computing the Minimum Initiation Interval.**
  The algorithm starts by computing a lower bound on the Initiation Interval ($II$), which is called the Minimum Initiation Interval ($M_{II}$). The $II$ is the number of cycles between the initiation of two consecutive iterations of the loop. The lower the $II$, the higher the amount of parallelism that is exploited. The $M_{II}$ depends on the resources used by each loop iteration and the recurrences in the dependence graph [27].

- **Scheduling step.** This is the most important step of the technique. The main differences among alternative techniques are found in this step. The scheduling step computes a cycle to schedule each operation, and then, produces a code that is laid out accordingly. The main issues involved in the scheduling step are the following:
  - Order in which the operations are handled.
  - Heuristics to find a feasible cycle for each operation.
  - The use of backtracking. The scheduling step is usually an iterative step that builds the schedule progressively, adding instructions to a partial schedule. Sometimes the scheduler reaches a partial schedule that does not allow to include the remaining instructions. In those cases, there are two alternative solutions: increasing the $II$ and trying to find a schedule for the new value of the $II$ [20], or applying backtracking [27, 17, 10]. Backtracking involves unscheduling some operations of the partial schedule in order to schedule them again in different slots. Techniques that include backtracking are also called iterative techniques. The amount of backtracking has to be limited; otherwise, the algorithm might end up exploring all possible alternatives, which would have an exponential complexity. In addition, iterative algorithms are not guaranteed to find all possible alternatives as they may indefinitely cycle through a subset of all possible modulo schedules. The amount of backtracking is usually limited by the $\text{BudgetRatio}$ [27], which is a parameter that indicates the ratio of the maximum number of operation scheduling steps attempted (before trying a larger $II$) to the number of operations in the loop.

- **Increasing the $II$.** Increasing the $II$ reduces the parallelism but increases the number of feasible slots for each operation. Thus, the larger the $II$, the more likely it is to find a feasible schedule. Note that increasing the $II$ beyond the schedule length of a single iteration does not make sense, since the performance would be lower than that achieved by the non-pipelined loop. We provide statistics about the percentage of non-pipelined loops due to this cause for each technique.

2.2 Most Relevant Techniques

In this section we review the most relevant modulo scheduling techniques. These techniques can be applied to loops with a single basic block in the loop body. The use of inlining [21] and if-conversion [5] significantly increases the
number of loops that can be handled. These loops typically account for the majority of the execution time in numeric applications, as reported in Section 4.

2.2.1 Iterative Modulo Scheduling

The Iterative Modulo Scheduling (IMS) algorithm was proposed by Rau [27]. The main feature of the algorithm is its iterative nature in the sense that each operation can be scheduled and unscheduled at several time slots before a suitable slot is found. IMS orders the nodes based on a height-based priority function and schedules each operation as early as possible taking into account suitable time slots in the partial schedule. However, if eventually the scheduler does not find any feasible slot for an operation, some operations are unscheduled in order to create some feasible slot for it. This backtracking feature is key to the performance of the IMS algorithm. On the other hand, IMS does not include any particular heuristic to minimize the register requirements.

2.2.2 Slack Modulo Scheduling

The Slack Modulo Scheduling (Slack) algorithm was proposed by Huff [17]. The main idea of the algorithm is the use of a bidirectional scheduling approach, instead of the traditional unidirectional top-down or bottom-up framework. Another critical issue of the algorithm is the order in which instructions are scheduled. Instead of a height-based priority order such as the one used by the IMS, nodes are scheduled according to their slack. The slack of a non-scheduled instruction with respect to a partial schedule is a measure of the freedom that this instruction would have if it was inserted in the partial schedule. Slack schedules each operation in such a way that the lifetimes corresponding to its inputs and outputs are minimized. In this way, the register requirements are reduced. Slack is an iterative technique. When the scheduler does not find a feasible slot for an instruction, some of the previously scheduled operations are ejected from the partial schedule.

2.2.3 Integrated Register-Sensitive Iterative Software Pipelining

The Integrated Register-sensitive Iterative Software pipelining (IRSP) scheme was proposed by Dani, Ramanan and Govindarajan [10]. This technique results from modifying the IMS scheme to integrate some of the heuristics of the Stage scheduling (see Section 2.2.5) in the scheduling step, instead of applying them as a post-pass. These heuristics are basically used to decide whether each particular operation is scheduled as early as possible or as late as possible, with the objective of reducing register pressure.

2.2.4 Swing Modulo Scheduling

The Swing Modulo Scheduling (SMS) scheme was proposed by Lloa, González, Ayguadé and Valero [20]. It is a heuristic technique that produces very effective schedules with a low computational cost, since it does not include backtracking, unlike the other techniques considered in this work. Its high performance relies on a relatively sophisticated ordering of the nodes.

SMS orders the nodes taking into account the RecMII of the recurrence to which each node belongs (if any), and the criticality of the path to which the node belongs as a secondary factor. Besides, the ordering guarantees for most of the nodes that the list of nodes that are located before it in the ordered list only contains predecessors or successors of it but not both. This property is leveraged by the scheduling step to achieve a very low register pressure.

2.2.5 Stage Scheduling

The Stage Scheduling was proposed by Eichenberger and Davidson [15]. Stage Scheduling is not a modulo scheduling algorithm but a set of heuristics that try to reduce the register requirements of any given modulo schedule by shifting instructions by multiples of II cycles. It is applied as a post-pass of a modulo scheduler. We analyze its effectiveness for all the above modulo scheduling techniques, by applying the transformation 3UP+RSS, which the authors reported to be the one with better performance.

3. PERFORMANCE STATISTICS

In this section we present the benchmarks, compiling platform, architectures, and performance metrics used in this work.

3.1 Benchmarks and Compiling Platform

The Module Scheduling techniques studied in this work have been included in the research compiling platform described in [7]. They have been implemented in C++ using the LEDA libraries [19]. For the evaluation we used all the innermost loops of the Perfect Club benchmark suite and SPECfp95 benchmark suite that have neither subroutine calls nor conditional exits. Loops with conditional structures in their bodies have been IF-converted [5] into a single basic block loop. A total of 1936 loops have been scheduled: 1258 loops from the Perfect Club and 678 loops from the SPECfp95 which represent 78 and 95 percent of the total execution time of each suite respectively.

3.2 Architectures

We have studied the performance of the Module Scheduling techniques using several processor architectures. In this paper, we report results for three of these architectures: a low complexity architecture, a medium complexity architecture and a complex architecture, in terms of the reservation tables of operations. The complexity of the architectures increases gradually in order to evaluate the behavior of the different modulo scheduling techniques for a broad range of architecture classes.

Low Complexity Architecture. For the low complexity architecture we have assumed fully pipelined operations with two memory buses, three integer functional units and three floating point functional units. We have also assumed that the issue width is eight, so that this parameter does not restrict the number of initiated operations per cycle. For this architecture we have considered that we have enough access ports to the register file so that this resource never constraints the scheduling of operations.

Medium Complexity Architecture. For the medium complexity architecture we have assumed fully pipelined operations with two memory buses, two integer functional units and two floating point functional units. We have also assumed that the number of issued operations per cycle is
Table 1: Latency of the operations for the three processor architectures.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Low complexity</th>
<th>Medium complexity</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>int +, -, comp</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>int *</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>int /, mod, sqrt</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>fp +, -, comp</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>fp *</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>fp /, mod, sqrt</td>
<td>18</td>
<td>18</td>
<td>20</td>
</tr>
</tbody>
</table>

3.3 Performance Metrics

The performance metrics that will be used to compare the different modulo scheduling techniques have been grouped in four different categories:

- Effectiveness and cost metrics.
  1. Number of non-scheduled loops. This metric shows how many loops are not pipelined because the technique has failed to find a valid schedule with an II smaller than the schedule length of the non-pipelined loop. If a technique fails to schedule many loops, its applicability scope may be too limited.
  2. Percentage of non-scheduled operations. This is a metric complementary to the previous one that shows how many individual operations are in non-scheduled loops.
  3. Total scheduling time. This metric shows the time spent scheduling all the loops. A low cost is desirable for many production compilers. Note that techniques that fail to schedule some loops will spend a significant amount of time exploring all the possible IIs until the schedule length of the non-pipelined loop is reached.
  4. Scheduling time for the scheduled loops. This metric shows the scheduling time for the loops that have been successfully pipelined by all techniques. It allows a comparison of the scheduling time for all techniques using the same workload.

- Parallelism metrics. This group of metrics shows the effectiveness of the techniques to exploit ILP. In order to make a fair comparison only those loops that have been successfully scheduled by all techniques are considered.
  1. Sum of MII for all scheduled loops. The MII is the lower bound on the II for a given architecture and loop. This metric is the same for all techniques for a given architecture.
  2. Sum of II for all scheduled loops. This metric gives an idea of how much parallelism is obtained. The lower this number more parallelism is extracted.
  3. Average (II/MII) for all scheduled loops. This metric gives an idea of the optimality of the techniques in terms of parallelism. The closer to one this number is, the closer to the optimal II the scheduled loops are.
  4. Number of scheduled loops such that II > MII. These are the loops for which we cannot guarantee that the schedule is optimal in terms of parallelism.

- Register pressure metrics. Although a main objective of modulo scheduling is to obtain the minimum II, register pressure is another factor that is critical for performance. In general software pipelining tends to result in high register requirements. If the number of required registers is higher than the number of available registers in the processor some actions have to be taken in order to produce a valid schedule with the available number of registers [26]. Some options are scheduling the loop with an increased II or the addition of spill code [28]. Both alternatives have a negative effect on the final performance. The following metrics are used to measure register pressure taking into account loop-variant and loop-invariant values, and consider only the scheduled loops, unless otherwise stated.
  1. Sum of register requirements. This metric reflects overall register pressure.
  2. Sum of register requirements for all loops scheduled with the same II by all techniques. This is the metric that best reflects the ability of the schedulers to reduce the register pressure since the register requirements usually increase when the II decreases.
  3. Number of loops that require more than 32 registers. This indicates how many loops can have its performance degraded if only 32 registers are available in the processor.
4. Number of loops that require more than 64 registers. This indicates how many loops can have its performance degraded if only 64 registers are available in the processor.

5. Maximum number of registers required by a loop.

6. Sum of MaxLive / Sum of MinAvg. MaxLive is an accurate estimation of a schedule register requirements due to loop-variants (register requirements of loop-invariants are independent of the schedule). MinAvg is a schedule independent lower-bound of the loop-variant register requirements for a given \( II \). Therefore, this metric tries to measure how optimal are the techniques regarding register requirements. Details about the computation of MaxLive and MinAvg can be found in [17].

- Code size and execution time metrics.

1. Sum of stages for all scheduled loops. This metric reflects the length of the prolog and epilogue parts [27], which influences the code size.

2. Sum of stages for all loops scheduled with the same \( II \) for all techniques. The number of stages of a schedule usually decreases when the \( II \) increases. This metric measures the effectiveness of each technique regarding the size of the prolog and epilogue, avoiding the effects due to different \( IIs \).

3. Total number of execution cycles for all scheduled loops. This metric has been computed by considering the \( II \), the number of stages, the number of iterations each loop performs, and the number of times the loop is executed. It is a good estimation of final performance of the code without considering register requirements. Note that when a limited number of registers is considered, performance will be degraded for some loops and the techniques with less register requirements will be benefited.

3.4 Study of the BudgetRatio

The performance of iterative techniques such as IMS, Slack and IRIS is very sensitive to the BudgetRatio parameter. Thus, for each considered architecture, we first study the influence of the BudgetRatio parameter. We have explored four different values for the BudgetRatio: 1, 2.5, 5 and 10. In order to choose the most appropriate BudgetRatio we take into account effectiveness, performance and cost factors.

Effectiveness. There are some cases in which a technique cannot find a legal schedule for a given loop for any possible value of the \( II \). We measure the effectiveness of a Modulo Scheduling technique by the number of operations in loops that have not been scheduled. In particular, we measure the ratio of operations in unsuccessful loops to the total number of operations in candidate loops.

Performance. The performance of a Modulo Scheduled loop depends on several factors. The most important factors are the \( II \) and the register requirements. We have observed that the register pressure shows almost no variation with the BudgetRatio (in fact it increases slightly due to lower \( IIIs \)). Therefore, we will just quantify the influence of the BudgetRatio on the performance by measuring the \( II \) to \( MII \) ratio.

Cost. We measure the cost by the time spent by the scheduler. In general, a larger BudgetRatio increases the cost because more operations are scheduled and unscheduled before trying a new \( II \). However, in some cases, a larger BudgetRatio allows to find a valid schedule with a smaller \( II \) and may result in a lower number of operation scheduling steps, and therefore in a lower cost.

4. COMPARISON OF THE TECHNIQUES

In this section we compare the performance of the different modulo scheduling techniques for the three processor architectures. For each architecture, we first choose the BudgetRatio for the iterative techniques as the best tradeoff among effectiveness, performance and cost. Finally, we compare each modulo scheduling technique taking into account all the metrics described in Section 3.3.

4.1 Low Complexity Architecture

Figure 2.a shows the percentage of non-scheduled operations as a function of the BudgetRatio for the low complexity architecture. Both IMS and IRIS find a legal schedule for all loops of the benchmarks, for all considered values of the BudgetRatio. On the other hand, Slack cannot schedule all loops for any BudgetRatio we tried. Increasing the BudgetRatio tends to decrease the number of non-scheduled loops, and therefore the percentage of non-scheduled operations. For instance, when using a value of 10 almost all operations can be scheduled, and there is just one non-scheduled loop. However, a BudgetRatio of 2.5 achieves a very close performance in terms of scheduled operations.

Figure 2.b shows the \( II \) to \( MII \) ratio. All the iterative techniques follow a similar trend when the BudgetRatio is varied. We can observe that a BudgetRatio of 2.5 is enough in order to have a reasonable \( II \) to \( MII \) ratio.

Figure 2.c shows the total scheduling time as a function of the BudgetRatio. IMS and IRIS require very low scheduling times for all BudgetRatio values. On the other hand, Slack consumes much more computing time due to a more frequent use of backtracking. IRIS requires more time than IMS, but both of them remain almost steady for the BudgetRatios explored. Slack behaves in a different and unexpected way. It shows an oscillating behavior making it difficult to relate BudgetRatio with scheduling time. For BudgetRatio = 1 it fails to find a valid schedule for 238 loops, which requires increasing the \( II \) and rescheduling the loops too many times. For BudgetRatio = 2.5 the scheduling time is smaller because it finds a valid schedule for more loops. When the BudgetRatio is 5, the scheduling time increases again due to the extra backtracking. Finally, for BudgetRatio = 10 the scheduling time decreases again due to the lower number of non-scheduled loops. We may expect that for larger BudgetRatios the scheduling time would increase significantly because the additional backtracking would not hardly improve the schedules.

From the analysis of these factors we selected a BudgetRatio of 10 as the best trade-off among effectiveness, performance and cost.

Table 2 shows the effectiveness and cost statistics. In particular, the different rows show the number of scheduled loops, the percentage of non-scheduled operations, the time
Figure 2: Influence of the BudgetRatio on the iterative techniques for the simple architecture: a) percentage of non-scheduled operations, b) sum of II / sum of MII and c) total scheduling time.

<table>
<thead>
<tr>
<th>Metric</th>
<th>IMS</th>
<th>IMS+ST</th>
<th>SMS</th>
<th>SMS+ST</th>
<th>IRIS</th>
<th>IRIS+ST</th>
<th>SLACK</th>
<th>SLACK+ST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Effectiveness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># non scheduled loops</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>% ops no sched iteration</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0.11%</td>
<td>0.11%</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total time</td>
<td>14.8s</td>
<td>427.7s</td>
<td>11.7s</td>
<td>451.7s</td>
<td>55.3s</td>
<td>468.5s</td>
<td>119.8s</td>
<td>446.6s</td>
</tr>
<tr>
<td>Sched loops time</td>
<td>14.6s</td>
<td>337.6s</td>
<td>11.5s</td>
<td>359.8s</td>
<td>50.4s</td>
<td>381.6s</td>
<td>39.1s</td>
<td>366.2s</td>
</tr>
<tr>
<td><strong>Parallelism</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sum ) MI</td>
<td>11952</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sum ) II</td>
<td>12917</td>
<td>11965</td>
<td>12050</td>
<td>12041</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average (II/MII)</td>
<td>1.0027</td>
<td>1.0062</td>
<td>1.0035</td>
<td>1.0021</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II &gt; MII</td>
<td>43</td>
<td>9</td>
<td>58</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register pressure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sum ) registers</td>
<td>44805</td>
<td>41835</td>
<td>36766</td>
<td>36413</td>
<td>43210</td>
<td>40440</td>
<td>37213</td>
<td>36697</td>
</tr>
<tr>
<td>( \sum ) registers = II</td>
<td>40130</td>
<td>37453</td>
<td>32794</td>
<td>32466</td>
<td>38842</td>
<td>36337</td>
<td>33200</td>
<td>32742</td>
</tr>
<tr>
<td># loops &gt; 32</td>
<td>493</td>
<td>437</td>
<td>360</td>
<td>352</td>
<td>476</td>
<td>420</td>
<td>371</td>
<td>363</td>
</tr>
<tr>
<td># loops &gt; 64</td>
<td>138</td>
<td>114</td>
<td>58</td>
<td>56</td>
<td>130</td>
<td>105</td>
<td>69</td>
<td>64</td>
</tr>
<tr>
<td>Max # registers</td>
<td>188</td>
<td>188</td>
<td>193</td>
<td>193</td>
<td>200</td>
<td>200</td>
<td>199</td>
<td>199</td>
</tr>
<tr>
<td>MaxLive/MinAvg</td>
<td>1.7704</td>
<td>1.6212</td>
<td>1.3674</td>
<td>1.3457</td>
<td>1.6948</td>
<td>1.5553</td>
<td>1.3903</td>
<td>1.3643</td>
</tr>
<tr>
<td>Code size and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sum ) stages</td>
<td>10214</td>
<td>10214</td>
<td>10360</td>
<td>10344</td>
<td>10029</td>
<td>10024</td>
<td>10385</td>
<td>10386</td>
</tr>
<tr>
<td>( \sum ) stages = II</td>
<td>9722</td>
<td>9722</td>
<td>9870</td>
<td>9854</td>
<td>9865</td>
<td>9839</td>
<td>9909</td>
<td>9909</td>
</tr>
<tr>
<td>Execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total cycles (in millions)</td>
<td>28431</td>
<td>28431</td>
<td>28227</td>
<td>28226</td>
<td>28558</td>
<td>28558</td>
<td>28631</td>
<td>28631</td>
</tr>
</tbody>
</table>

Table 2: Effectiveness, cost, parallelism, register pressure, code size and execution time for low complexity architecture.

to schedule all the loops, and finally the time to schedule the loops for which a legal schedule can be found by all techniques. Each Module Scheduling technique is considered both alone and followed by the Stage scheduling technique (+ST columns in the table).

IMS, SMS and IRIS can schedule all 1936 loops. On the other hand, Slack fails to schedule just 1 loop which represents slightly more than 0.1% of the operations. Note also that Stage does not affect the number of scheduled loops since it is a post-pass step applied to a valid schedule.

In terms of total scheduling time, Slack requires much more time than the others. If only loops with a valid schedule are considered, IRIS requires a scheduling time similar to Slack. This shows that Slack spends a significant amount of time for a single loop for which it finally fails to generate a valid schedule. SMS is the fastest technique although IMS is almost as fast despite requiring backtracking. Finally note that when Stage is applied, the scheduling time increases by about one order of magnitude for all techniques.

Table 2 shows the capability of the techniques to exploit instruction-level parallelism. In particular, the rows show the sum of MII, the sum of II, the average II/MII, and the number of loops for which II > MII. These numbers show that SMS is the technique that obtains the best II. It only needs to increase the II beyond MII for 9 loops out of 1936. Thus, it finds optimal schedules in terms of II for at least the remaining 1925 loops. The other techniques are also very effective: the II is not higher than 1% of the lower bound (MII) for every technique. Note also that applying Stage does not influence the II since it only shifts operations from one stage to another.

Table 2 shows the performance of the techniques in terms of register requirements. In particular the rows show: the sum of registers, the sum of registers of those loops for which all techniques obtain the same II, the number of loops requiring more than 32 registers, the number of loops requiring more than 64 registers, the maximum number of registers required by a loop, and the MaxLive to MinAverage ratio. These numbers show that SMS and Slack are particularly effective in order to obtain schedules with low register requirements. SMS is even more effective than Slack since it obtains schedules with slightly less register requirements and
lower IIs. This can be observed by looking at the register requirements for loops with the same II and the MaxLive to MinAverage ratio. SMS also produces schedules requiring more than 32 and 64 registers for a lower number of loops. IRIS, despite being a register sensitive technique, has still high register requirements and it is only better than IMS (a technique that does not consider register requirements). Finally, applying Stage is quite effective for both IMS and IRIS, but has a small effect for SMS and Slack. Note also that both SMS and Slack without Stage are more effective than IMS and IRIS after applying Stage.

Finally, Table 2 shows the effect of the techniques on code size and execution time by showing the number of stages for all loops, the number of stages of the loops that have the same II for all techniques and the number of execution cycles (in millions) of the resulting codes. IMS is the most effective technique to reduce the number of stages. Note also that Stage sometimes produces a slight reduction in the number of stages since it changes the stage where an operation is executed. Finally SMS is the best approach in terms of execution cycles.

We can conclude that for simple architectures SMS is the most effective technique since it is the fastest one, can schedule all loops, generates the schedules with the smallest IIs, and the resulting codes execute in less cycles. Besides, it requires less registers and therefore the codes will degrade less due to the addition of spill code. A Stage post-pass after SMS is not justified unless the scheduling time is irrelevant and the tiny improvement in register requirements is really critical.

4.2 Medium Complexity Architecture

Figure 3.a shows the percentage of non-scheduled operations as a function of the BudgetRatio for the medium complexity architecture. IMS and IRIS find a legal schedule for all loops for any BudgetRatio. Slack, however, fails to schedule some loops for any BudgetRatio. Increasing the BudgetRatio tends to decrease the number of non-scheduled loops, and therefore the percentage of non-scheduled operations. For instance, using a value of 10, Slack schedules almost all operations, failing to schedule only two loops. Choosing a BudgetRatio of 5 or higher will be reasonable from the point of view of non-scheduled operations. Figure 3.b shows the II to MII ratio. Again the results improve with larger BudgetRatios. However, the improvements are quite small for BudgetRatios larger than 2.5. Finally, Figure 3.c shows the scheduling time for the three iterative techniques as a function of the BudgetRatio. IMS and IRIS follow a similar trend (IMS being almost twice as fast). Both techniques increase scheduling time with the BudgetRatio. Slack shows an opposite trend, it decreases scheduling time with BudgetRatio, especially for a BudgetRatio between 1 and 5. Overall, a BudgetRatio of 5 seems a good trade-off among effectiveness, performance and cost, and is the value chosen for the following experiments.

In terms of effectiveness, Table 3 shows that IMS, SMS and IRIS schedule all 1936 loops. On the other hand, Slack fails to schedule 9 loops that represent 0.5% of the operations. In terms of scheduling time, Table 3 shows that Slack is again the slowest technique. A significant part of this time is spent in the loops that it is unable to schedule (note that only 76 seconds are spent by all loops but 9, while 206 seconds are spent in total). The other three techniques are much faster, but there are significant differences among them. IMS requires less than half the time of IRIS and SMS requires less than half the time of IMS. The lack of backtracking of SMS makes it again the fastest technique and, unlike the previous architecture, by a noticeable difference. Note again that the use of Stage as a post-pass strongly increases the scheduling time and results in a similar computing time for all techniques.

Table 3 shows that SMS is the best approach in terms of exploited parallelism for this architecture. The number of loops in which the technique fails to obtain a schedule with II = MII is 118. IMS is very close to SMS in terms of II while IRIS and Slack obtain slightly worse results.

Table 3 shows that SMS and Slack obtain the best results in terms of register requirements, SMS being slightly better. The only metric for which SMS is much worse than the other techniques is the maximum number of registers, which shows that there can be pathological cases in terms of register requirements. IMS and IRIS obtain significantly worse results in terms of register requirements. However, Slack can decrease the register requirements of these techniques, reducing the difference between them and SMS or Slack. Note again that applying Stage after SMS or Slack has almost no effect on the register requirements whereas it requires a large computing cost (see Table 3).
<table>
<thead>
<tr>
<th>Metric</th>
<th>IMS</th>
<th>IMS+ST</th>
<th>SMS</th>
<th>SMS+ST</th>
<th>IRIS</th>
<th>IRIS+ST</th>
<th>SLACK</th>
<th>SLACK+ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effectiveness and Cost</td>
<td># non scheduled loops</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>% ops non scheduled</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>total time</td>
<td>31.2s</td>
<td>473.2s</td>
<td>14.3s</td>
<td>443.0s</td>
<td>70.6s</td>
<td>507.4s</td>
<td>206.3s</td>
</tr>
<tr>
<td></td>
<td>sched loops time</td>
<td>28.0s</td>
<td>364.5s</td>
<td>13.5s</td>
<td>342.8s</td>
<td>65.4s</td>
<td>400.7s</td>
<td>76.3s</td>
</tr>
<tr>
<td>Parallelism</td>
<td>( y_{M1} )</td>
<td>14667</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( y_{II} )</td>
<td>14885</td>
<td>14843</td>
<td>14973</td>
<td>15307</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Average (( y_{M1} / y_{II} ))</td>
<td>1.0082</td>
<td>1.0091</td>
<td>1.0090</td>
<td>1.0132</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register pressure</td>
<td># registers</td>
<td>37386</td>
<td>35578</td>
<td>31417</td>
<td>31151</td>
<td>35614</td>
<td>34019</td>
<td>31705</td>
</tr>
<tr>
<td></td>
<td># registers = ( y_{II} )</td>
<td>24626</td>
<td>23579</td>
<td>20656</td>
<td>20525</td>
<td>23625</td>
<td>22692</td>
<td>21037</td>
</tr>
<tr>
<td></td>
<td># loops &gt; 32</td>
<td>361</td>
<td>311</td>
<td>214</td>
<td>207</td>
<td>337</td>
<td>283</td>
<td>237</td>
</tr>
<tr>
<td></td>
<td># loops &gt; 64</td>
<td>61</td>
<td>43</td>
<td>33</td>
<td>29</td>
<td>45</td>
<td>34</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Max # regs</td>
<td>128</td>
<td>116</td>
<td>191</td>
<td>191</td>
<td>101</td>
<td>101</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>MaxLive/MinAvg</td>
<td>1.8317</td>
<td>1.7147</td>
<td>1.4489</td>
<td>1.4307</td>
<td>1.7227</td>
<td>1.6188</td>
<td>1.4737</td>
</tr>
<tr>
<td>Code size and Execution time</td>
<td># stages</td>
<td>8794</td>
<td>8794</td>
<td>8773</td>
<td>8766</td>
<td>8484</td>
<td>8485</td>
<td>8862</td>
</tr>
<tr>
<td></td>
<td># stages = ( y_{II} )</td>
<td>6246</td>
<td>6246</td>
<td>6281</td>
<td>6273</td>
<td>6173</td>
<td>6173</td>
<td>6356</td>
</tr>
<tr>
<td></td>
<td>Total cycles (in millions)</td>
<td>39185</td>
<td>39183</td>
<td>38628</td>
<td>38628</td>
<td>39343</td>
<td>39343</td>
<td>41223</td>
</tr>
</tbody>
</table>

Table 3: Effectiveness, cost, parallelism, register pressure, code size and execution time for the medium complexity architecture.

![Diagram](a)

![Diagram](b)

![Diagram](c)

Figure 4: Influence of the BudgetRatio on the iterative techniques for the complex architecture: a) percentage of non-scheduled operations, b) sum of II / sum of MII and c) total scheduling time.

Table 3 shows that IRIS again obtains the best results in terms of number of stages. On the other hand, we can see that SMS is again the best approach in terms of the execution time (cycles) of the final code. The main reason for this is that SMS is the best approach in terms of II.

We can conclude that for this architecture the best approach is SMS, because it obtains better results in terms of the total execution time, II, and registers (Slack being very close in register pressure). In addition, it requires less computing time than the other techniques. Applying Stage after SMS has some benefits in terms of registers, and stages. However, these benefits may not justify the large scheduling time required by Stage.

### 4.3 Complex Architecture

Figure 4.a shows the percentage of non-scheduled operations as a function of the BudgetRatio. As in the previous cases IMS and IRIS find a legal schedule for all loops. Slack again fails to schedule some loops for every BudgetRatio that we tried. As expected, increasing the BudgetRatio tends to decrease the number of non-scheduled loops (only 1 loop cannot be scheduled with BudgetRatio = 10). A BudgetRatio of 2.5 can be acceptable in terms of non-scheduled loops. In terms of II (Figure 4.b) all techniques follow a similar trend. Beyond a BudgetRatio of 2.5 the II remains practically the same.

Regarding the scheduling time (Figure 4.c) each technique has a different behavior. IRIS is close to a linear curve whereas SMS follows a super-linear curve, which leads to extremely high scheduling times for high BudgetRatios (800 seconds for BudgetRatio = 10). Finally Slack shows an oscillating behavior like in the simple architecture. We have chosen BudgetRatio = 2.5 to make a comparison of the techniques since it gives reasonable scheduling times for all the iterative techniques with good performance results.

In terms of effectiveness, Table 4 shows that IMS, SMS and IRIS manage to schedule all 1936 loops. On the other hand, Slack fails to schedule 20 loops representing 1.08% of the operations. In terms of scheduling time, there is not much difference among the iterative techniques, while SMS is several times faster than the others due to the lack of
5. CONCLUSIONS

Modulo Scheduling is a family of techniques that exploit instruction level parallelism in loops by overlapping the execution of several consecutive iterations. There have been many proposals in the recent years, but it is difficult to compare them since each proposal was evaluated with a different set of benchmarks, different compiler platforms and different architectures.

In this work we have performed a comparison, based on a quantitative evaluation, of the most relevant modulo scheduling techniques: Iterative Modulo Scheduling (IMS), Swing Modulo Scheduling (SMS), Slack Modulo Scheduling (Slack) and Integrated Register-Sensitive Iterative Software pipelining (IRIS). In addition we have evaluated the combination of these techniques with the Stage Scheduling technique, which is a post-pass that reduces the register requirements of a modulo schedule.

The quantitative evaluation has been performed with the majority of loops of the Perfect Chub Benchmark Suite and SPECfp95. This evaluation shows that SMS (the only technique that does not perform backtracking) generates the best schedules in terms of II for low and medium complexity architectures although the other techniques are quite close. In addition SMS is the technique with the lowest register pressure and the one that requires less scheduling time for all the architectures evaluated.

For complex architectures the iterative techniques produce schedules with slightly better II, IMS being the most effective technique. However, IMS has the highest register requirements for all tested architectures whereas SMS always has the lowest. Stage scheduling results in a reduction of register requirements in all cases, however it requires a noticeable long scheduling time and produces minimal effects on the techniques with the lowest register requirements (SMS and Slack). Also, the least effective techniques in terms of registers (IMS and IRIS) plus a Stage post-pass still have more register requirements than SMS and Slack without the Stage post-pass.

Overall, the statistics reported in this work shed some light on the problem of choosing the most effective modulo scheduling technique for each particular scenario.

6. ACKNOWLEDGMENTS

This work has been partially supported by the ESPRIT project MAHOTEU (EP 24942), the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01, Direcció General de Recerca de la Generalitat de Catalunya under grant 2001FI 00664 UPC APTIND and Analog Devices.
7. REFERENCES


