Modulo Scheduling with Reduced Register Pressure

Josep Llosa, Mateo Valero, *Member*, *IEEE*, Eduard Ayguadé, and Antonio González, *Member*, *IEEE Computer Society*

Abstract—Software pipelining is a scheduling technique that is used by some product compilers in order to expose more instruction level parallelism out of innermost loops. Modulo scheduling refers to a class of algorithms for software pipelining. Most previous research on modulo scheduling has focused on reducing the number of cycles between the initiation of consecutive iterations (which is termed *II*) but has not considered the effect of the register pressure of the produced schedules. The register pressure increases as the instruction level parallelism increases. When the register requirements of a schedule are higher than the available number of registers, the loop must be rescheduled perhaps with a higher *II*. Therefore, the register pressure has an important impact on the performance of a schedule. This paper presents a novel heuristic modulo scheduling strategy that tries to generate schedules with the lowest *II*, and, from all the possible schedules with such *II*, it tries to select that with the lowest register requirements. The proposed method has been implemented in an experimental compiler and has been tested for the Perfect Club benchmarks. The results show that the proposed method achieves an optimal *II* for at least 97.5 percent of the loops and its compilation time is comparable to a conventional top-down approach, whereas the register requirements are lower. In addition, the proposed method is compared with some other existing methods. The results indicate that the proposed method performs better than other heuristic methods and almost as well as linear programming methods, which obtain optimal solutions but are impractical for product compilers because their computing cost grows exponentially with the number of operations in the loop body.

Index Terms—Instruction scheduling, loop scheduling, software pipelining, register allocation, register spilling.

1 Introduction

Increasing the instruction level parallelism is an observed trend in the design of current microprocessors. This requires a combined effort from the hardware and software in order to be effective. Since most of the execution time of common programs is spent in loops, many efforts to improve performance have targeted loop nests.

Software pipelining [5] is an instruction scheduling technique that exploits the instruction level parallelism of loops by overlapping the execution of successive iterations of a loop. There are different approaches to generate a software pipelined schedule for a loop [1]. Modulo scheduling is a class of software pipelining algorithms that was proposed at the begining of last decade [24] and has been incorporated into some product compilers (e.g., [22], [7]). Besides, many research papers have recently appeared on this topic [11], [14], [26], [13], [29], [12], [27], [23], [30], [18].

Modulo scheduling framework relies on generating a schedule for an iteration of the loop such that when this same schedule is repeated at regular intervals, no dependence is violated and no resource usage conflict arises. The interval between the successive iterations is termed **Initiation Interval** (*II*). Having a constant initiation interval implies that no resource may be used more than once at the same time modulo *II*.

Most modulo scheduling approaches consists of two steps. First, they compute a schedule trying to minimize the *II* but without caring about register allocation and then, allocate variables to registers. The execution time of a software pipelined loop depends on the *II*, the length of the schedule for one iteration and the number of registers required by this schedule. The *II* directly impacts performance since it determines the issue rate of loop iterations. The schedule length also has a direct impact on performance since it determines the overhead each time a loop is entered, however its impact on performance is negligible for loops with large trip counts. Finally, the register requirements of the schedule may also impact performance if they exceed the number of available registers. In this case, the schedule is unfeasible and some actions must be taken in order to reduce the register pressure. Some possible solutions outlined in [25] and evaluated in [17] are:

- Reschedule the loop with an increased *II*. In general, increasing the *II* reduces the register requirements but decreases the issue rate, which has a direct negative effect on the execution time.
- Add spill code. This again has a negative effect since it increases the required memory bandwidth and it will result in additional memory penalties (e.g., cache misses). Besides, memory may become the most saturated resource and, therefore, adding spill code may result in increased *II*.

Most previous works have focused on reducing the *II* and, sometimes, also the length of the schedule for one iteration, but they have not considered the register requirements

- *The authors are with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: {josepll, mateo, eduard, antonio}@ac.upc.es.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106424.

0018-9340/98/$10.00 © 1998 IEEE

of the proposed schedule, which may have a severe impact on the performance as outlined above. A current trend in the design of new processors is the increase in the amount of instruction level parallelism that they can exploit. Exploiting more instruction level parallelism results in a significant increase in the register pressure [20], [19], which exacerbates the problem of ignoring its effect on the performance of a given schedule.

In order to obtain more effective schedules, a few recently proposed modulo scheduling approaches try to minimize both the *II* and the register requirements of the produced schedules.

Some of these approaches [10], [9] are based on formulating the problem in terms of an optimization problem and solving it using an integer linear programming approach. This may produce optimal schedules but, unfortunately, this approach has a computing cost that grows exponentially with the number of basic operations in the loop body. Therefore, they are impractical for big loops, which in most cases are the most time consuming parts of a program and, thus, they may be the ones that most benefit from software pipelining.

Practical modulo scheduling approaches used by product compilers use some heuristics to guide the scheduling process. The two most relevant heuristic approaches proposed in the literature that try to minimize both the *II* and the register pressure are: Slack Scheduling [12] and Stage Scheduling [8].

Slack Scheduling is an iterative algorithm with limited backtracking. At each iteration, the scheduler chooses an operation based on a previouly computed dynamic priority. This priority is a function of the slack of each operation (i.e., a measure of the scheduling freedom for that operation) and it also depends on how much critical the resources used by that operation are. The selected operation is placed in the partial schedule either as early as possible or as late as possible. The choice between these two alternatives is made basically by determining how many of the operation's inputs and outputs are stretchable and choosing the one that minimizes the involved values' lifetimes. If the scheduler cannot place the selected operation due to a lack of conflict-free issue slots, then it is forced to a particular slot and all the conflicting operations are ejected from the partial schedule. In order to limit this type of backtracking, if operations are ejected too many times, the *II* is incremented and the scheduling is started all over again.

Stage Scheduling is not a whole modulo scheduler by itself but a set of heuristic techniques that reduce the register requirements of any given modulo schedule. This objective is achieved by shifting operations by multiples of *II* cycles. The resulting schedule has the same *II* but lower register requirements.

This paper presents *Hypernode Reduction Modulo Scheduling (HRMS)*,[1] a heuristic modulo scheduling approach that tries to generate schedules with the lowest *II* and, from all the possible schedules with such *II*, it tries to select that with the lowest register requirements. The main part of HRMS is the ordering strategy. The ordering phase orders the nodes before scheduling them, so that only predecessors or successors of a node can be scheduled before it is scheduled (except for recurrences), reducing the register requirements. In addition, the ordering step gives priority to recurrence circuits (i.e., nodes belonging to recurrences are scheduled first), so that the *II* is minimized. During the scheduling step, the nodes are scheduled as early/late as possible if their predecessors/successors have been previously scheduled.

HRMS tries to reduce the register requirements during the scheduling like Slack Scheduling, while Stage Scheduling tries to reduce them during a postscheduling step. In general, a postpass step has less freedom to move the nodes leading to suboptimal schedules. However, since either HRMS and Slack are based on heuristics, we do not discard the use of a postpass like Stage as a complementary step for further reducing the register requirements. In addition, since HRMS gives priority to recurrence circuits, it can obtain good schedules without requiring backtracking, as in Slack, leading to a faster algorithm. However, backtracking can produce better schedules if the reservation tables of the operations are very complex. Notwithstanding, most of the current microprocessors have simple reservation tables like in the configurations used for the evaluation of HRMS.

The performance of HRMS is evaluated and compared with that of a conventional approach (a top-down scheduler) that does not care about register pressure. For this evaluation, we have used over a thousand loops from the Perfect Club Benchmark Suite [4] that account for 78 percent of its execution time. The results show that HRMS achieves an optimal *II* for at least 97.5 percent of the loops and its compilation time is comparable to the top-down approach, whereas the register requirements are lower.

In addition, HRMS has been tested for a set of loops taken from [10] and compared against two other heuristic strategies. These two strategies are the previously mentioned Slack Scheduling, and FRLC [28], which is a heuristic strategy that does not take into account the register requirements. In addition, HRMS is compared with SPILP [10], which is a linear programming formulation of the problem. Because of the computing requirements of this latter approach, only small loops are used for this comparison. The results indicate that HRMS obtains better schedules than the other two heuristic approaches and its results are very close to the ones produced by the optimal scheduler. The compilation time of HRMS is similar to the other heuristic methods and much lower than the linear programming approach.

The rest of this paper is organized as follows: In Section 2, an example is used to illustrate the motivation for this work, that is, reducing the register pressure in modulo scheduled loops while achieving near optimal *II*. Section 3 describes the proposed modulo scheduling algorithm that is called HRMS. Section 4 evaluates the performance of the proposed approach and, finally, Section 5 states the main conclusions of this work.

1. A preliminary version of this work appeared in [18].

2 Overview of Modulo Scheduling and Motivating Example

This section includes an overview of modulo scheduling and the motivation for the work presented in this paper. For a more detailed discussion on modulo scheduling, refer to [1].

2.1 Overview of Modulo Scheduling

In a software pipelined loop, the schedule for an iteration is divided into stages so that the execution of consecutive iterations that are in distinct stages is overlapped. The number of stages in one iteration is termed **stage count**(*SC*). The number of cycles per stage is *II*.

Fig. 1 shows the dependence graph for the running example used along this section. In this graph, nodes represent basic operations of the loop and edges represent values generated and consumed by these operations. For this graph, Fig. 2a shows the execution of the six iterations of the software pipelined loop with an *II* of 2 and an *SC* of 5. The operations have been scheduled assuming a four-wide issue machine, with general-purpose functional units (fully pipelined with a latency of two cycles). The scheduling of each iteration has been obtained using a top-down strategy that gives priority to operations in the critical path with the additional constraint that no resource can be used more than once at the same cycle modulo *II*. The figure also shows the corresponding lifetimes of the values generated in each iteration.

The execution of a loop can be divided into three phases: a ramp up phase that fills the software pipeline, a steady state phase where the software pipeline achieves maximum overlap of iterations, and a ramp down phase that drains the software pipeline. The code that implements the ramp up phase is termed the **prologue**. During the steady state phase of the execution, the same pattern of operations is executed in each stage. This is achieved by iterating on a piece of code, termed the **kernel**, that corresponds to one stage of the steady state phase. A third piece of code, called the **epilogue**, is required to drain the software pipeline after the execution of the steady state phase.

The initiation interval *II* between two successive iterations is bounded either by loop-carried dependences in the graph (*RecMII*) or by resource constraints of the architecture (*ResMII*). This lower bound on the *II* is termed the **Minimum Initiation Interval** (*MII* = max (*RecMII*, *ResMII*)). The reader is referred to [7], [23] for an extensive dissertation on how to calculate *ResMII* and *RecMII*.

Since the graph in Fig. 1 has no recurrence circuits, its initiation interval is constrained only by the available resources: $MII = \left\lceil \frac{7}{4} \right\rceil = 2$ (i.e., number of operations divided by number of resources). Notice that, in the scheduling of Fig. 2a, no dependence is violated and every functional unit is used, at most, once at all even cycles (cycle modulo 2 = 0) and, at most, once at all odd cycles (cycle modulo 2 = 1).

The code corresponding to the kernel of the software pipelined loop is obtained by overlapping the different stages that constitute the schedule of one iteration. This is shown in Fig. 2b. The subscripts in the code indicate relative iteration distance in the original loop between operations. For instance, in this example, each iteration of the kernel executes an instance of operation A and an instance of operation B of the previous iteration in the initial loop.

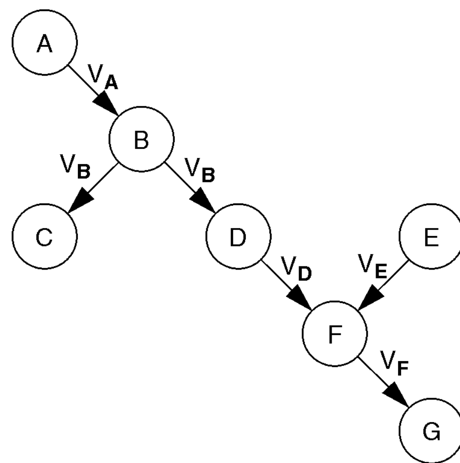

Fig. 1. A sample dependence graph.

Values used in a loop correspond either to loop-invariant variables or to loop-variant variables. Loop-invariants are repeatedly used but never defined during loop execution. Loop-invariants have a single value for all the iterations of the loop and, therefore, they require one register each, regardless of the scheduling and the machine configuration.

For loop-variants, a value is generated in each iteration of the loop and, therefore, there is a different value corresponding to each iteration. Because of the nature of software pipelining, lifetimes of values defined in an iteration can overlap with lifetimes of values defined in subsequent iterations. Fig. 2a shows the lifetimes for the loop-variants corresponding to every iteration of the loop. By overlapping the lifetimes of the different iterations, a pattern of length *II* cycles that is indefinitely repeated is obtained. This pattern is shown in Fig. 2c. This pattern indicates the number of values that are live at any given cycle. As shown in [25], the maximum number of simultaneously live values *MaxLive* is an accurate approximation of the number of register required by the schedule.[2] In this section, the register requirements of a given schedule will be approximated by *MaxLive*. However, in the experiments section, we will measure the actual register requirements after register allocation.

Values with a lifetime greater than *II* pose an additional difficulty, since new values are generated before previous ones are used. One approach to fix this problem is to provide some form of register renaming so that successive definitions of a value use distinct registers. Renaming can be performed at compile time by using modulo variable expansion [15], i.e., unrolling the kernel and renaming at compile time the multiple definitions of each variable that exist in the unrolled kernel. A rotating register file can be used to solve this problem without replicating code by renaming different instantiations of a loop-variant at execution time [6].

2. For an extensive discussion on the problem of allocating registers for software-pipelined loops, refer to [25]. The strategies presented in that paper almost always achieve the *MaxLive* lower bound. In particular, the wands-only strategy using end-fit with adjacency ordering never required more than *MaxLive* + 1 registers.

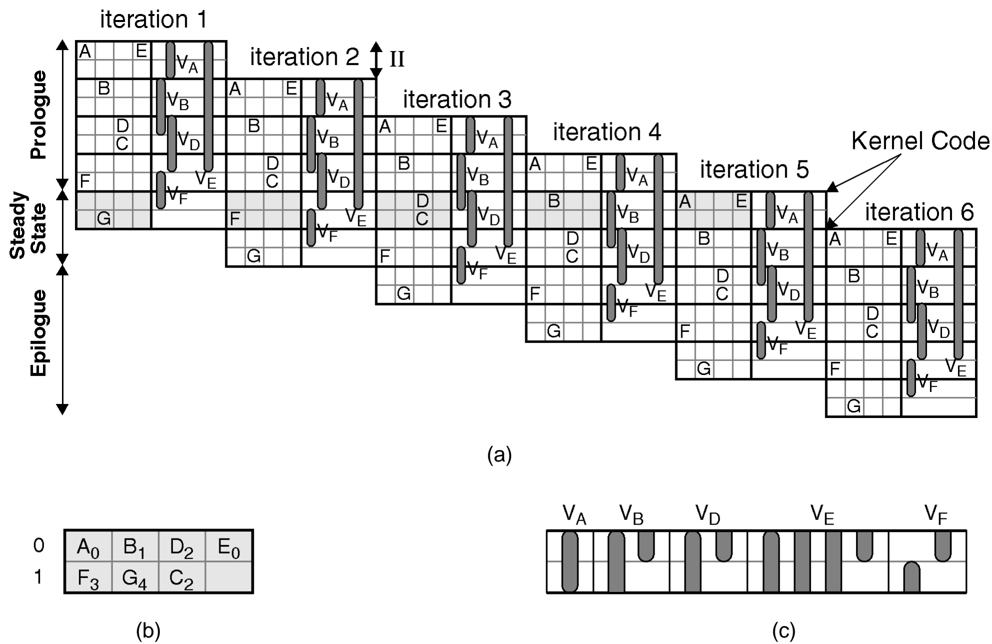

Fig. 2. (a) Software pipelined loop execution, (b) kernel, and (c) register requirements.

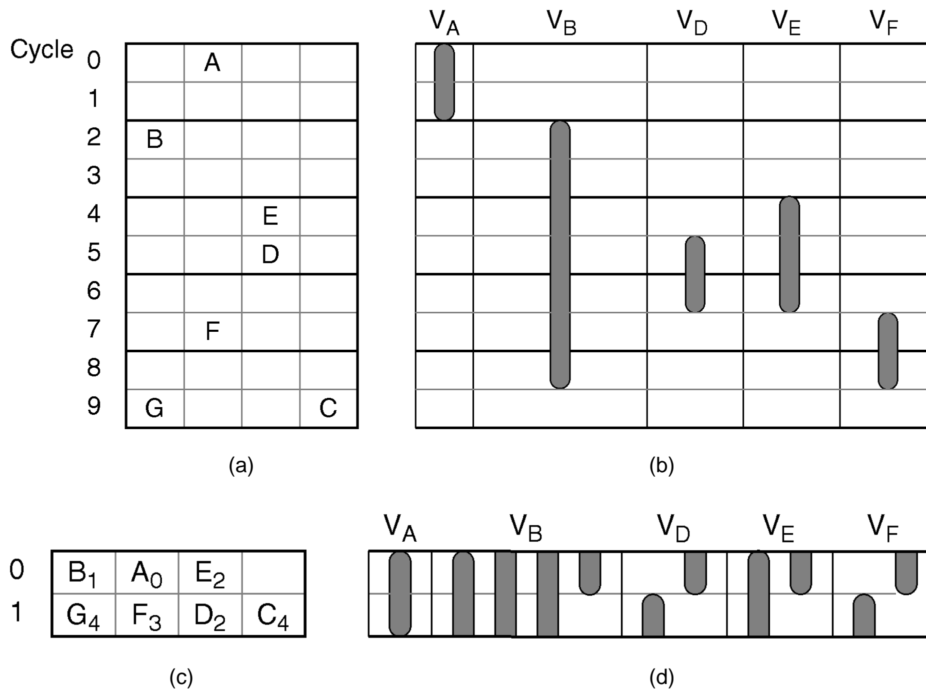

Fig. 3. Bottom-up scheduling: (a) schedule of one iteration, (b) lifetimes of variables, (c) kernel, and (d) register requirements.

2.2 Motivating Example

In many modulo scheduling approaches, the lifetimes of some values can be unnecessarily large. As an example, Fig. 2a shows a top-down scheduling, and Fig. 3a a bottom-up scheduling for the example graph of Fig. 1 and a machine with four general-purpose functional units with a two-cycle latency.

In a top-down strategy, operations can only be scheduled if all their predecessors have already been scheduled. Each node is placed as early as possible in order not to delay any possible successors. Similarly, in a bottom-up strategy, an operation is ready for scheduling if all its successors have already been scheduled. In this case, each node is placed as late as possible in order not to delay possible predecessors. In both strategies, when there are several candidates to be scheduled, the algorithm chooses the one that is more critical in the scheduling.

In the top-down scheduling, node E is scheduled before node F. Since E has no predecessors, it can be placed at any cycle, but, in order not to delay any possible successor, it is placed as early as possible. Fig. 2a shows the lifetimes of loop variants for the top-down scheduling assuming that a value is alive from the beginning of the producer operation

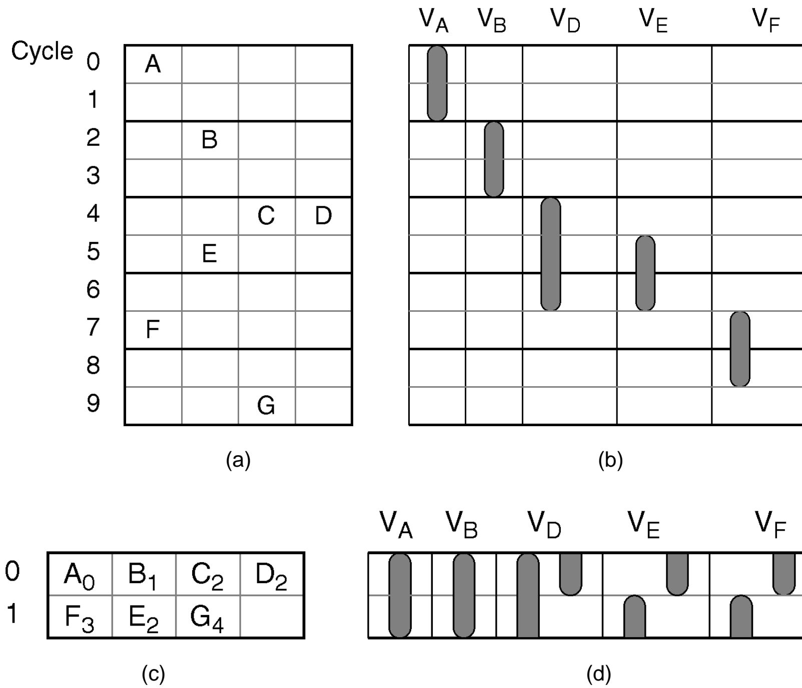

Fig. 4. HRMS scheduling: (a) schedule of one iteration, (b) lifetimes of variables, (c) kernel, and (d) register requirements.

to the beginning of the last consumer. Notice that loop variant V_E has an unnecessarily large lifetime due to the early placement of E during the scheduling.

In the bottom-up approach, E is scheduled after F, therefore, it is placed as late as possible reducing the lifetime of V_E (Fig. 3b). Unfortunately, C is scheduled before B and, in order to not delay any possible predecessor, it is scheduled as late as possible. Notice that the V_B has an unnecessarily large lifetime due to the late placement of C.

In HRMS, an operation will be ready for scheduling even if some of its predecessors and successors have not been scheduled. The only condition (to be guaranteed by the preordering step) is that when an operation is scheduled, the partial schedule contains only predecessors or successors or none of them, but not both of them (in the absence of recurrences). The ordering is done with the aim that all operations have a previously scheduled reference operation (except for the first operation to be scheduled). For instance, consider that nodes of the graph in Fig. 1 are scheduled in the order {A, B, C, D, F, E, G}. Notice that node F will be scheduled before nodes {E, G}, a predecessor and a successor, respectively, and that the partial scheduling will contain only a predecessor (D) of F. With this scheduling order, both C and E (the two conflicting operations in the top-down and bottom-up strategies) have a reference operation already scheduled when they are placed in the partial schedule.

Fig. 4a shows the HRMS scheduling for one iteration. Operation A will be scheduled in cycle 0. Operation B, which depends on A, will be scheduled in cycle 2. Then, C and, later, D are scheduled in cycle 4. At this point, operation F is scheduled as early as possible, i.e., at cycle 6 (because it depends on D), but there are no available resources at this cycle, so it is delayed to cycle 7. Now, the scheduler places operation E as late as possible in the scheduling because there is a successor of E previously placed in the partial scheduling, thus, operation E is placed at cycle 5. And, finally, since operation G has a predecessor previously scheduled, it is placed as early as possible in the scheduling, i.e., at cycle 9.

Fig. 4b shows the lifetimes of loop variants. Notice that neither C nor E have been placed too late or too early because the scheduler always takes previously scheduled operations as a reference point. Since F has been scheduled before E, the scheduler has a reference operation to decide a late start for E. Fig. 4d shows the number of live values in the kernel (Fig. 4c) during the steady state phase of the execution of the loop. There are six live values in the first row and five in the second. In contrast, the top-down schedule has 10 simultaneously live values and the bottom-up schedule has nine.

The following section describes the algorithm that orders the nodes before scheduling and the scheduling step.

3 Hypernode Reduction Modulo Scheduling

The dependences of an innermost loop can be represented by a Dependence Graph $G = DG(V, E, \delta, \lambda)$. V is the set of vertices of the graph G, where each vertex $v \in V$ represents an operation of the loop. E is the dependence edge set, where each edge $(u, v) \in E$ represents a dependence between two operations u, v. Edges may correspond to any of the following types of dependences: register dependences, memory dependences, or control dependences. The dependence distance $\delta_{(u,v)}$ is a nonnegative integer associated with each edge $(u, v) \in E$. There is a dependence of distance $\delta_{(u,v)}$ between two nodes u and v if the execution of operation v depends on the execution of operation u at $\delta_{(u,v)}$ iterations before. The latency λ_u is a nonzero positive integer associated with each node $u \in V$ and is defined as the number

of cycles taken by the corresponding operation to produce a result.

HRMS tries to minimize the register requirements of the loop by scheduling any operation u as close as possible to their *relatives* i.e., the *predecessors* of u, $Pred(u)$, and the *successors* of u, $Succ(u)$. Scheduling operations in this way shortens operand's lifetime and, therefore, reduces the register requirements of the loop.

To software pipeline a loop, the scheduler must handle cyclic dependences caused by recurrence circuits. The scheduling of the operations in a recurrence circuit must not be stretched beyond $\Omega \times II$, where Ω is the sum of the distances in the edges that constitute the recurrence circuit.

HRMS solves these problems by splitting the scheduling into two steps: A preordering step that orders nodes and the actual scheduling step that schedules nodes (once at a time) in the order given by the preordering step.

The preordering step orders the nodes of the dependence graph with the goal of scheduling the loop with an *II* as close as possible to *MII* and using the minimum number of registers. It gives priority to recurrence circuits in order not to stretch any recurrence circuit. It also ensures that, when a node is scheduled, the current partial scheduling contains only *predecessors* or *successors* of the node, but never both (unless the node is the last node of a recurrence circuit to be scheduled).

The ordering step assumes that the dependence graph, $G = (V, E, \delta, \lambda)$, is connected component. If G is not a connected component, it is decomposed into a set of connected components $\{G_i\}$, each G_i is ordered separately and, finally, the lists of nodes of all G_i are concatenated, giving a higher priority to the G_i with a more restrictive recurrence circuit (in terms of *RecMII*).

Next, the preordering step is presented. First we will assume that the dependence graph has no recurrence circuits (Section 3.1), and, in Section 3.2, we introduce modifications in order to deal with recurrence circuits. Finally, Section 3.3 presents the scheduling step.

3.1 Preordering of Graphs without Recurrence Circuits

To order the nodes of a graph, an initial node, that we call *Hypernode*, is selected. In an iterative process, all the nodes in the dependence graph are reduced to this *Hypernode*. The *reduction* of a set of nodes to the *Hypernode* consists of: Deleting the set of edges among the nodes of the set and the *Hypernode*, replacing the edges between the rest of the nodes and the reduced set of nodes by edges between the rest of the nodes and the *Hypernode*, and, finally, deleting the set of nodes being reduced.

The preordering step (Fig. 5) requires an initial *Hypernode* and a partial list of ordered nodes. The current implementation selects the first node of the graph (i.e., the node corresponding to the first operation in the program order), but any node of the graph can be taken as the initial *Hypernode*.[3] This node is inserted in the partial list of ordered nodes, then the preordering algorithm sorts the rest of the nodes.

```
function Pre_Ordering(G, L, h)
{Returns a list with the nodes of G ordered}
{It takes as input: }
{The dependence graph (G) }
{A list of nodes partially ordered (L) }
{An initial node (i.e., the hypernode) (h) }
  List := L;
  while ( Pred(h) ≠ ∅ or Succ(h ≠ ∅)
      V' := Pred(h);
      V' := Search_All_Paths(V', G);
      G' := Hypernode_Reduction(V', G, h);
      L' := Sort_PALA(G');
      List := Concatenate(List, L');
      V' := Succ(h);
      V' := Search_All_Paths(V', G);
      G' := Hypernode_Reducton(V', G, h);
      L' := Sort_ASAP(G');
      List := Concatenate(List, L')
return List
```

Fig. 5. Function that preorders the nodes in a dependence graph without recurrence circuits.

```
function Hypernode_Reduction(V', G, h)
{ G = (V, E, δ, λ); V' ⊂ V; h ∈ V }
{ Creates the subgraph G' = (V', E', δ, λ) ⊂ G }
{ And reduces G' to the node h in the graph G }
  E' := ∅;
  for each u ∈ V' do
     for each e = (v1, v2) ∈ Adj_edges(u) do
        E := E − {e};
        if v1 ∈ V' and v2 ∈ V'
           then E' := E ∪ {e}
        else
           if v1 = u and v2 ≠ h
              then E := E ∪ {(h, v2)}
           if v2 = u and v1 ≠ h
              then E := E ∪ {(v1, h)}
     V := V − {u}
return G'
```

Fig. 6. Function *Hpernode_Reduction*.

At each step, the predecessors and successors of the *Hypernode* are alternatively determined. Then, the nodes that appear in any path among the predecessors (successors) are obtained (function *Search_All_Paths*).[4] Once the predecessors (successors) and all the paths connecting them have been obtained, all these nodes are reduced (see function *Hypernode_Reduction* in Fig. 6) to the *Hypernode*, and the subgraph which contains them is topologically sorted. The topological sort determines the partial order of predecessors (successors), which is appended to the ordered list of nodes. The predecessors are topologically sorted using the PALA algorithm. The PALA algorithm is like an ALAP (As Late As Possible) algorithm, but the list of ordered nodes is inverted. The successors are topologically sorted using an ASAP (As Soon As Possible) algorithm.

As an example, consider the dependence graph in Fig. 7a. Next, we illustrate the ordering of the nodes of this graph step by step.

1) Initially, the list of ordered nodes is empty (*List* = {}). We start by designating a node of the graph as the

3. Preliminary experiments showed that selecting different initial nodes produced different schedules that had approximately the same register requirements (there were minor differences caused by resource constraints).

4. The execution time of *Search_All_Paths* is $O(\|V\| + \|E\|)$.

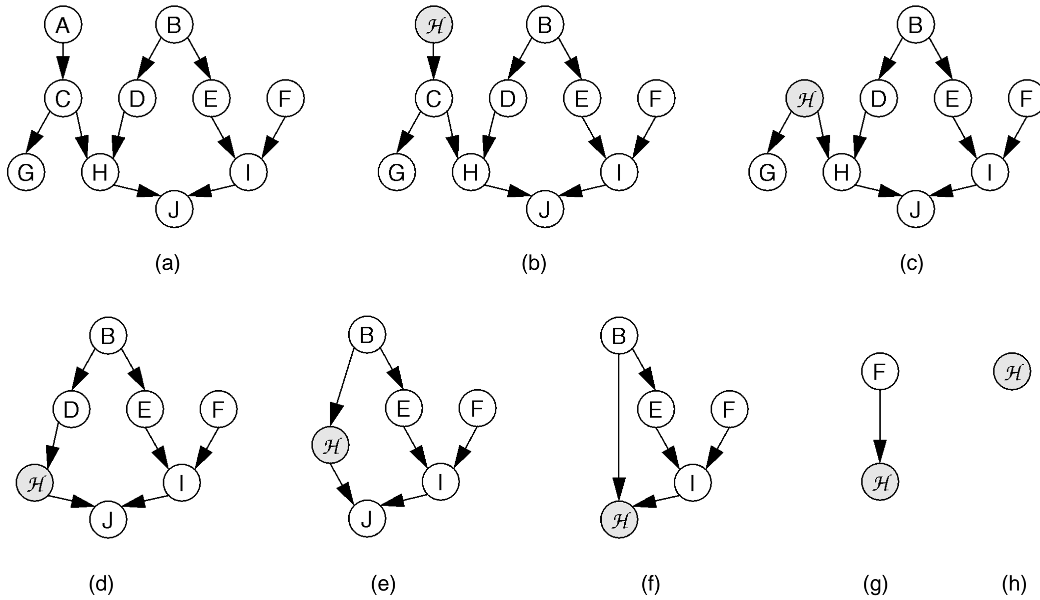

Fig. 7. Example of reordering without recurrences.

Hypernode ($\mathcal{H}$ in Fig. 7). Assume that A is the first node of the graph. The resulting graph is shown in Fig. 7b. Then, A is appended to the list of ordered nodes (*List* = {*A*}).

2) In the next step, the predecessors of $\mathcal{H}$ are selected. Since it has no predecessors, the successors are selected (i.e., the node C). Node C is reduced to $\mathcal{H}$, resulting in the graph of Fig. 7c, and C is added to the list of ordered nodes (*List* = {*A*, *C*}).
3) The process is repeated, selecting nodes G and H. In the case of selecting multiple nodes, there may be paths connecting these nodes. The algorithm looks for the possible paths, and topologically sorts the nodes involved. Since there are no paths connecting G and H, they are added to the list (*List* = {*A*, *C*, *G*, *H*}), and reduced to the *Hypernode*, resulting the graph of Fig. 7d.
4) Now, $\mathcal{H}$ has D as a predecessor, thus D is reduced, producing the graph in Fig. 7e, and appended to the list (*List* = {*A*, *C*, *G*, *H*, *D*}).
5) Then, J, the successor of $\mathcal{H}$, is ordered (*List* = {*A*, *C*, *G*, *H*, *D*, *J*}) and reduced, producing the graph in Fig. 7f.
6) At this point, $\mathcal{H}$ has two predecessors B and I, and there is a path between B and I that contains the node E. Therefore, B, E, and I are reduced to $\mathcal{H}$, producing the graph of Fig. 7g. Then, the subgraph that contains B, E, and I is topologically sorted, and the partially ordered list {*I*, *E*, *B*} is appended to the list of ordered nodes (*List* = {*A*, *C*, *G*, *H*, *D*, *J*, *I*, *E*, *B*}).
7) Finally, node F is reduced to $\mathcal{H}$, producing the graph of Fig. 7h with only the *Hypernode*, which is the stop condition of the ordering algorithm.

After performing the ordering phase, the nodes will be scheduled in the order {A, C, G, H, D, J, I, E, B, F}. Notice that the nodes that have been ordered as predecessors (i.e., I, E, B, and F) will be scheduled as late as possible, while the nodes ordered as successors will be scheduled as early as possible.

3.2 Preordering of Graphs with Recurrence Circuits

In order not to degrade performance when there are recurrence circuits, the ordering step is performed giving more priority to the recurrence circuits with higher *RecMII*. The main idea is to reduce all the recurrence circuits to the *Hypernode*, while ordering their nodes. After this step, we have a dependence graph without recurrence circuits, with an initial *Hypernode* and with a partial ordering of all the nodes that were contained in recurrence circuits. Then, we order this dependence graph as shown in Subsection 3.1.

Before presenting the ordering algorithm for recurrence circuits, let us put forward some considerations about recurrences. Recurrence circuits can be classified as:

- Single recurrence circuits (Fig. 8a).
- Recurrence circuits that share the same set of backward edges (Fig. 8b). We call the set of recurrence circuits that share the same set of backward edges *recurrence subgraph*. In this way, Figs. 8a and 8b are recurrence subgraphs.
- Several recurrence circuits can share some of their nodes (Figs. 8c and 8d) but have distinct sets of backward edges. In this case, we consider that these recurrence circuits are different recurrence subgraphs.

All recurrence circuits are identified during the calculation of *RecMII*. For instance, the recurrence circuits of the graph of Fig. 8b are {A, D, E} and {A, B, C, E}. Recurrence circuits are grouped into recurrence subgraphs (in the worst case there may be a recurrence subgraph for each backward edge). For instance, the recurrence circuits of Fig. 8b are grouped into the recurrence subgraph {A, B, C, D, E}. Recurrence subgraphs are ordered based on the highest *RecMII* value of the recurrence circuits contained in each subgraph, in a decreasing order. The nodes that appear in more than one subgraph are removed from all of them except for the most restrictive subgraph in terms of *RecMII*. For instance, the list of recurrence subgraphs associated with

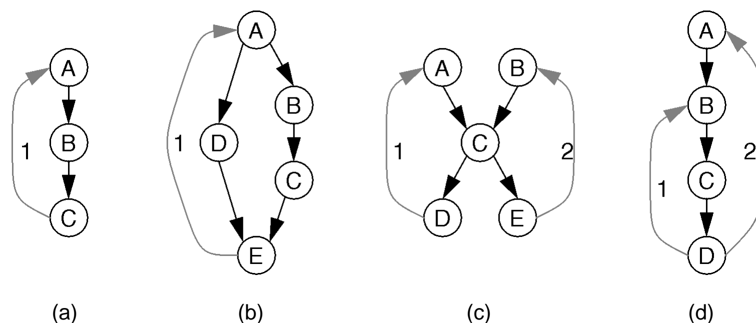

Fig. 8. Types of recurrences.

Fig. 8c {{A, C, D}, {B, C, E}} will be simplified to the list {{A, C, D}, {B, E}}.

The algorithm that orders the nodes of a grah with recurrence circuits (see Fig. 9) takes as input a list L of the recurrence subgraphs ordered by decreasing values of their *RecMII*. Each entry in this list is a list of the nodes traversed by the associated recurrence subgraph. Trivial recurrence circuits, i.e., dependences from an operation to itself, do not affect the preordering step since they do not impose scheduling constraints, as the scheduler previously ensured that $II \geq RecMII$. The algorithm starts by generating the corresponding subgraph for the first recurrence circuit, but without one of the backward edges that causes the recurrence (we remove the backward edge with higher $\delta_{(u,v)}$). Therefore, the resulting subgraph has no recurrences and can be ordered using the algorithm without recurrences presented in Section 3.1. The whole subgraph is reduced to the *Hypernode*. Then, all the nodes in any path between the *Hypernode* and the next recurrence subgraph are identified (in order to properly use the algorithm *Search_All_Paths* it is required that all the backward edges causing recurrences have been removed from the graph). After that, the graph containing the *Hypernode*, the next recurrence circuit, and all the nodes that are in paths that connect them are ordered, applying the algorithm without recurrence circuits, and reduced to the *Hypernode*. If there is no path between the *Hypernode* and the next recurrence circuit, any node of the recurrence circuit is reduced to the *Hypernode*, so that the recurrence circuit is now connected to the *Hypernode*.

```
procedure Ordering_Recurrences(G, L, List, h)
{This procedure takes the dependence graph (G)}
{and the simplified list of recurrence subgraphs (L)}
{It returns a partial list of ordered nodes (List)}
{and the resulting hypernode (h)}
  V' := Head(L);
  G' := Generate_Subgraph(V', G);
  h := First(G');
  List := Pre_Ordering(G', List, h);
  while L ≠ ∅ do
      V' := Search_All_Paths({h, Head(L)}, G);
      G' := Generate_Subgraph(V', G);
      List := Pre_Ordering(G', List, h);

function Generate_Subgraph(V, G)
{This function takes the dependence graph (G) and a subset of nodes V}
{And returns the graph that consists of all the nodes in V and the edges}
{among them}
```

Fig. 9. Procedure to order the nodes in the recurrence circuits.

This process is repeated until there are no more recurrence subgraphs in the list. At this point, all the nodes in recurrence circuits or in paths connecting them have been ordered and reduced to the *Hypernode*. Therefore, the graph that contains the *Hypernode* and the remaining nodes is a graph without recurrence circuits that can be ordered using the algorithm presented in the previous subsection.

For instance, consider the dependence graph of Fig. 10a. This graph has two recurrence subgraphs {A, C, D, F} and {G, J, M}. Next, we will illustrate the reduction of the recurrence subgraphs:

1) The subgraph {A, C, D, F} is the one with the highest *RecMII*. Therefore, the algorithm starts by ordering it. By isolating this subgraph and removing the backward edge, we obtain the graph of Fig. 10b. After ordering this graph, the list of ordered nodes is (*List* = {*A*, *C*, *D*, *F*}). When the graph of Fig. 10b is reduced to the *Hypernode*, $\mathcal{H}$ in the original graph (Fig. 10a), we obtain the dependence graph of Fig. 10c.
2) The next step is to reduce the following recurrence subgraph {G, J, M}. For this purpose, the algorithm searches for all the nodes that are in all possible paths between $\mathcal{H}$ and the recurrence subgraphs. Then, the graph that contains these nodes is constructed (see Fig. 10d). Since backward edges have been removed, this graph has no recurrence circuits, so it can be ordered using the algorithm presented in the previous section. When the graph has been ordered, the list of nodes is appended to the previous one, resulting in the partial list (*List* = {*A*, *C*, *D*, *F*, *I*, *G*, *J*, *M*}). Then, this subgraph is reduced to the *Hypernode* in the graph of Fig. 10c, producing the graph of Fig. 10e.
3) At this point, we have a partial ordering of the nodes belonging to recurrences, and the initial graph has been reduced to a graph without recurrence circuits (Fig. 10e). This graph without recurrence circuits is ordered as presented in Subsection 3.1. So, finally, the list of ordered nodes is *List* = {*A*, *C*, *D*, *F*, *I*, *G*, *J*, *M*, *H*, *E*, *B*, *L*, *K*}.

3.3 Scheduling Step

The scheduling step places the operations in the order given by the ordering step. The scheduling tries to schedule the operations as close as possible to the *neighbors* that have already been scheduled. When an operation is to be scheduled, it is scheduled in different ways depending on the *neighbors* of these operations that are in the partial schedule.

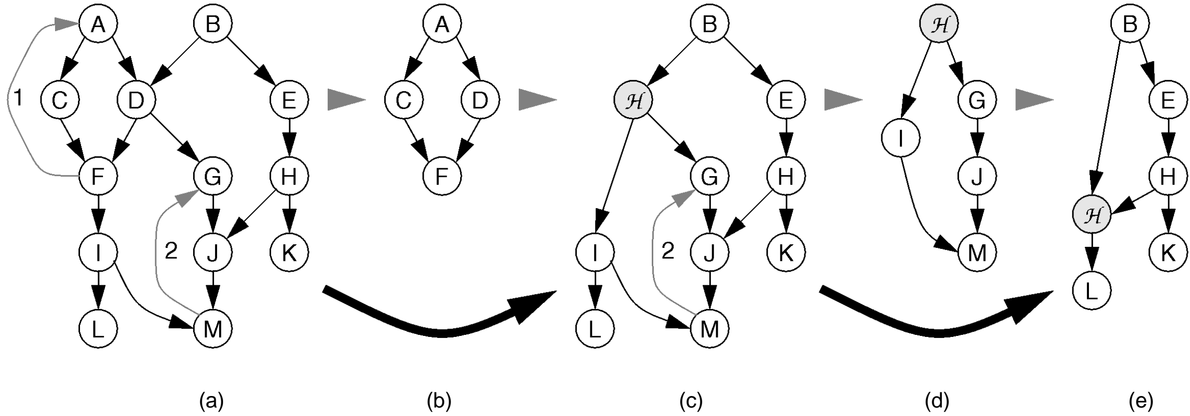

Fig. 10. Example for *Ordering_Recurrences* procedure.

TABLE 1
CLASSIFICATION OF THE PERFECT CLUB LOOPS

	Neither	Conditional	Recurrences	Both	Total
Number of loops	735	18	438	67	1,258

- If an operation u has only *predecessors* in the partial schedule, then u is scheduled as early as possible. In this case, the scheduler computes the *Early_Start* of u as:

$$Early_Start_u = \max_{v \in PSP(u)} t_v + \lambda_v - \delta_{(v,u)} \times II,$$

  where t_v is the cycle where v has been scheduled, λ_v is the latency of v, $\delta_{(v,u)}$ is the dependence distance from v to u, and $PSP(u)$ is the set of *predecessors* of u that have been previously scheduled. Then, the scheduler scans in the partial schedule for a free slot for the node u starting at cycle $Early_Start_u$ until the cycle $Early_Start_u + II - 1$. Notice that, due to the modulo constraint, it makes no sense to scan more than II cycles.
- If an operation u has only *successors* in the partial schedule, then u is scheduled as late as possible. In this case, the scheduler computes the *Late_Start* of u as:

$$Late_Start_u = \min_{v \in PSS(u)} t_v - \lambda_u + \delta_{(u,v)} \times II,$$

  where $PSS(u)$ is the set of *successors* of u that have been previously scheduled. Then, the scheduler scans in the partial schedule for a free slot for the node u starting at cycle $Late_Start_u$ until the cycle $Late_Start_u - II + 1$.
- If an operation u has *predecessors* and *successors*, then the scheduler scans the partial schedule starting at cycle $Early_Start_u$ until the cycle $\min(Late_Start_u, Early_Start_u + II - 1)$.
- Finally, if an operation u has neither predecessors nor successors, the scheduler computes the *Early_Start* of u as: $Early_Start_u = ASAP_u$ and scans the partial schedule for a free slot for the node u from cycle $Early_Start_u$ to cycle $Early_Start_u + II - 1$

If no free slots are found for a node, then the II is increased by 1. The scheduling step is repeated with the increased II, which will result in more opportunities for finding free slots. An advantage of HRMS is that the nodes are ordered only once, even if the scheduling step has to do several trials.

4 EVALUATION OF HRMS

In this section, we present some results of our experimental study. First, the complexity and performance of HRMS are evaluated for a benchmark suite composed of a large number of innermost DO loops in the Perfect Club [4]. We have selected those loops that include a single basic block. Loops with conditionals in their body have been previously converted to single basic block loops using IF-conversion [2]. We have not included loops with subroutine calls or with conditional exits. The dependence graphs have been obtained using the experimental ICTINEO compiler [3]. A total of 1,258 loops, which account for 78 percent of the total execution time[5] of the Perfect Club, have been scheduled. For these loops, the performance of HRMS is compared with the performance of a Top-Down scheduler. Second, we compare HRMS with other scheduling methods proposed in the literature using a small set of dependence graphs for which there are previously published results [10].

In order to give an idea of the complexity of the graphs, Table 1 characterizes the extracted loops in terms of: the presence of conditionals and recurrences. The first column shows the number of loops that have neither a conditional statement nor a recurrence circuit. The second and third columns show the number of loops that have, respectively, conditionals and recurrences. Finally, the fourth column shows the number of loops that have both. Table 2 characterizes the loops in terms of number and type of operations, and number and type of dependences. The 50% column refers to the median (e.g., 50 percent of the loops have seven or less operations) and the 90% column refers to the 90th percentile (e.g., 90 percent of the loops have 40 or less operations). Finally, we also show the maximum number for any loop and the sum for all loops.

5. Executed on an HP 9000/735 workstation and compiled with the +O3 flag (which performs software pipelining among other optimizations).

TABLE 2
COMPLEXITY OF THE PERFECT CLUB LOOPS

Metric	50%	90%	Max	Total
Operations	7	40	376	20,318
Memory references	4	14	44	7,368
Compute operations	3	27	356	12,950
nonpipelined	0	1	15	262
Dependences	6	54	530	25,613
Intraloop	6	49	518	23,468
Loop carried	0	5	24	2,145
Conditional	0	0	95	901

4.1 Performance Evaluation of HRMS

We have used two machine configurations to evaluate the performance of HRMS. Both configurations have two load/store units, two adders, two multipliers, and two Div/Sqrt units. We assume a unit latency for store instructions, a latency of two for loads, a latency of four (configuration L4) or six (configuration L6) for additions and multiplications, a latency of 17 for divisions, and a latency of 30 for square roots. All units are fully pipelined except the Div/Sqrt units, which are not pipelined at all.

In order to evaluate performance, the execution time (in cycles) of a scheduled loop has been estimated as the *II* of this loop times the number of iterations this loop performs (i.e., the number of times the body of the loop is executed). For this purpose, the programs of the Perfect Club have been instrumented to obtain the number of iterations of the selected loops.

HRMS achieved $II = MII$ for 1,227 loops, which means that it is optimal in terms of *II* for at least 97.5 percent of the loops. On average, the scheduler achieved an $II = 1.01 \times MII$. Considering dynamic execution time, the scheduled loops would execute at 98.4 percent of the maximum performance.

Register allocation has been performed using the wands-only strategy and the end-fit with adjacency ordering. For an extensive discussion of the problem of allocating registers for software-pipelined loops, refer to [25].

Fig. 11 compares the register requirements of loop-variants for the two scheduling techniques (top-down that does not care about register requirements and HRMS) for the two configurations mentioned above. This figure plots the percentage of loops that can be scheduled with a given number of registers without spill code. On average, HRMS requires 87 percent of the registers required by the top-down scheduler. Notice that "top-down" produces schedules with higher *II* and that, in general, a bigger *II* requires fewer registers. Therefore, HRMS is less register demanding, despite producing more aggressive schedules.

Since machines have a limited number of registers, it is also of interest to evaluate the effect of the register requirements on performance and memory traffic. When a loop requires more than the available number of registers, spill code has to be added and the loop has to be rescheduled. In [16], different alternatives and heuristics are proposed to speed up the generation of spill code. Among them, we have used the heuristic that spills the variable that maximizes the quotient between lifetime and the number of additional loads and stores required to spill the variable; this heuristic is the one that produces the best results.

Figs. 12 and 13 show the memory traffic and the execution time, respectively, of the loops scheduled with both schedulers when there are infinite, 64 and 32 registers available. Notice that, in general, HRMS requires less memory traffic than top-down when the number of registers is limited. The difference in memory traffic requirements between both schedulers increases as the number of available registers decreases. For instance, for configuration L6, HRMS requires 88 percent of the traffic required by the top-down scheduler if 64 registers are available. If only 32 registers are available, it requires 82.5 percent of the traffic required by the top-down scheduler.

In addition, assuming an ideal memory system, the loops scheduled by HRMS execute faster than the ones scheduled by top-down. This is because HRMS gives priority to recurrence circuits, so loops with recurrences usually produces better results than top-down. An additional factor that increases the performance of HRMS over top-down is that it reduces the register requirements. For instance, for configuration L6, scheduling the loops with HRMS produces a speed-up over top-down of 1.18 under the ideal assumption that an infinite register file is available. The speed-up is 1.20 if the register file has 64 registers and 1.25 if it has only 32 registers.

Notice that, for both schedulers, the most aggressive configuration (L6) requires more registers than the L4 configuration. This is because the degree of pipelining of the functional units has an important effect on the register pressure [20], [16]. The high register requirements of aggressive configurations produces a significant degradation of performance and memory traffic when a limited number of registers is available [16]. For instance, the loops scheduled with HRMS require 6 percent more cycles to execute for configuration L6 than for L4 if an infinite number of registers is assumed. If only 32 registers are available, L6 requires 16 percent more cycles than L4.

4.2 Complexity of HRMS

Scheduling our testbench consumed 55 seconds in a Sparc-10/40 workstation. This time compares to the 69 seconds consumed by the top-Down scheduler. The breakdown of the scheduler execution time in the different steps is shown in Fig. 14. Notice that, in HRMS, computing the recurrence circuits consumed only 7 percent, the preordering step consumed 66 percent, and the scheduling step consumed 27 percent. Even though most of the time is spent in the preordering step, the overall time is extremely short. The extra time lost in preordering the nodes allows for a very simple (and fast) scheduling step. In the top-Down scheduler, the preordering step consumed a small percentage of the time but the scheduling step required a lot of time; when the scheduler fails to find an schedule with a given *II*, the loop has to be rescheduled again with an increased initiation interval, and top-Down has to reschedule the loops much more often than HRMS.

4.3 Comparison with Other Scheduling Methods

In this section, we compare HRMS with three schedulers: an heuristic method that does not take into account register requirements (FRLC [28]), a lifetime sensitive heuristic

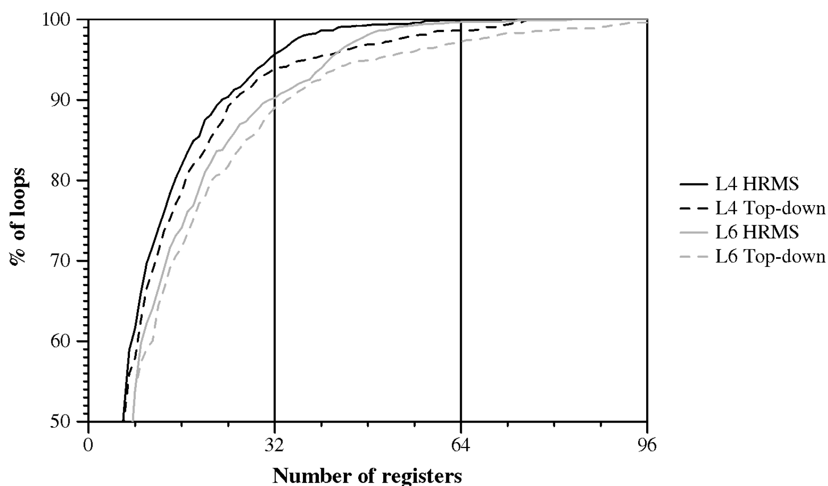

Fig. 11. Static cumulative distribution of register requirements of loop variants.

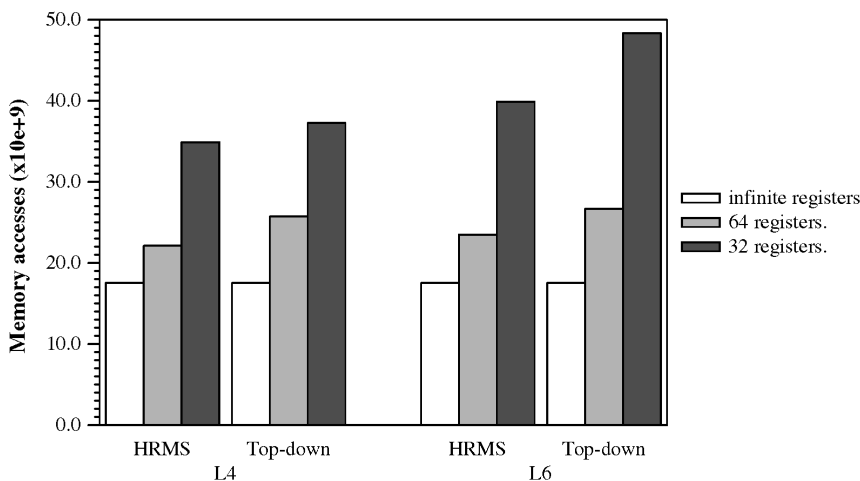

Fig. 12. memory traffic with infinite registers, 64 registers, and 32 registers.

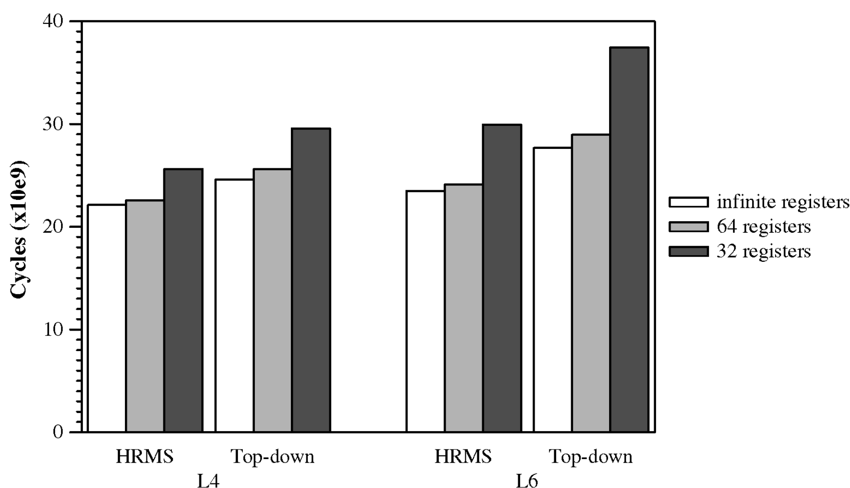

Fig. 13. Cycles required to execute the loops with infinite registers, 64 registers, and 32 registers.

method (Slack [12]) and a linear programming approach (SPILP [10]).

We have scheduled 24 dependence graphs for a machine with one FP Adder, one FP Multiplier, one FP Divider, and one Load/Store unit. We have assumed a unit latency for add, subtract, and store instructions, a latency of two for multiply and load, and a latency of 17 for divide.

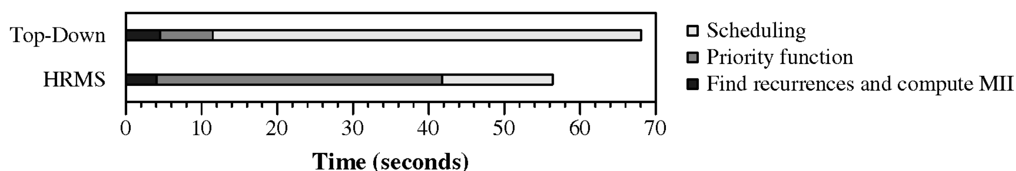

Fig. 14. Time to schedule all 1,258 loops for the HRMS and top-down schedulers.

TABLE 3
COMPARISON OF HRMS SCHEDULES WITH OTHER SCHEDULING METHODS

Application Program		HRMS			SPILP			Slack			FRLC		
		II	Buf	Secs	II	Buf	Secs	II	Buf	Secs	II	Buf	Secs
Spice	Loop1	1	3	0.01	1	3	0.82	1	3	0.01	2	2	0.02
	Loop2	6	9	0.03	6	9	12.47	7	9	0.03	6	16	0.03
	Loop3	6	4	0.01	6	4	0.72	6	4	0.02	6	4	0.02
	Loop4	11	12	0.20	11	12	3.60	12	12	0.10	12	12	0.03
	Loop5	2	2	0.01	2	2	0.70	2	2	0.02	2	2	0.02
	Loop6	2	16	0.08	2	16	7.67	3	11	0.03	17	9	0.03
	Loop7	3	17	0.08	3	17	0.70	3	17	0.03	17	11	0.01
	Loop8	3	6	0.02	3	6	3.15	5	5	0.03	3	8	0.02
	Loop10	3	4	0.02	3	4	1.88	3	4	0.02	3	5	0.02
Doduc	loop1	20	12	0.17	20	12	4.35	20	13	0.03	20	15	0.03
	Loop3	20	11	0.15	20	11	1.03	20	11	0.03	20	22	0.03
	Loop7	2	20	0.10	2	20	0.70	2	20	0.01	18	5	0.03
Fpppp	Loop1	20	5	0.13	20	5	0.93	20	5	0.03	20	6	0.02
Liver	Loop1	3	10	0.02	3	10	1.97	5	10	0.05	4	15	0.02
	Loop5	3	5	0.02	3	5	0.73	3	5	0.05	3	6	0.02
	Loop23	9	23	0.10	9	23	233.41	9	23	0.13	9	40	0.12
Linpack	Loop1	2	5	0.02	2	5	2.62	2	5	0.02	3	4	0.02
Whets.	Loop1	17	16	0.10	17	16	4.25	18	16	0.17	18	16	0.08
	Loop2	6	9	0.08	6	9	2.05	7	9	0.03	17	7	0.03
	Loop3	5	5	0.02	5	5	0.73	5	5	0.02	5	5	0.02
	Cycle1	4	4	0.02	4	4	0.75	4	4	0.02	4	4	0.02
	Cycle2	4	5	0.02	4	5	1.87	4	5	0.02	4	5	0.02
	Cycle4	4	7	0.02	4	7	1.85	4	7	0.01	4	7	0.03
	Cycle8	4	11	0.02	4	11	1.77	4	11	0.02	4	11	0.02

Table 3 compares the initiation interval *II*, the number of buffers (Buf), and the total execution time of the scheduler on a Sparc-10/40 workstation for the four scheduling methods. The results for the other three methods have been obtained from [10] and the dependence graphs to perform the comparison supplied by its authors. The number of buffers required by a schedule is defined in [10] as the sum of the buffers required by each value in the loop. A value requires as many buffers as the number of times the producer instruction is issued before the issue of the last consumer. In addition, stores require one buffer. In [21], it was shown that the buffer requirements provide a very tight upper bound on the total register requirements.

Table 4 summarizes the main conclusions of the comparison. The entries of the table represent the number of loops for which the schedules obtained by HRMS are better (*II* <), equal (*II* =), or worse (*II* >) than the schedules obtained by the other methods in terms of the initiation interval. When the initiation interval is the same, it also shows the number of loops for which HRMS requires fewer buffers (*Buf* <), equal number of buffers (*Buf* =), or more buffers (*Buf* >). Notice that HRMS achieves the same performance as the SPILP method both in terms of *II* and buffer requirements. When compared to the other methods, HRMS obtains a lower *II* in about 33 percent of the loops. For the remaining 66 percent of the loops, the *II* is the same, but, in many cases, HRMS requires less buffers, especially when compared with FRLC.

Finally, Table 5 compares the total compilation time in seconds for the four methods. Notice that HRMS is slightly faster than the other two heuristic methods; in addition, these methods perform noticeably worse in finding good schedulings. On the other hand, the linear programming method (SPILP) requires a much higher time to construct a scheduling that turns out to have the same performance as the scheduling produced by HRMS. In fact, most of the time spent by SPILP is due to Livermore Loop 23, but even without taking into account this loop, HRMS is over 40 times faster.

5 CONCLUSIONS

In this paper, we have presented *Hypernode Reduction Modulo Scheduling* (HRMS), a novel and effective heuristic technique for resource-constrained software pipelining.

TABLE 4
COMPARISON OF HRMS PERFORMANCE VS. THE OTHER THREE METHODS

	II <	*II* =			*II* >
		Buff <	Buff =	Buff >	
SPILP	0	0	24	0	0
Slack	7	1	16	0	0
FRLC	9	8	7	0	0

TABLE 5
COMPARISON OF HRMS COMPILATION TIME TO THE OTHER THREE METHODS

	HRMS	SPILP	Slack	FRLC
Compilation Time	0.32	290.72	0.93	0.71

HRMS attempts to optimize the initiation interval while reducing the register requirements of the schedule.

HRMS works in three main steps: computation of *MII*, preordering of the nodes of the dependence graph using a priority function, and scheduling of the nodes following this order. The ordering function ensures that, when a node is scheduled, the partial scheduling contains at least a reference node (a predecessor or a successor), except for the particular case of recurrences. This tends to reduce the lifetime of loop variants and, thus, reduce register requirements. In addition, the ordering function gives priority to recurrence circuits in order not to penalize the initiation interval. This ordering step, assuming resources without very complex reservation tables, allows HRMS to generate efficient schedules, even for recurrence intensive loops, without requiring the use of backtracking during the scheduling phase, which could be more time-consuming.

We provided a comprehensive evaluation of HRMS using 1,258 loops from the Perfect Club Benchmark Suite. We have seen that HRMS generates schedules that are optimal in terms of *II* for at least 97.5 percent of the loops. Although the preordering step consumes a high percentage of the total compilation time, the total scheduling time is smaller than the time required by a conventional top-down scheduler. In addition, HRMS provides a significant performance advantage over a top-down scheduler when there is a limited number of registers. This better performance comes from a reduction of the execution time and the memory traffic (due to spill code) of the software pipelined execution.

We have also compared our proposal with three other methods: the SPILP integer programming formulation, Slack Scheduling, and FRLC Scheduling. Our schedules exhibit significant improvement in performance in terms of initiation interval and buffer requirements compared to FRLC, and a significant improvement in the initiation interval when compared to Slack lifetime sensitive heuristic. We obtained similar results to SPILP, which is an integer linear programming approach that obtains optimal solutions, but has a prohibitive compilation time for real loops.

ACKNOWLEDGMENTS

The authors would like to thank Q. Ning, R. Govindarajan, Erik R. Altman, and Guang R. Gao for supplying us the dependence graphs they used in [10] in order to compare our proposal with other methods. Thanks are also due to the anonymous reviewers for their fruitful comments that undoubtedly have contributed to improve the quality of this paper. This work has been supported by the Ministry of Education of Spain under contract TIC 429/95, and by CEPBA (European Center for Parallelism of Barcelona).

REFERENCES

[1] V.H. Allan, R.B Jones, R.M. Lee, and S.J. Allan, “Software Pipelining,” *ACM Computing Surveys*, vol. 27, no. 3, pp. 367-432, Sept. 1995.

[2] J.R. Allen, K. Kennedy, and J. Warren, “Conversion of Control Dependence to Data Dependence,” *Proc. 10th Ann. Symp. Principles of Programming Languages*, Jan. 1983.

[3] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera, and M. Valero, “Ictineo: A Tool for Instruction Level Parallelism Research,” Technical Report UPC-DAC-96-61, Universitat Politècnica de Catalunya, Dec. 1996.

[4] M. Berry, D. Chen, P. Koss, and D. Kuck, “The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers,” Technical Report 827, Center for Supercomputing Research and Development, Nov. 1988.

[5] A.E. Charlesworth, “An Approach to Scientific Array Processing: The Architectural Design of the AP120B/FPS-164 Family,” *Computer*, vol. 14, no. 9, pp. 18-27, Sept. 1981.

[6] J.C. Dehnert, P.Y.T. Hsu, and J.P. Bratt, “Overlapped Loop Support in the Cydra 5,” *Proc. Third Int’l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 26-38, Apr. 1989.

[7] J.C. Dehnert and R.A. Towle, “Compiling for the Cydra 5,” *J. Supercomputing*, vol. 7, nos. 1/2, pp. 181-228, May 1993.

[8] A.E. Eichenberger and E.S. Davidson, “Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule,” *Proc. 28th Ann. Int’l Symp. Microarchitecture (MICRO-28)*, pp. 338-349, Nov. 1995.

[9] A.E. Eichenberger, E.S. Davidson, and S.G. Abraham, “Optimum Modulo Schedules for Minimum Register Requirements,” *Proc. Int’l Conf. Supercomputing*, pp. 31-40, July 1995.

[10] R. Govindarajan, E.R. Altman, and G.R. Gao, “Minimal Register Requirements Under Resource-Constrained Software Pipelining,” *Proc. 27th Ann. Int’l Symp. Microarchitecture (MICRO-27)*, pp. 85-94, Nov. 1994.

[11] P.Y.T. Hsu, “Highly Concurrent Scalar Processing,” PhD thesis, Univ. of Illinois, Urbana-Champaign, 1986.

[12] R.A. Huff, “Lifetime-Sensitive Modulo Scheduling,” *Proc. Sixth Conf. Programming Language, Design, and Implementation*, pp. 258-267, 1993.

[13] S. Jain, “Circular Scheduling: A New Technique to Perform Software Pipelining,” *Proc. ACM SIGPLAN ‘91 Conf. Programming Language Design and Implementation*, pp. 219-228, June 1991.

[14] M.S. Lam, “Software Pipelining: An Effective Scheduling Technique for VLIW Machines,” *Proc. SIGPLAN ’88 Conf. Programming Language Design, and Implementation*, pp. 318-328, June 1988.

[15] M.S. Lam, *A Systolic Array Optimizing Compiler*. Kluwer Academic, 1989.

[16] J. Llosa, “Reducing the Impact of Register Pressure on Software Pipelining,” PhD thesis, UPC. Universitat Politècnica de Catalunya, Jan. 1996.

[17] J. Llosa, M. Valero, and E. Ayguadé, “Heuristics for Register-Constrained Software Pipelining,” *Proc. 29th Ann. Int’l Symp. Microarchitecture (MICRO-29)*, pp. 250-261, Dec. 1996.

[18] J. Llosa, M. Valero, E. Ayguadé, and A. González, “Hypernode Reduction Modulo Scheduling,” *Proc. 28th Ann. Int’l Symp. Microarchitecture (MICRO-28)*, pp. 350-360, Nov. 1995.

[19] J. Llosa, M. Valero, E. Ayguadé, and J. Labarta, “Register Requirements of Pipelined Loops and Their Effect on Performance,” *Proc. Second Int’l Workshop Massive Parallelism: Hardware, Software, and Applications*, pp. 173-189, Oct. 1994.

[20] W. Mangione-Smith, S.G. Abraham, and E.S. Davidson, “Register Requirements of Pipelined Processors,” *Proc. Int’l Conf. Supercomputing*, pp. 260-246, July 1992.

[21] Q. Ning and G.R. Gao, "A Novel Framework of Register Allocation for Software Pipelining," *Conf. Rec. 20th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pp. 29-42, Jan. 1993.
[22] S. Ramakrishnan, "Software Pipelining in PA–RISC Compilers," *Hewlett-Packard J.*, pp. 39-45, July 1992.
[23] B.R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Ann. Int'l Symp. Microarchitecture*, pp. 63-74, Nov. 1994.
[24] B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proc. 14th Ann. Microprogramming Workshop*, pp. 183-197, Oct. 1981.
[25] B.R. Rau, M. Lee, P. Tirumalai, and P. Schlansker, "Register Allocation for Software Pipelined Loops," *Proc. ACM SIGPLAN '92 Conf. Programming Language Design and Implementation*, pp. 283-299, June 1992.
[26] B. Su and J. Wang, "GURPR*: A New Global Software Pipelining Algorithm," *Proc. 24th Ann. Int'l Symp. Microarchitecture (MICRO-24)*, pp. 212-216, Nov. 1991.
[27] P. Tirumalai, M. Lee, and M.S. Schlansker, "Parallelisation of Loops with Exits on Pipelined Architectures," *Proc. Supercomputing '90*, pp. 200-212, Nov. 1990.
[28] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su, "Decomposed Software Pipelining: A New Perspective and a New Approach," *Int'l J. Parallel Programming*, vol. 22, no. 3, pp. 357-379, 1994.
[29] N.J. Warter, G.E. Haab, and J.W. Bockhaus, "Enhanced Modulo Scheduling for Loops with Conditional Branches," *Proc. 25th Int'l Symp. Microarchitecture*, pp. 170-179, Dec. 1992.
[30] N.J. Warter and N. Partamian, "Modulo Scheduling with Multiple Initiation Intervals," *Proc. 28th Int'l Symp. Microarchitecture*, pp. 111-118, Nov. 1995.

Josep Llosa received his degree in computer science in 1990 and his PhD in computer science in 1996, both from the Polytechnic University of Catalonia (UPC) in Barcelona (Spain) In 1990, he joined the Computer Architecture Department at UPC, where he is presently an associate professor. His research interests include processor microarchitecture, memory hierarchy, and compilation techniques, with a special emphasis on instruction scheduling.

Mateo Valero obtained his Telecommunication Engineering Degree from the Polytechnic University of Madrid in 1974 and his PhD from the Polytechnic University of Catalonia (UPC) in 1980. He is a professor in the Computer Architecture Department at the Polytechnic University of Catalonia. His current research interests are in the field of high performance architectures with a special strength on the following topics: processor organization, memory hierarchy, interconnection networks, compilation techniques, and computer benchmarking.

Dr. Valero has been distinguished with several awards: Narcis Monturiol presented by the Catalan Government, the Salva i Campillo presented by the Telecommunication Engineer Association, and the King Jaime I by the Generalitat Valenciana. He is a member of the IEEE, and director of the C4 (Catalan Center for Computation and Communications). Since 1994, he has been a member of the Spanish Engineering Academy.

Eduard Ayguade received the engineering degree in telecommunications in 1986 and the PhD degree in computer science in 1989, both from the Polytechnic University of Catalunya (UPC), Barcelona (Spain). In 1986, he joined the Computer Architecture Department of the UPC, where he is presently a full professor. His main research interests include novel processor and memory organizations, and optimizing compilers for high-performance superscalar and parallel architectures.

Antonio González received his degree in computer science in 1986 and his PhD in computer science in 1989, both from the Polytechnic University of Catalonia at Barcelona (Spain). He is currently an associate professor in the Computer Architecture Department at the Polytechnic University of Catalonia. His research interests center on computer architecture, compilers, and parallel processing, with a special emphasis on processor microarchitecture, memory hierarchy, and instruction scheduling. Dr. Gonzalez is a member of the IEEE Computer Society and the ACM.