Lifetime-Sensitive Modulo Scheduling in a Production Environment

Josep Llosa, Eduard Ayguadé, Antonio Gonzalez, *Member*, *IEEE Computer Society*, Mateo Valero, *Fellow*, *IEEE*, and Jason Eckhardt

Abstract—This paper presents a novel software pipelining approach, which is called Swing Modulo Scheduling (SMS). It generates schedules that are near optimal in terms of initiation interval, register requirements, and stage count. Swing Modulo Scheduling is a heuristic approach that has a low computational cost. This paper first describes the technique and evaluates it for the Perfect Club benchmark suite on a generic VLIW architecture. SMS is compared with other heuristic methods, showing that it outperforms them in terms of the quality of the obtained schedules and compilation time. To further explore the effectiveness of SMS, the experience of incorporating it into a production quality compiler for the Equator MAP1000 processor is described; implementation issues are discussed, as well as modifications and improvements to the original algorithm. Finally, experimental results from using a set of industrial multimedia applications are presented.

Index Terms—Fine grain parallelism, instruction scheduling, loop scheduling, software pipelining, register requirements, VLIW, superscalar architectures.

1 Introduction

Software pipelining [5] is an instruction scheduling technique that exploits instruction level parallelism out of loops by overlapping successive iterations of the loop and executing them in parallel. The key idea is to find a pattern of operations (named the kernel code) so that, when repeatedly iterating over this pattern, it produces the effect that an iteration is initiated before the previous ones have completed.

The drawback of aggressive scheduling techniques, such as software pipelining, is their high register pressure. The register requirements increase as the concurrency increases [27], [22], due to either machines with deeper pipelines or wider issue or a combination of both. Registers, like functional units, are a limited resource. Therefore, if a schedule requires more registers than available, some actions, such as adding spill code, have to be performed. The addition of spill code can degrade performance [22] due to additional cycles in the schedule or due to memory interferences.

Some research groups have targeted their work toward exact methods that find the optimal solution to the problem. For instance, the proposals in [16] search the entire scheduling space to find the optimal resource-constrained schedule with minimum buffer requirements, while the proposals in [2], [7], [13] find schedules with the actual minimum register requirements. The task of generating an optimal (in terms of throughput and register requirements) resource-constrained schedule for loops is known to be NP-hard. All these exact approaches require a prohibitive time to construct the schedules and, therefore, their applicability is restricted to very small loops. Therefore, practical algorithms use some heuristics to guide the scheduling process. Some of the proposals in the literature only care about achieving high throughput [11], [19], [20], [31], [32], [37], while other proposals have also been targeted toward minimizing the register requirements [9], [12], [18], [24], which result in more effective schedules.

Stage Scheduling [12] is not a whole modulo scheduler by itself, but a set of heuristics targeted to reduce the register requirements of any given modulo schedule. This objective is achieved by moving operations in the schedule. The resulting schedule has the same throughput, but lower register requirements. Unfortunately, there are constraints in the movement of operations that might yield to suboptimal reductions of the register requirements. Similar heuristics have been included in the IRIS [9] scheduler, which is based on the Iterative Modulo Scheduling [11], [31] in order to reduce the register pressure at the same time as the scheduling is performed.

Slack Scheduling [18] is a heuristic technique that simultaneously schedules some operations late and other operations early with the aim of reducing the register requirements and achieving maximum execution rate. The algorithm integrates recurrence constraints and critical-path considerations in order to decide when each operation is scheduled. The algorithm is similar to Iterative Modulo Scheduling in the sense that it uses a limited amount of backtracking by possibly ejecting operations already scheduled to give place to a new one.

Hypernode Reduction Modulo Scheduling (HRMS) [24], [25] is a heuristic strategy that tries to shorten loop variant lifetimes, without sacrificing performance. The main contribution of HRMS is the node ordering strategy. The

• *J. Llosa, E. Ayguade, A. Gonzalez, and M. Valero are with the Computer Architecture Department, Technical University of Catalonia, c/ Jordi Girona 1-3, Modul D6, 08034, Barcelona, Spain. E-mail: mateo@ac.upc.es.*
• *J. Eckhardt is with the Department of Computer Science, Rice University, Houston, Texas.*

Manuscript received 12 Dec. 1999; revised 22 Sept. 2000; accepted 14 Nov. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111112.

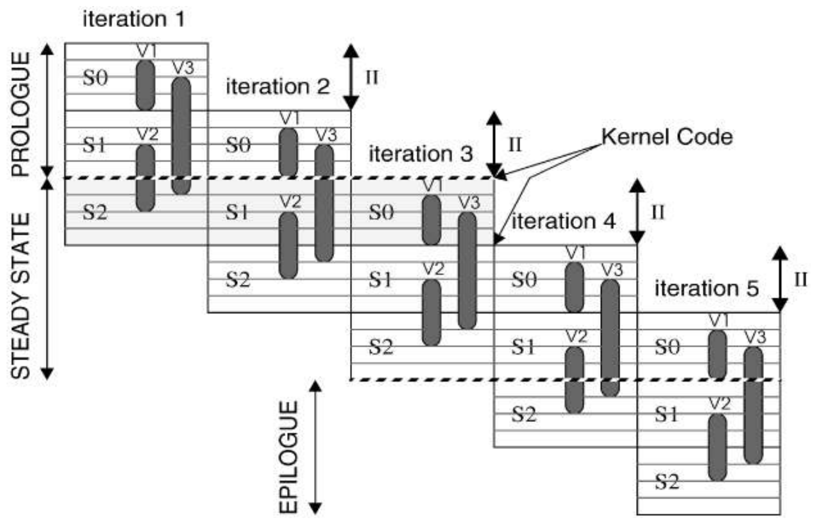


Fig. 1. Basic execution model for a software pipelined loop.

ordering phase sorts the nodes before scheduling them such that only predecessors or successors of a node can be scheduled before it is scheduled (except for recurrences). During the scheduling step, the nodes are scheduled as soon/late as possible if predecessors/successors have been previously scheduled. The effectiveness of HRMS has been compared in terms of achieved throughput and compilation time against other heuristic methods [18], [37] showing better performance. The main drawback of HRMS is that the scheduling heuristic does not take into account the criticality of the nodes.

In this paper, we present a novel scheduling strategy, *Swing Modulo Scheduling* (SMS[1]), which considers the criticality of the nodes. It is a heuristic technique that has a low computational cost (e.g., compiling all the innermost loops without conditional exits and procedure calls of the Perfect Club takes less than half a minute). The paper also describes its implementation in a production compiler for specific VLIW processors targeting digital consumer products. The performance figures reveal the efficiency of the schedules generated on a variety of customer workloads.

The rest of the paper is organized as follows: Section 2 presents an overview of the main concepts underlying software pipelining. Section 3 discusses an example to motivate our proposal, which is formalized in Section 4. Section 5 shows the main results of our experimental evaluation of the schedules generated by SMS. Section 6 is devoted to describing the experience of incorporating SMS into a production compiler and its evaluation on some real workloads. The main concluding remarks are given in Section 7.

2 Overview of Software Pipelining

In a software-pipelined loop, the schedule for an iteration is divided into stages so that the execution of consecutive iterations which are in distinct stages is overlapped. The number of stages in one iteration is termed stage count *(SC)*. The number of cycles between the initiation of successive iterations (i.e., the number of cycles per stage) in a software pipelined loop is termed the Initiation Interval *(II)* [32]. Fig. 1 shows a simple example with the execution of a software-pipelined loop composed of three operations (V1, V2, and V3). In this example, II = 4 and SC = 3.

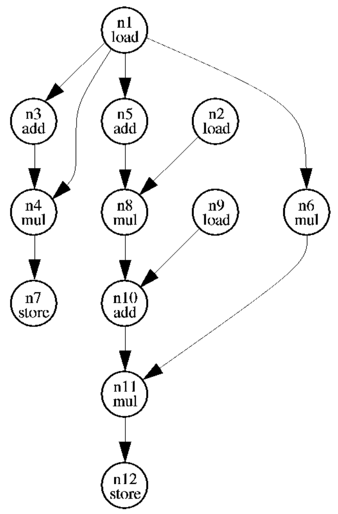
The Initiation Interval *II* between two successive iterations is bounded by both recurrence circuits in the graph *(RecMII)* and resource constraints of the architecture *(ResMII)*. This lower bound on the *II* is termed the Minimum Initiation Interval *(MII = max(RecMII, ResMII))*. The reader is referred to [11], [31] for an extensive dissertation on how to calculate *ResMII* and *RecMII*.

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. Each loop-invariant has a single value for all iterations of the loop thus requiring one register regardless of the schedule and the machine configuration.

For loop-variants, a value is generated in each iteration of the loop and, therefore, there is a different lifetime 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. This is the main reason why the register requirements are increased. In addition, for values with a lifetime larger than the *II*, new values are generated before the previous ones are used. To fix this problem, software solutions (modulo variable expansion [21]) as well as hardware solutions (rotating register files [10], [17]) have been proposed.

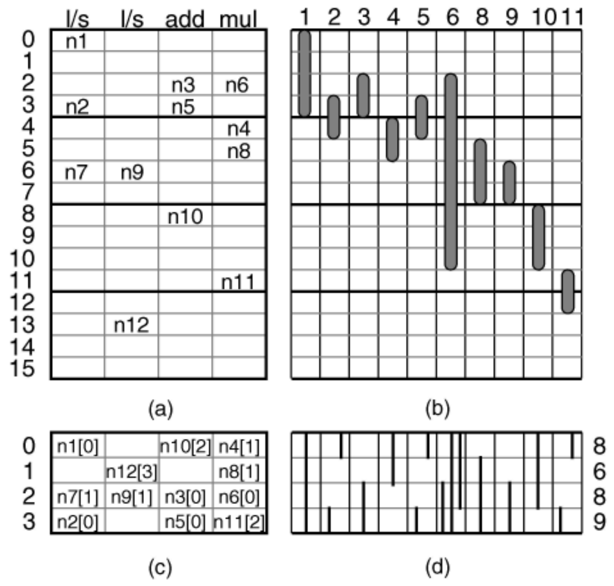
Some of the software pipelining approaches can be regarded as the sequencing of two independent steps: node ordering and node scheduling. These two steps are performed assuming *MII* as the initial value for *II*. If it is not possible to obtain a schedule with this *II*, the scheduling step is performed again with an increased *II*. The next section shows how the ordering step influences the register requirements of the loop.

1. This paper extends the previously proposed SMS technique [23] with novel features targeted to a specific DSP processor. It also includes performance figures for an industrial workload.



Hardware configuration:
1 add unit
1 mul unit
2 load/store units
Latencies:
add: 2 cycles
mul: 2 cycles
load: 2 cycles
store: 1 cycle

Fig. 2. Dependence graph for the motivating example.



	l/s	l/s	add	mul
0	n1			
1				
2			n3	n6
3	n2		n5	
4				n4
5				n8
6	n7	n9		
7				
8			n10	
9				
10				
11				n11
12				
13		n12		
14				
15				

0	n1[0]		n10[2]	n4[1]
1		n12[3]		n8[1]
2	n7[1]	n9[1]	n3[0]	n6[0]
3	n2[0]		n5[0]	n11[2]

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

3 Motivating Example

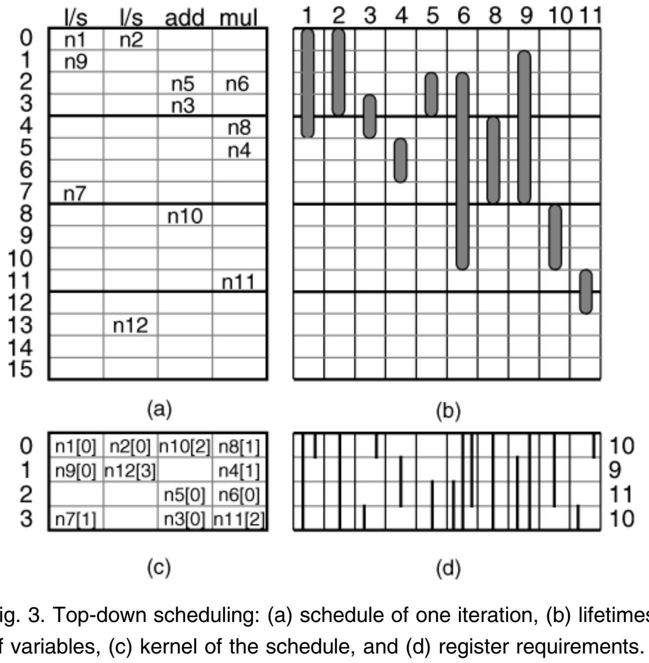
Consider the dependence graph in Fig. 2 and an architecture configuration with the pipelined functional units and latencies specified in the same figure. Since the graph in Fig. 2 has no recurrence circuits, its initiation interval is constrained only by the available resources; in this case, the most constraining resource is the multiplier, which causes *MII = 4/1 = 4*.

A possible approach to order the operations to be scheduled would be to use a top-down strategy that gives priority to operations in the critical path; with this ordering, nodes would be scheduled in the following order: <*n1, n2, n5, n8, n9, n3, n10, n6, n4, n11, n12, n7*>. Fig. 3a shows the top-down schedule for one iteration and Fig. 3c the kernel code (numbers in brackets represent the stage to which the operation belongs). Fig. 3b shows the lifetimes of loop variants. The lifetime of a loop variant starts when the producer is issued and ends when the last consumer is issued. Fig. 3d shows the register requirements for this schedule; for each cycle, it shows the number of live values required by the schedule. The maximum number of simultaneously live values at any cycle can approximate the number of registers required, which is called *MaxLive* (in [33] it is shown that register allocation never required more than *MaxLive + 1* registers for a large number of loops). In Fig. 3d, *MaxLive = 11*. Notice that, with this approach, variables generated by nodes *n2* and *n9* have an unnecessarily large lifetime due to the early placement of the corresponding operations in the schedule; as a consequence, the register requirements for the loop increase.

In HRMS [24], the ordering is done with the aim that all operations (except for the first one) have a previously scheduled reference operation. For instance, for the previous example, they would suggest the following order to schedule operations <*n1, n3, n5, n6, n4, n7, n8, n10, n11, n9, n2, n12*>. Notice that, with this scheduling order, both *n2* and *n9* (the two conflicting operations in the top-down strategy) have a reference operation (*n8* and *n10*, respectively) already scheduled when they are going to be placed in the partial schedule.

Fig. 4a shows the final schedule for one iteration. For instance, when operation *n9* is scheduled, operation *n10* has already been placed in the schedule (at cycle 8), so it will be scheduled as close as possible to it (at cycle 6), thus reducing the lifetime of the value generated by *n9*. Something similar happens with operation *n2*, which is placed in the schedule once its successor is scheduled. Fig. 4b shows the lifetimes of loop variants and Fig. 4d shows the register requirements for this schedule. In this case, *MaxLive = 9*.

The ordering suggested by HRMS does not give preference to operations in the critical path. For instance, operation *n5* should be scheduled two cycles after the initiation of operation *n1*; however, this is not possible since, during this cycle, the adder is busy executing



	l/s	l/s	add	mul
0	n1	n2		
1	n9			
2			n5	n6
3			n3	
4				n8
5				n4
6				
7	n7			
8			n10	
9				
10				
11				n11
12				
13		n12		
14				
15				

0	n1[0]	n2[0]	n10[2]	n8[1]
1	n9[0]	n12[3]		n4[1]
2			n5[0]	n6[0]
3	n7[1]		n3[0]	n11[2]

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

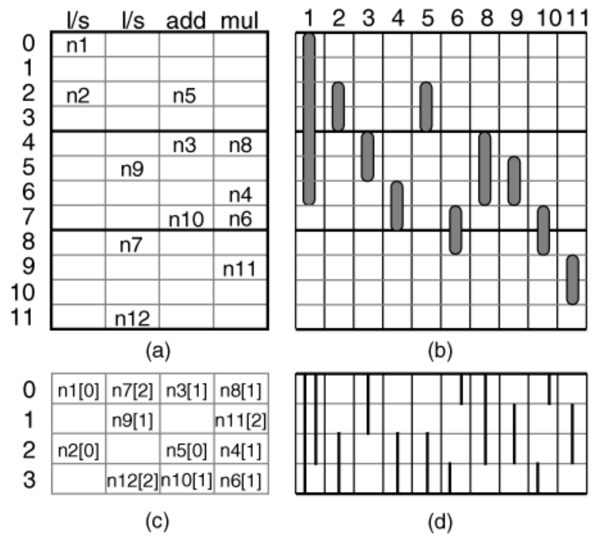


Fig. 5. SMS scheduling: (a) schedule of one iteration, (b) lifetimes of variables, (c) kernel of the schedule, and (d) register requirements.

operation *n3*, which has been scheduled before. Due to that, an operation in a more critical path *(n5)* is delayed in front of another operation that belongs to a less critical path *(n3)*. Something similar happens with operation *n11* that conflicts with the placement of operation *n6*, which belongs to a less critical path but the ordering has selected it before. Fig. 5a and Fig. 5c show the schedule obtained by our proposal and Fig. 5b and Fig. 5d the lifetime of variables and register requirements for this schedule. *MaxLive* for this schedule is 8. The schedule is obtained using the following ordering *<n12, n11, n10, n8, n5, n6, n1, n2, n9, n3, n4, n7>*. Notice that nodes in the critical path are scheduled with a certain preference with respect to the others. The following section details the algorithm that orders the nodes based on these ideas and the scheduling step.

4 Swing Modulo Scheduling (SMS)

Most modulo scheduling approaches consist of two steps. First, they compute a schedule trying to minimize the *II*, but without caring about register pressure, and then variables are allocated to registers. The execution time of a software pipelined loop depends on the *II*, the maximum number of live values of the schedule *(MaxLive)* and the stage count. The *II* determines the issue rate of loop iterations. Regarding the second factor, if *MaxLive* is not higher than the number of available registers, then the computed schedule is feasible and then it does not influence the execution time. Otherwise, some action must be taken in order to reduce the register pressure. Some possible solutions outlined in [33] and evaluated in [22] are:

- Reschedule the loop with an increased *II*. In general, increasing the *II* will reduce *MaxLive*, but it decreases the issue rate.
- Add spill code. This again has a negative effect since it increases the required memory bandwidth and it will result in more memory penalties (e.g., cache misses). In addition, memory may become the most utilized resource and, therefore, adding spill code may require an increase of the *II*.

Finally, the stage count determines the number of iterations of the epilogue part of the loop (it is exactly equal to the stage count minus one).

Swing Modulo Scheduling (SMS) is a modulo scheduling technique that tries to achieve a minimum *II*, reduce *MaxLive*, and minimize the stage count. It is a heuristic technique that has a low computational cost while producing schedules very close to those generated by optimal approaches based on exhaustive search, which have a prohibitive computational cost for real programs. In order to have this low computation cost, SMS schedules each node only once (unlike other methods that are based on backtracking [9], [11], [18], [31]. Despite not using backtracking, SMS produces effective schedules because nodes are scheduled in a precomputed order that guarantees certain properties, as described in Section 4.2.

In order to achieve a minimum *II* and to reduce the stage count, SMS schedules the nodes in an order that takes into account the *RecMII* of the recurrence to which each node belongs (if any) and as a secondary factor it considers the criticality of the path to which the node belongs.

To reduce *MaxLive*, SMS tries to minimize the lifetime of all the values of the loop. To achieve that, it tries to keep every operation as close as possible to both its predecessors and successors. When an operation is to be scheduled, if the partial schedule has only predecessors, it is scheduled as soon as possible. If the partial schedule contains only successors, it is scheduled as late as possible. The situation in which the partial schedule contains both predecessors and successors of the operation to be scheduled is undesirable since in this case, if the lifetime from the predecessors to the operation is minimized, the lifetime from the operation to its successors is increased. This situation happens only for one node in each recurrence and it is avoided completely if the loop does not contain any recurrence.

The algorithm followed by SMS consists of the following three steps which are described in detail below:

- computation and analysis of the dependence graph,
- ordering of the nodes,
- scheduling.

SMS can be applied to generate code for innermost loops without subroutine calls. Loops containing conditional statements (IF) can be handled after applying *if-conversion* [1] and provided that either the processor supports predicated execution [10] or reverse *if-conversion* [38] follows pipelining.

4.1 Computation and Analysis of the Dependence Graph

The *dependence graph* of an innermost loop consists of a set of four elements $(DG = \{V, E, \delta, \lambda\})$:

- V is the set of nodes (vertices) of the graph, where each node $v \in V$ corresponds to an operation of the loop.
- E is the set of edges, where each edge $(u, v) \in E$ represents a dependence from operation u to

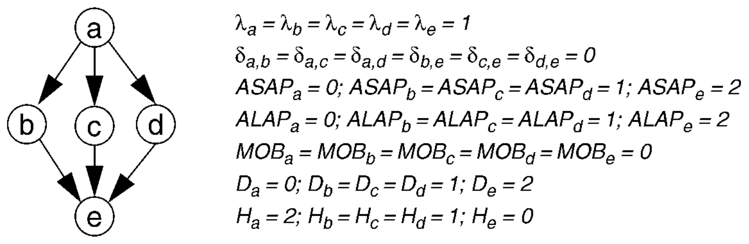

a
b
c
d
e
$\lambda_a = \lambda_b = \lambda_c = \lambda_d = \lambda_e = 1$
$\delta_{a,b} = \delta_{a,c} = \delta_{a,d} = \delta_{b,e} = \delta_{c,e} = \delta_{d,e} = 0$
$ASAP_a = 0; ASAP_b = ASAP_c = ASAP_d = 1; ASAP_e = 2$
$ALAP_a = 0; ALAP_b = ALAP_c = ALAP_d = 1; ALAP_e = 2$
$MOB_a = MOB_b = MOB_c = MOB_d = MOB_e = 0$
$D_a = 0; D_b = D_c = D_d = 1; D_e = 2$
$H_a = 2; H_b = H_c = H_d = 1; H_e = 0$

Fig. 6. A sample dependence graph.

operation v. Only data dependences (flow, anti and output dependences) are included since the type of loops that SMS can handle only includes one branch instruction at the end that is associated to the iteration count. Other branches have been previously eliminated by the if-conversion phase.

- $\delta_{u,v}$ is called the distance function. It assigns a nonnegative integer to each edge $(u, v) \in E$. This value indicates that operation v of iteration I depends on operation u of iteration $I - \delta_{u,v}$.
- λ_u is called the latency function. For each node of the graph, it indicates the number of cycles that the corresponding operation takes.[2]

Given a node $v \in V$ of the graph, *Pred(v)* is the set of all the predecessors of v. That is, $Pred(v) = \{u | u \in V and\ (u, v) \in E\}$. In a similar way, *Succ(v)* is the set of all the successors of v. That is, $Succ(v) = \{u | u \in V and\ (v, u) \in E\}$.

Once the dependence graph has been computed, some additional functions that will be used by the scheduler are calculated. In order to avoid cycles, one backward edge of each recurrence is ignored for performing these computations. These functions are the following:

- $ASAP_u$ is a function that assigns an integer to each node of the graph. It indicates the earliest time at which the corresponding operation could be scheduled. It is computed as follows:

$$\begin{aligned}&If\ Pred(u) = \emptyset\ \ then\ ASAP_u = 0\\ &else\ ASAP_u = max(ASAP_v + \lambda_v - \delta_{v,u} \times MII)\\ &\forall v \in Pred(u).\end{aligned}$$

- $ALAP_u$ is a function that assigns an integer to each node of the graph. It indicates the latest time at which the corresponding operation could be scheduled. It is computed as follows:

$$\begin{aligned}&If\ Succ(u) = \emptyset\ then\ ALAP_u = max(ASAP_v) \forall v \in V\\ &else\ ALAP_u = min(ALAP_v - \lambda_u + \delta_{u,v} \times MII)\\ &\forall v \in Succ(u).\end{aligned}$$

- MOB_u is called the mobility (slack) function. For each node of the graph, it denotes the number of time slots at which the corresponding operation could be scheduled. Nodes in the most critical path have a mobility equal to zero and the mobility will increase as the path in which the operation is located is less critical. It is computed as follows:

$$MOB_u = ALAP_u - ASAP_u.$$

- D_u is called the depth of each node. It is defined as its maximum distance to a node without predecessors. It is computed as follows:

$$\begin{aligned}&If\ Pred(u) = \emptyset\ \ then\ D_u = 0\\ &else\ D_u = max(D_v + \lambda_v) \forall v \in Pred(u).\end{aligned}$$

- H_u is called the height of each node. It is defined as the maximum distance to a node without successors. It is computed as follows:

$$\begin{aligned}&If\ Succ(u) = \emptyset\ \ then\ H_u = 0\\ &else\ H_u = max(H_v + \lambda_u) \forall v \in Succ(u).\end{aligned}$$

2. In some architectures, the latency of an operation may also depend on the consumer operation (i.e., $\lambda_{u,v}$). The techniques presented in this paper can be easily adapted to handle this situation.

4.2 Node Ordering

The ordering phase takes as input the dependence graph previously calculated and produces an ordered list containing all the nodes of the graph. This list indicates the order in which the nodes of the graph will be analyzed by the scheduling phase. That is, the scheduling phase (see the next section) first allocates a time slot for the first node of the list, then it looks for a suitable time slot for the second node of the list, and so on. Notice that as the number of nodes already placed in the partial schedule increases there are more constraints to be met by the remaining nodes and, therefore, it is more difficult to find a suitable location for them.

As previously outlined, the target of the ordering phase is twofold:

- Give priority to the operations that are located in the most critical paths. In this way, the fact that the last operations to be scheduled should meet more constraints is offset by their higher mobility (MOB_u). This approach tends to reduce the *II* and the stage count.
- Try to reduce *MaxLive*. In order to achieve this, the scheduler will place each node as close as possible to both its predecessors and successors. However, the order in which the nodes are scheduled has a severe impact on the final result. For instance, assume the sample dependence graph of Fig. 6 and a dual-issue processor.

If node *a* is scheduled at cycle 0 and then node e is scheduled at cycle 2 (that is, they are scheduled based on their *ASAP* or *ALAP* values), it is not possible to find a suitable placement for nodes *b*, *c*, and *d* since there are not enough slots between *a* and *e*. On the other hand, if nodes *a* and *e* are scheduled too far away, there are many possible locations for the remaining nodes. However, *MaxLive* will be too high no matter which possible schedule is chosen. For instance, if we try to reduce the lifetime from *a* to *b*, we are increasing by the same amount the lifetime from *b* to *e*. In general, having scheduled both predecessors and successors of a node before scheduling it may result in a poor schedule. Because of this, the ordering of the nodes tries to avoid this situation whenever possible (notice that, in the case of a recurrence, it can be avoided for all the nodes except one).

If the graph has no recurrences, the intuitive idea to achieve these two objectives is to compute an ordering based on a traversal of the dependence graph. The traversal starts with the node at the bottom of the most critical path and moves upward, visiting all the ancestors. The order in which the ancestors are visited depends on their depth. In the case of equal depth, nodes are ordered from less to more mobility. Once all the ancestors have been visited, all the descendants of the already ordered nodes are visited but now moving downward and in the order given by their height. Successive upward and downward sweeps of the graph are performed alternately until the entire graph has been traversed.

If the graph has recurrences, the graph traversal starts at the recurrence with the highest *RecMII* and applies the previous algorithm considering only the nodes of the recurrence. Once this subgraph has been traversed, the nodes of the recurrence with the second highest *RecMII* are traversed. At this step, the nodes located on any path between the previous and the current recurrence are also considered in order to avoid having scheduled both predecessors and successors of a node before scheduling it. When all the nodes belonging to recurrences or any path among them have been traversed, then the remaining nodes are traversed in a similar way.

Concretely, the ordering phase is a two-level algorithm. First, a partial order is computed. This partial order consists of an ordered list of sets. The sets are ordered from the highest to the lowest priority set, but there is no order within each set. Each node of the graph belongs to just one set.

The highest priority set consists of all the nodes of the recurrence with the highest *RecMII*. In general, the *i*th set consists of the nodes of the recurrence with the *i*th highest *RecMII*, eliminating those nodes that belong to any previous set (if any) and adding all the nodes located in any path that joins the nodes in any previous set and the recurrence of this set. Finally, the remaining nodes are grouped into sets of the same priority, but this priority is lower than that of the sets containing recurrences. Each one of these sets consists of the nodes of a connected component of the graph that do not belong to any previous set.

Once this partial order has been computed, then the nodes of each set are ordered to produce the final and complete order. This step takes as input the previous list of sets and the whole dependence graph. The sets are handled in the order previously computed. For each recurrence of the graph, a backward edge is ignored in order to obtain a graph without cycles. The final result of the ordering phase is a list of ordered nodes O containing all the nodes of the graph.

The ordering algorithm is shown in Fig. 7, where | denotes the list append operation and *Succ_L(O)* and *Pred_L(O)* are the sets of predecessors and successors of a list of nodes, respectively, which are defined as follows:

$$Pred\_L(O) = \{v | \exists u \in O \textit{ such that } v \in Pred(u) \textit{ and } v \notin O\}$$
$$Succ\_L(O) = \{v | \exists u \in O \textit{ such that } v \in Succ(u) \textit{ and } v \notin O\}.$$

4.3 Filling the Modulo Reservation Table

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

- If an operation *u* has only predecessors in the partial schedule, then *u* is scheduled as soon 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 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 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 both predecessors and successors, then the scheduler computes $Early\_Start_u$ and $Late\_Start_u$ as described above and scans the partial schedule starting at cycle $Early\_Start_u$ until the cycle $min(Late\_Start_u, Early\_Start_u + II - 1)$. This situation will only happen for exactly one node of each recurrence circuit.

```
O := Empty_list
For each set of nodes S in decreasing priority do
  if (Pred_L(O) $\cap$ S) $\neq \emptyset$ then
    R := Pred_L(O) $\cap$ S
    order := bottom-up
  else if (Succ_L(O) $\cap$ S) $\neq \emptyset$ then
    R := Succ_L(O) $\cap$ S
    order := top-down
  else
    R := {node with the highest ASAP value in S} ;
       if more than one, choose anyone
    order := bottom-up
  end if

  Repeat
    if order = top-down
     while R $\neq \emptyset$ do
       v := Element of R with the highest $H_v$ ;
          if more than one, choose node
          with lowest $MOB_u$
       O := O | <v>
       R := R - {v} $\cup$ (Succ (v) $\cap$ S)
     endwhile
     order := bottom-up
     R := Pred_L(O) $\cap$ S
    else
     while R $\neq \emptyset$ do
       v := Element of R with the highest $D_v$ ;
          if more than one, choose node
          with lowest $MOB_u$
       O := O | <v>
       R := R - {v} $\cup$ (Pred(v) $\cap$ S)
     endwhile
     order := top-down
     R := Succ_L(O) $\cap$ S
    endif
  until R = $\emptyset$
endfor
```

Fig. 7. Ordering algorithm.

- 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 provide more opportunities for finding free slots. One of the advantages of our proposal is that the nodes are ordered only once, even if the scheduling step has to do several trials.

4.4 Examples

This section illustrates the performance of SMS by means of two examples. The first example is a small loop without recurrences and the second example uses a dependence graph with recurrences.

Assume that the dependence graph of the body of the innermost loop to be scheduled is that of Fig. 2, where all the edges represent dependences of distance zero. Assume also a four-issue processor with four functional units (one adder, one multiplier, and two load/store units) fully pipelined with the latencies listed in Fig. 2.

The first step of the scheduling is to compute the *MII* and the *ASAP*, *ALAP*, *mobility*, *depth*, and *height* of each node of the graph. *MII* is equal to 4. Table 1 shows the remaining values for each node.

Then, the nodes are ordered. The first level of the ordering algorithm groups all the nodes into the same set since there are not recurrences. Then, the elements of this set are ordered as follows:

- Initially, *R = {n12}* and *order = bottom-up*.
- Then, all the ancestors of *n12* are ordered depending on their depth and their mobility as a secondary factor. This gives the partial order *O = <n12, n11, n10, n8, n5, n6, n1, n2, n9>*.
- Then, the order shifts to top-down and all the descendants are ordered based on their height and mobility. This gives the final ordering *O = <n12, n11, n10, n8, n5, n6, n1, n2, n9, n3, n4, n7>*.

TABLE 1
ASAP, ALAP, Mobility (M), Depth (D), and Height (H) of Nodes in Fig. 2

Node	ASAP	ALAP	M	D	H
n1	0	0	0	0	10
n2	0	2	2	0	8
n3	2	6	4	2	4
n4	4	8	4	4	2
n5	2	2	0	2	8
n6	2	6	4	2	4
n7	6	10	4	6	0
n8	4	4	0	4	6
n9	0	4	4	0	6
n10	6	6	0	6	4
n11	8	8	0	8	2
n12	10	10	0	10	0

The next step is to schedule the operations following the previous order. *II* is initialized to *MII* and the operations are scheduled, as shown in Fig. 5:

- The first node of the list, *n12*, is scheduled at cycle 10 (given by its *ASAP*) since there are neither predecessors nor successors in the partial schedule.[3] Once the schedule is folded, this will become cycle 3 of stage 2.
- For the remaining nodes, the partial schedule contains either predecessors or successors of it, but not both of them. Nodes are scheduled as close as possible to their predecessors/successors. For instance, node *n11* is scheduled as late as possible since the partial schedule only contains the successor of it. Because of resource constraints, this is not always possible as it happens for nodes *n8* and *n3*. For instance, *n8* tries to be scheduled as late as possible, which should be cycle 5 in Fig. 5. However, at this cycle, the multiplier is already occupied by *n11*, which forces node *n8* to move one cycle above.

The second example consists of a loop with a more complex dependence graph with recurrences, as depicted in Fig. 8. We will assume a four-issue machine with four general-purpose functional units fully pipelined and with two-cycle latency.

In this example, *MII* is equal to 6. The first step of the ordering phase is to group nodes into an ordered list of sets. As a result, the following list of three sets is obtained:

- *S1 = {A, C, D, F}*. This is the first set since it contains the recurrence with the highest *RecMII* (i.e., $(3 \text{ nodes} \times 2 \text{ cycles})/(1 \text{ distance}) = 6$).
- *S2 = {G, J, M, I}*. This is the set that contains the second recurrence

$$(RecMII = (3 \text{ nodes} \times 2 \text{ cycles})/(2 \text{ distance}) = 3)$$

and the nodes in all paths between *S1* and this recurrence (i.e., node *I*).

3. In fact, the resulting schedule stretches from cycles -1 to 10, but, in all the figures, we have normalized the representation, always starting at cycle 0, so *n12* is in cycle 11 of Fig. 5.

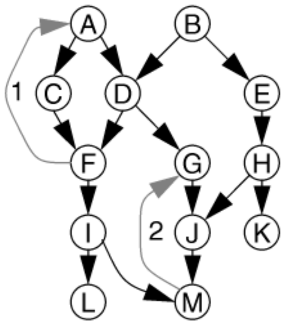


Fig. 8. A sample dependency graph.

- *S3 = {B, E, H, K, L}*. This is the set with all remaining nodes.

Then, the nodes are ordered as follows:

- First, the nodes of *S1* are ordered, producing the partial order *O = <F, C, D, A>*.
- Then, the ordering algorithm computes the predecessors of these four nodes, but finds that none of them belongs to *S2*. It then computes the successors and finds that *I* and *G* belong to *S2*, so it proceeds with a top-down sweep. This produces the following partial ordering: *O = <F, C, D, A, G, I, J, M>*.
- Finally, the nodes of *S3* are considered. The traversal proceeds with the predecessor of *S1* and *S2* and performs a bottom-up sweep which produces the partial order *O = <F, C, D, A, G, I, J, M, H, E, B>*. Then, the direction shifts to top-down and all the successors are traversed producing the final order: *O = <F, C, D, A, G, I, J, M, H, E, B, L, K>*.

The scheduling phase generates the schedule shown in Fig. 9.

5 Performance Evaluation

5.1 Experimental Framework

In this section, we present some results of our experimental study. We compare SMS with two other scheduling methods: HRMS and Top-Down.[4] Both methods have been implemented in C++ using the LEDA libraries [29]. For this evaluation, we used all the innermost loops of the Perfect Club benchmark suite [4] that have neither subroutine calls nor conditional exits. Subroutine calls prevent the loops from being software pipelined (unless they are inlined). Although loops with conditional exits can be software pipelined [36], this experimental feature has not been added to our scheduler and is out of the scope of this work. Loops with conditional structures in their bodies have been IF-converted [1] so that they behave as a single basic block loop. The dependence graphs of the loops have been obtained with the compiler described in [3].

A total of 1,258 loops that represent the 78 percent of the total execution time of the Perfect Club (measured on a HP-PA 735) have been scheduled. From those loops, 438 (34.8 percent) have recurrence circuits, 18 (1.4 percent) have conditionals, and 67 (5.4 percent) have both, while the

4. A comparison with other scheduling approaches can be found in [23].

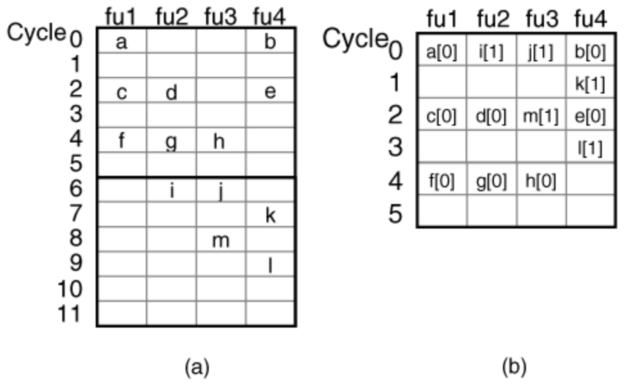

Cycle
fu1 fu2 fu3 fu4
0 1 2 3 4 5 6 7 8 9 10 11
a b c d e f g h i j k m l
(a)
a[0] i[1] j[1] b[0] k[1] c[0] d[0] m[1] e[0] l[1] f[0] g[0] h[0]
(b)

Fig. 9. SMS scheduling of the dependence graph of Fig. 6. (a) Schedule of one iteration and (b) kernel of the scheduling.

remaining 735 (58.4 percent) loops have neither recurrences nor conditionals. Also, 152 (12 percent) of the loops have nonpipelined operations (i.e., modulo operations, divisions, and square roots) that complicate the scheduling task. The scheduled loops have a maximum of 376 nodes and 530 dependence edges, even though the average is slightly more than 16 nodes and 20 edges per graph.

We assume unit latency for store instructions, a latency of 2 for loads, a latency of 4 for additions and multiplications, a latency of 17 for divisions, and a latency of 30 for square roots. The loops have been scheduled for a machine configuration with two load/store units, two adders, two multipliers, and two Div/Sqrt units. All units are fully pipelined except the Div/Sqrt units, which are not pipelined at all.

5.2 Performance Results

Table 2 shows some performance figures for the three schedulers. Notice that SMS obtains an *II* equal to the *MII* for more loops than the other methods. It also requires fewer registers and obtains schedules with fewer stages than the other methods. In general, it produces results much better than the Top-Down scheduler, somewhat better than HRMS and very close to the optimal (SMS only fails to obtain a schedule with *II* = *MII* for 18 loops; in other words, it is optimal for at least 98.6 percent of the loops). There is only one parameter (stage count, *SC*) for which it obtains worse results than the Top-Down scheduler, but it is due to the fact that Top-Down obtains larger initiation intervals. Larger initiation intervals mean that less parallelism is exploited and that less overlapping between iterations is obtained, requiring, in general, fewer stages but a higher execution time. Despite this, notice that SMS has smaller initiation intervals than HRMS, but it requires slightly fewer stages. This is because SMS has been designed to optimize all three parameters: *II*, register requirements, and *SC*.

TABLE 2
Comparison of Performance Metrics for the Three Schedulers

Metric	SMS	HRMS	Top-Down
Σ II	8815	8839	10113
II / MII	1.0101	1.0128	1.1588
Loops with II > MII	18	31	157
Σ SC	5302	5408	5039
Σ Regs.	15157	15332	17183

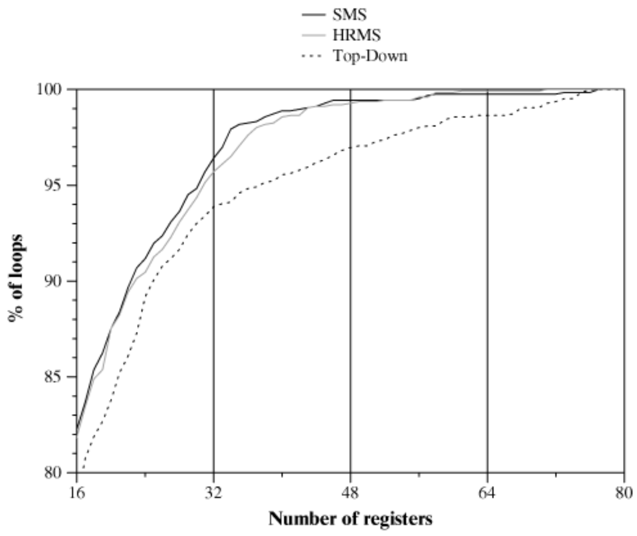


Fig. 10. Cumulative distribution of the register requirements of loop-variants

Once the loops have been scheduled, a lower bound of the register requirements *(MaxLive)* can be found by computing the maximum number of live values at any cycle of the schedule. As shown in [33], the actual register allocation almost never requires more than *MaxLive + 1* registers; therefore, we use *MaxLive* as a measurement of the register requirements.

Fig. 10 shows the cumulative distribution of the register requirements for the three schedulers. Each point *(x, y)* in the graph represents that *y* percent of the loops can be scheduled with *x* registers or less. Since SMS and HRMS have the objective of minimizing the register requirements, there is little difference among them, even though SMS is slightly better in all aspects. This plot only considers the register requirements caused by the loop variants; the requirements for the loop invariants do not depend on the quality of the scheduling.

5.3 Compilation Time

In the context of using software pipelining as a code generation technique, it is also important to consider the cost of computing the schedules. In fact, this is the main reason why integer linear programming approaches are not used. The time to produce the schedule has, for instance, extreme importance when dynamic rescheduling techniques are used [6]. Fig. 11 compares the execution time of the three schedulers running on a Sparc-10/40 workstation. SMS only requires 27.5 seconds to schedule the 1,258 loops of the Perfect Club. Fig. 11 also compares the time required to compute the *MII*, to order the nodes (or compute the priority of the nodes), and the time required to perform the scheduling. Notice that Top-Down (which is the simplest scheduler) requires less time than the others to compute the priority of the nodes, but, surprisingly, it requires much more time to schedule the nodes. This is because, when the scheduler fails to find a schedule with *MII* cycles, the loop is

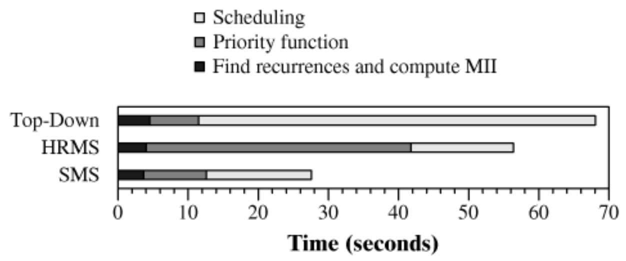


Fig. 11. Time to schedule all the 1,258 loops in the Perfect Club.

rescheduled with an increased initiation interval, and Top-Down has to reschedule the loops much more often than the other schedulers.

HRMS obtains much better schedules (requiring less time to schedule the loops) at the expense of a sophisticated and more time-consuming preordering step. SMS uses a simple, but very effective, heuristic to order the nodes that requires almost the same time as Top-Down to order the nodes and the same time as HRMS to schedule them. In total, it is about twice as fast as the two other schedulers.

6 SMS in a Production Compiler

In this section, we describe an industrial implementation of SMS in the Equator Technologies, Inc. (ETI) optimizing compiler (introduced in [8]). ETI is a descendent of Multiflow Computer, Inc. [26] that produces a family of VLIW processors for digital consumer products.

6.1 Target Architecture

ETI's MAP1000 processor is the target architecture used here. It is the first implementation of ETI's series of Media Accelerated Processors (MAP). The experiments were executed on a preproduction (engineering prototype) MAP1000 chip running at 170 MHz. The MAP1000 is a quad-issue VLIW processor, composed of two identical clusters *cl0* and *cl1*, as depicted in Fig. 12.

Each cluster contains:

- I-ALU unit (32-bit integer, load/store, and branch subunits),
- IFG-ALU unit (single-precision floating-point, DSP, and 64-bit integer subunits),
- general register file (64×32-bit registers),
- predicate register file (16×1-bit predicate registers),
- special-purpose *PLV* register (1×128 bits),
- special-purpose *PLC* register (1×128 bits).

An instruction word is 136-bits long and consists of four operations to drive the two clusters. Most operations can only be executed on either an I-ALU or an IFG-ALU. However, some operations, such as simple integer operations, can execute on both units, which gives a software pipeliner more flexibility when placing them. All functional units except for the divide units are fully pipelined and thus can accept a new operation on every cycle.

Branch instructions are delayed—the branch is issued at cycle i, but does not commit until cycle $i + 2$. Thus, there are 11 operations in the "delay slots" that must be filled (three operations in the instruction word containing the branch plus eight operations in the two following instruction words). This is significant for modulo scheduling as it forces *MinII* to be at least three cycles and those kernels with three cycles of work or less execute entirely in the delay slots. Of course, it is sometimes necessary to unroll small loops in order to produce enough work to populate these cycles.

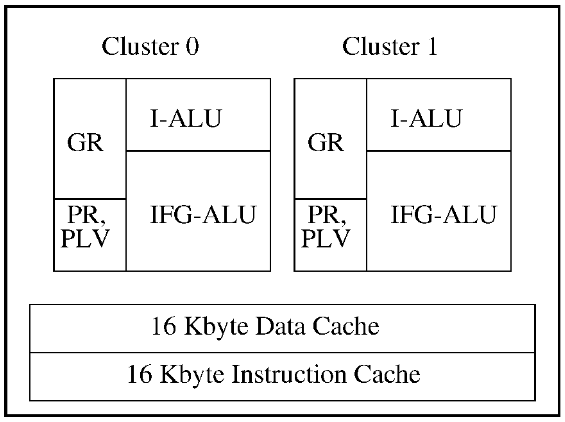


Fig. 12. MAP1000 block diagram.

The architecture contains limited support for software-pipelined loops, including a fully predicated instruction set (supporting *if-conversion*, for example) and speculative memory operations. Further, a *select* instruction is provided which selects one of two general register inputs based on a third predicate input. There is no other hardware support for overlapped loops—specifically, there are no rotating register files or corresponding loop instructions. Thus, for each pipelined loop, we must generate prologue and epilogue compensation and at least one copy of the compacted kernel (more if there are any lifetimes that overlap themselves).

Processor resources must be managed precisely by the compiler as there is no hardware interlocking (with the exceptions of bank stalls and cache misses). Further, due to the clustered functional units and register files, the compiler must be concerned with the cost of data movement between clusters. Cross-cluster movement can be accomplished by a simple register-register copy operation or by result *broadcasting*. Broadcasting refers to the ability of an operation to target its result to both the local register file as well as a register file on a remote cluster.

Each general-purpose register file holds integer, floating-point, and multimedia data. The registers are viewed as a set of overlapping classes depending on the instructions used to write or read them. These classes present complications for the software pipeliner and register allocator. Instructions with a *restricted* register operand, for example, must read the operand from r0 through r7 or r16 through r23. Further, instructions that broadcast can only write destination registers r0 through r15. Finally, 64-bit instructions read and write register pairs rN:rN+1 (where N is even and the instruction references N).

One class of operations, the *sigma* operations, needs special mention as they significantly affect the implementation of SMS. Consider one such operation

```
srshinprod.ps64.ps16 rd, rs, iw, ip,
```

where `rd` and `rs` are general register pairs and `iw` and `ip` are immediate operands:

$$PLV = rs[8 \times (ip + iw) - 1 : 8 \times ip] \mid PLV[127 : 8 \times iw]$$

$$rd = \sum_{i=0}^{7} PLC.16i \times PLV.16i.$$

The notation *[x : y]* denotes a range of bits and $x \mid y$ represents concatenation of bits. The operation first updates the PLV register by shifting it to the right with the leftmost bits being replaced by bits from `rs`. Then, an inner product is computed into `rd` by treating the 128-bit PLC and PLV registers each as a vector of eight signed 16-bit integers. Due to the fact that there is only one PLV register per cluster, it is not possible to have more than one corresponding lifetime intersecting in any given cycle (on the same cluster). This causes a problem for software pipelining which attempts to overlap operations. Section 6.2 addresses the issue in more detail, along with a method to handle such operations.

6.2 Improvements and Modifications to SMS

While the addition of SMS to the existing software pipeliner was fairly straightforward, there were a few aspects that needed special attention. Some modifications were done without changing the essential characteristics of SMS, but rather to allow it to perform better when dealing with the complexities presented by the target VLIW architecture.

First of all, the interaction of the ordering algorithm with the ETI dependence graph structure presented a problem. Consider the section of the algorithm in Fig. 7 between lines 15 and 19 (and the analogous section between lines 23 and 27). An implicit assumption made here is that nodes are topologically distinguishable based on their height or depth. That is, it is assumed that nodes with dependence relationships will have distinct values for height and depth that correspond to their topological position in the graph. This is a reasonable assumption, yet the ETI graph structure does not satisfy it because some nodes may have an associated latency of 0. In fact, a negative latency will exist for nodes that constrain a branch due to the branch delay slots of the architecture. For example, if the latency of a node (with one successor) is 0, then that node and its successor will have the same height and depth. In these cases, the SMS algorithm cannot rely on just the height/depth values since they can be ambiguous when nonpositive latencies are involved. A simple modification is made in the ETI version so that, when choosing between related nodes (e.g., lines 16 and 24 in Fig. 7), the intervening graph edges are examined and not just the *height/depth* values. In other words, we pay attention to the full graph structure when height and depth don't give a complete characterization of the graph topology. This is slightly more expensive, but is only necessary in compilers with graph representations allowing nodes with nonpositive latencies, which is a feature not used by many compilers.

A second modification relates to a special group of operations that are particularly troublesome for pipelining. The sigma operations, in addition to using the general registers, rely on an exclusive register resource. This special-purpose 128-bit register *(PLV)* is larger than the general registers but only one of them exists per cluster.

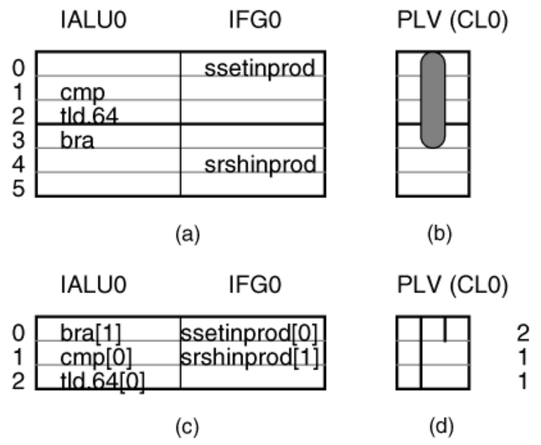


Fig. 13. Illegal sigma-op schedule: (a) schedule of one iteration, (b) lifetime of PLV register only, (c) kernel, (d) PLV register requirements.

Sigma operations read the value of the cluster-local *PLV* and write a new value to it (in addition to the general register destination). Because there is only one *PLV* (per cluster), the modifying operations must be issued sequentially. Typically, sigma operations appear in chains of four or more (at least in the programs developers are producing). Since these instructions appear in groups and they read/modify an exclusive resource, it is important that they be issued as close as possible to each other. This increases the chance that the *MII* will be achieved or that the chains can be issued at all. In most cases, the kernel is not large relative to the size of the chain and, so, issuing them atomically is crucial. To achieve this, SMS has been extended by treating a chain of sigma operations like a recurrence, that is, as a high priority subgraph. During the first phase of SMS, chains are detected and a separate set is created for each one. The sets are ordered with the longest chains having highest priority. These sets are prioritized higher than recurrence sets since the resources consumed on a recurrence will likely prevent a chain from being able to be issued. It is usually easier to schedule all other nodes around the chains. Fig. 13 depicts the problem presented by the exclusive *PLV* resource. The `ssetinprod` operation initializes the *PLV* register while the `srshinprod` operation consumes the *PLV*, resulting in the lifetime shown in Fig. 13b. However, the *PLV* requirement of the kernel (Fig. 13d) is greater than one in cycle 0, which is illegal. Normal operations can simply write to another register, but, for sigma operations, the scheduler must ensure that such a situation never arises.

Two additional improvements aim at obtaining the smallest possible *II* by ensuring good resource compaction. SMS tries to simultaneously optimize a number of factors, such as register pressure, *II*, and *Stage Count*. Sometimes, optimizing one factor can have a negative impact on another. In a few cases, the optimization of register usage by SMS produced schedules with a larger *II* than could have been achieved. This does not happen frequently, but can be seen more often on machines with very complex resource usage. The behavior has been observed on the target VLIW, which has multiple clusters and end-of-pipeline resource contention, such as register write ports (all managed by the scheduler). Because of the resource complexity, it is possible to place certain operations which will prevent others from being placed even when there are enough total resources available. The third modification involves the choice of nodes that occur at various points in the algorithm: lines 10, 16, and 24. In all three cases, there is the possibility that more than one node matching the criteria will be available to choose from. The original algorithm will arbitrarily pick one of the multiple nodes. The actual node picked could depend on the data structure tracking the nodes or other factors. In this way, an undesirable node might be chosen which would later lock-out another node, thereby increasing the *II*. The modified version replaces the random choice with a guess based on resource consumption of the node. For instance, if one node is a long latency, nonpipelined divide operation and the other a single-cycle addition operation, it is assumed choosing the first would probably result in better resource compaction. Similarly, if we notice that, between two nodes, one is more constraining in terms of read or write port resources than the other, we choose it.

The fourth modification tries to obtain tight resource compaction by adding a symmetric case in the initialization of work set *R* (lines 3 to 12). The order of the conditions gives preference to the bottom-up direction which is desirable in most cases. The symmetric case below would give preference to the top-down direction instead:

```
3'  if $(Succ\_L(O) \cap S) \neq \emptyset$ then
4'      $R := Succ\_L(O) \cap S$
5'      order := top-down
6'  else if $(Pred\_L(O) \cap S) \neq \emptyset$ then
7'      $R := Pred\_L(O) \cap S$
8'      order := bottom-up
9'  else
10'     $R := \{$node with the smallest ASAP value in $S\}$;
            if more than one, choose anyone
11' order := top-down
12' end if
```

In the actual implementation, the loop is first scheduled with the original method and only scheduled a second time with the symmetric case if *MII* is not achieved. If the second attempt results in the same *II* as the first (or larger), then the first schedule is chosen since the bottom-up preference usually produces better lifetime reduction given the same *II*. A similar idea of using multiple scheduling attempts at each *II* is also used by the SGI MIPSpro compiler described in [34] and the PGI i860 compiler [28]. However, neither of those compilers simultaneously schedules some operations bottom-up and others top-down as in the SMS method.

6.3 Performance Results

In this section, we evaluate the effectiveness of SMS compared to the original ETI modulo scheduler and validate that the effort spent implementing it in a production compiler was worthwhile. The experiments here are based on a small number of critical customer application programs from the areas of signal processing and 2D/3D graphics as well as some benchmark codes. Table 3 describes the industrial workbench.

TABLE 3
Industrial Workbench

Name	Description
UWICSL	A library of signal and image processing routines from the University of Washington Image Computing Systems Lab.
Omika	A customer benchmark for 3D geometry setup and lighting calculations.
Rasterizer	Selected routines from a software 3D rasterizer.
MPEG2	ETI's MPEG2 video decoder.
Modem	A software v.34 modem.
NAS	A C language version of the NAS Parallel Benchmarks appbt and appsp.

There are 75 total loops with the following characteristics: 15 (20 percent) contain nontrivial recurrences; 17 (22.7 percent) contain conditionals; and five (6.7 percent) contain both recurrences and conditionals. While a detailed instruction breakdown is not presented, many of the loops contain complex operations such as nonpipelined divides and chains of sigma operations which complicate scheduling.

We first compare the two schedulers from the point of view of initiation interval, stage count (SC), replication factor (RF), and register requirements assuming an infinite number of registers. The replication factor is the number of copies of the kernel needed to perform modulo variable expansion [20]. Later, we consider the addition of spill code and its effect in performance when a finite number of registers is considered (64 registers per cluster).

Table 4 compares some performance metrics. The total register requirements are shown as well as per-cluster totals *(CL0, CL1)*. First of all, both schedulers achieve the *MII* for all the loops except two (one loop in UWICSL and one in the NAS APPBT applications) due to resource conflicts; they obtain the same *II* in the two loops.

The average number of registers per loop is 47.2 for SMS compared to 55.6 for top-down. Further, a detailed analysis of the individual results show that, of the 75 loops, SMS uses fewer registers than top-down in 63 of them. In six other cases, the register usage is identical. The top-down scheduler uses fewer registers than SMS in only six of the

TABLE 4
Static Comparison of the Two Schedulers in ETI before Adding Spill Code

Metric	SMS	Top-Down
Σ II (average)	1670 (22.3)	1670 (22.3)
II / MII	1.0103	1.0103
Σ SC (average)	291 (3.9)	328 (4.4)
Σ RF (average)	203 (2.7)	272 (3.6)
Σ CL0 / Σ CL1	1823 / 1720	2109 / 2063
Σ Regs. (average)	3543 (47.2)	4172 (55.6)

TABLE 5
Analysis of Individual Loop Results—Registers and Replication Factor

Method	Registers			Replications		
	<	=	>	<	=	>
SMS	63	6	6	44	26	5
Top-down	6	6	63	5	26	44

loops. Table 5 also shows that SMS performs considerably better than top-down in terms of *RF* (more on this later).

Although the average register requirements are reasonable for the architecture we are considering, it is important to look to the requirements of the individual loops in more detail. Fig. 14 shows the cumulative register requirements for this workbench. Notice that, in this case, the register pressure for this collection of software pipelined loops is much higher than the pressure for the loops in the Perfect Club. In particular, SMS is able to schedule only 45 percent and 81 percent of the loops with 32 and 64 registers, respectively; the original top-down approach does the same for only 40 percent and 71 percent of the loops.

The target architecture described earlier includes a 128-register file organized as two clusters of 64 registers each. For this configuration, we see that only two loops need the addition of spill code when scheduled using SMS; however, six loops need spill code when scheduled using the top-down approach. Table 6 shows the final *II* that is obtained after adding spill code and rescheduling the loops.

What is a bit more interesting than a static loop-level comparison is the dynamic speed up of the applications containing the affected loops. In the MPEG2 application, for example, the bottleneck loop (accounting for 70 percent of total run-time) was significantly slower when scheduled with top-down than with SMS. This is due to a larger final *II*, extra memory references resulting from spill code, and a larger replication factor that affected the instruction cache performance. As shown in Table 7, the resulting total

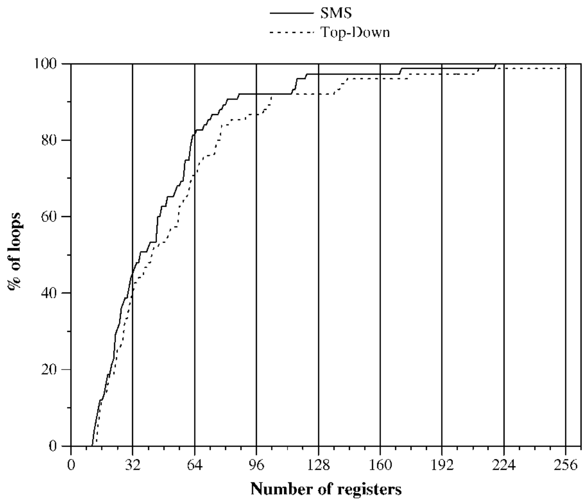


Fig. 14. Cumulative distribution of the register requirements for the industrial workbench.

TABLE 6
Static Comparison of the Two Schedulers in ETI after Adding Spill Code (128 Registers)

Metric	SMS	Top-Down
Σ II (average)	1702 (22.7)	1735 (23.1)
II / MII	1.03	1.05
# of loops w/ spills	2	6

MPEG2 speedup is 11 percent when compiled with SMS rather than top-down. This is one of ETI's most critical applications, so obtaining this improvement with a simple recompile is exciting. Also shown are the other affected applications and their speed-ups.

On the 128-register MAP1000, most of the loops are scheduled without needing spill code. Even in cases where spill code is not necessary, it is still important to reduce register pressure. An inner-loop reduction of register requirements can increase the availability of registers to outer-loops and to the overall surrounding code. However, we can get a better idea of the positive impact of SMS on this particular workbench by assuming a smaller number of registers. To this end, another experiment was performed by forcing the compiler to assume only 64 total general-purpose registers (two clusters of 32 registers each). The results of rescheduling the loops with the smaller configuration are shown in Table 8. For this trial, 13 loops required spilling with SMS, whereas 21 loops required spilling with top-down. One loop from the UWICSL suite could not be compiled at all with 64 registers due to its very high register requirements. This loop was excluded from the computations in the table.

As seen in Table 9, SMS compiled applications have significantly better dynamic run-time in 12 instances. As expected, SMS is more effective as the register file size decreases. The MPEG speed-up in the 64-register model was slightly less than in the 128-register model because, this time, both SMS and top-down schedules had some amount of spilling (there was no spilling for SMS in the 128-register case). The dynamic results indicate that, for typical RISC processors with 32 registers, lifetime-sensitive modulo scheduling would be very beneficial.

Finally, because SMS reduces register lifetimes, it seems intuitive that the replication factor might also be reduced. The experiments support this intuition. On average, loops scheduled with SMS require one less copy of the kernel than loops scheduled with the top-down scheduler (Table 4). While not completely unexpected, the results shown in

TABLE 7
Dynamic Speed-Up of SMS Compiled Applications of the Prototype MAP1000 (128 Registers)

Application or function	SMS (cycles)	Top-Down (cycles)	Speed-up
UWICSL: hurst	10,124,474	10,657,342	5%
Omika	983,627	1,101,662	12%
MPEG2	5,812,230	6,451,576	11%
Modem	38,315,392	39,081,699	2%

TABLE 8
Static Comparison of the Two Schedulers in ETI after Spill Code Is Added (64 Registers)

Metric	SMS	Top-Down
Σ II (average)	1390 (18.8)	1471 (19.9)
II / MII	1.06	1.12
# of loops w/ spills	13	21

Fig. 15 were a bit surprising. SMS was able to schedule 54 percent of the loops with two replications and 92 percent of the loops with four replications. Top-down, on the other hand, scheduled only 21 percent and 77 percent of the loops with two and four replications, respectively. Since most of the literature on modulo scheduling assumes that the target architectures have rotating register files, little attention is given to the replication factor issue. However, the VLIW targeted here has a relatively small instruction cache and code size reduction is very important. Examination of the replication factors shows a possible area for improvement in the ETI pipeliner, even considering the SMS results.

6.4 Additional Implementation Observations

Finally, we outline three additional aspects, primarily related to the target processor architecture, that have not been included in the current implementation and that need further attention. Future research is needed to determine how these aspects interact with our lifetime-sensitive software pipelining.

First, SMS was not originally designed to take into account the limited connectivity of clustered (partitioned) architectures (e.g., the MAP1000) and the data movement required when assigning functional units. This implies that minimizing lifetimes may not necessarily produce a register reduction since an imbalance across clusters may result. Even so, SMS performs quite well on average and the problem is rarely observed in the applications compiled. It is possible that the behavior would be more pronounced on a machine with more than two clusters. SMS has been extended to deal with clustered architectures [[35]. Other

TABLE 9
Dynamic Speed-Up of SMS Compiler Applications on the Prototype MAP1000 (64 Registers)

Application or function	SMS (cycles)	Top-Down (cycles)	Speed-up
UW: median3	1,182,304	1,288,712	9%
UW: affine8	2,696,915	3,020,544	12%
UW: persp8	4,912,518	5,860,634	19%
UW: rotate8	751,798	856,000	14%
UW: rgb2cmyk	2,825,179	2,912,760	3%
UW: div8	261,238	276,387	6%
UW: cfft	4,110,541	4,521,592	10%
Omika	1,658,147	1,940,033	17%
Rasterizer	1,118,834	1,208,341	8%
MPEG2	7,174,036	7,834,048	9%
NAS	61,697,495	64,165,395	4%
Modem	39,614,976	40,644,966	3%

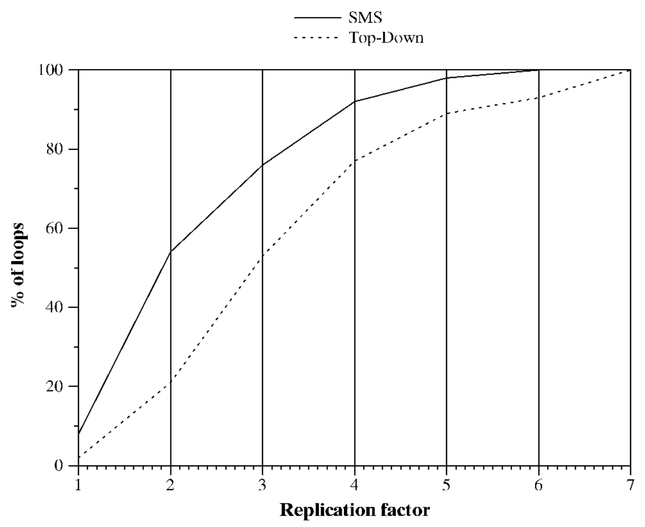


Fig. 15. Cumulative distribution of the replication factors for the industrial workbench.

modulo scheduling techniques for clustered architectures can be found elsewhere [14], [30].

Second, the broadcast feature mentioned earlier contributes to register pressure on both clusters simultaneously, so it is worthwhile to make any register-sensitive scheduler take it into consideration. Also important is determining which operations should broadcast their results. Another issue relates to certain operations requiring an operand to be in a restricted register—a register from a subset of the register file. These restricted register operands can cause high register pressure within the restricted subset (which is only 25 percent of the registers) even though there may not be high contention in the nonrestricted subset. It would be beneficial for SMS to take into account that these restricted lifetimes are usually more important to minimize than others.

And, third, applying loop unrolling before pipelining may provoke undesirable effects in the SMS algorithm. The dependence graph of a loop body unrolled n times will be roughly n times "wider" than it would be without unrolling. Further, each of the unrollends has the same length critical path as each of the others. SMS will begin ordering at the bottom-most node of one of the unrollends. It will then proceed to order all of the nodes at the same depth but from distant parts of the whole graph (i.e., from the different unrollends). Thus, the final node order may cause too wide a computation to be in progress at some point during scheduling. That is, too many simultaneously live values from distinct unrollends may consume all available registers. The problem is analogous to top-down list schedulers that order nodes in a breadth-first fashion, potentially causing too much parallelism and the corresponding increase in register pressure. One possible solution for reducing the register requirements would be to confine the ordering phase to smaller sections of the graph. For example, if it is assumed the unrolled graph contains no recurrences, then the current ordering phase is presented with one large set containing the entire graph (all the unrollends). This set could be partitioned into m new sets such that n/m unrollends are contained in each. During scheduling, the final ordering would allow a narrower computation with less register pressure, albeit probably at the expense of a larger stage count.

7 Conclusions

We have presented a novel software pipelining technique that is called *Swing Modulo Scheduling* (SMS). It is a heuristic technique that produces near optimal schedules in terms of initiation interval, prologue/epilogue size, and register requirements while requiring a very low compilation time.

The technique has been deeply evaluated using 1,258 loops of the Perfect Club that represent about 78 percent of the total execution time of this benchmark suite. We have shown that SMS outperforms other heuristic approaches in terms of quality of the obtained schedules, which is measured by the attained initiation interval, register requirements, and stage count. In addition, it requires less compilation time (about half of the time of the schedulers used for comparison).

In the paper, we have also evaluated an implementation of SMS in a production compiler for VLIW architectures targeted to digital consumer products. Experimental results show that it outperforms the original available software pipeliner implementation on a variety of customer workloads.

Acknowledgments

This work has been supported by the Ministry of Education of Spain (CICYT) under contract TIC98-0511.

References

[1] 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.

[2] E.R. Altman and G.R. Gao, "Optimal Modulo Scheduling through Enumeration," *Int'l J. Parallel Programming,* vol. 26, no. 3, pp. 313-344, 1988.

[3] E. Ayguade, C. Barrado, J. Labarta, D. Lopez, S. Moreno, D. Padua, and M. Valero, "A Uniform Representation for High-Level and Instruction-Level Transformations," Technical Report UPC-CEPBA 95-01, Universitat Politecnica de Catalunya, Jan. 1995.

[4] M. Berry, D. Chen, P. Koss, and D. Kuck, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," Technical Report 827, Center of 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] T.M. Conte and S.W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures," *Proc. 28th Int'l Ann. Symp. Microarchitecture,* pp. 208-218, Nov. 1995.

[7] J. Cortadella, R.M. Badia, and F. Sanchez, "A Mathematical Formulation of the Loop Pipelining Problem," *Proc. XI Design of Integrated Circuits and Systems Conf. (DCIS '96),* Oct. 1996.

[8] B.F. Cutler, "Deep Pipelines Schedule VLIW for Multimedia," *Electronic Eng. Times,* no. 1034, 9 Nov. 1998.

[9] A.K. Dani, V. Janaki, and R. Govindarajan, "Register-Sensitive Software Pipelining," *Proc. Merged 12th Int'l Parallel Processing Symp. and Ninth Int'l Symp. Parallel and Distributed Processing,* Mar. 1998.

[10] 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,* pp. 26-38, 1989.

[11] J.C. Dehnert and R.A. Towle, "Compiling for Cydra 5," *J. Supercomputing,* vol. 7, nos. 1/2, pp. 181-227, 1993.

[12] A.E. Eichenberger and E.S. Davidson, "Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule," *Proc. 28th Int'l Ann. Symp. Microarchitecture,* pp. 338-349, Nov. 1995.

[13] 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.

[14] M. Fernandes, J. Llosa, and N. Topham, "Distributed Modulo Scheduling," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture (HPCA '99),* pp. 130-134, Jan. 1999.

[15] P.N. Glaskowsky, "MAP1000 Unfolds at Equator," *Microprocessor Report,* vol. 12, no. 16, Dec. 1998.

[16] R. Govindarajan, E.R. Altman, and G.R. Gao, "Minimal Register Requirements under Resource-Constrained Software Pipelining," *Proc. 27th Int'l Ann. Symp. Microarchitecture,* pp. 85-94, Nov. 1994.

[17] L. Gwennap, "Intel Discloses New IA-64 Features," *Microprocessor Report,* vol. 13, no. 3, pp. 16-19, 8 Mar. 1999.

[18] R.A. Huff, "Lifetime-Sensitive Modulo Scheduling," *Proc. ACM SIGPLAN '93 Conf. Programming Language, Design and Implementation,* pp. 258-267, 1993.

[19] 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.

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

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

[22] J. Llosa, "Reducing the Impact of Register Pressure on Software Pipelined Loops," PhD thesis, UPC, Universitat Politècnica de Catalunya, Jan. 1996, http://www.ac.upc.es/hpc/HPC.ILP.html.

[23] J. Llosa, A. Gonzalez, M. Valero, and E. Ayguade, "Swing Modulo Scheduling: A Lifetime-Sensitive Approach," *Proc. Fourth Parallel Architectures and Compilation Techniques (PACT '96),* pp. 80-86, Oct. 1996.

[24] J. Llosa, M. Valero, E. Ayguade, and A. Gonzalez, "Hypernode Reduction Modulo Scheduling," *Proc. 28th Int'l Ann. Symp. Microarchitecture,* pp. 350-360, Nov. 1995.

[25] J. Llosa, M. Valero, E. Ayguade, and A. Gonzalez, "Modulo Scheduling with Reduced Register Pressure," *IEEE Trans. Computers,* vol. 47, no. 6, pp. 625-638, June 1998.

[26] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *J. Supercomputing,* vol. 7, nos. 1/2, pp. 51-142, 1993.

[27] W. Mangione-Smith, S.G. Abraham, and E.S. Davidson, "Register Requirements of Pipelined Processors," *Proc. Int'l Conf. Supercomputing,* pp. 260-271, July 1992.

[28] L. Meadows, S. Nakamoto, and V. Schuster, "A Vectorizing, Software Pipelining Compiler for LIW and Superscalar Architectures," *Proc. RISC '92,* Feb. 1992.

[29] K. Mehlhorn and S. Näher, "LEDA, a Library of Efficient Data Types and Algorithms," Technical Report TR A 04/89, Universität des Saarlandes, Saarbrücken, 1989 (available from ftp://ftp.mpi-sb.mpg.de/pub/LEDA).

[30] E. Nystrom and A.E. Eichenberger, "Effective Cluster Assignment for Modulo Scheduling," *Proc. 31st Int'l Symp. Microarchitecture,* pp. 103-114, Dec. 1998.

[31] B.R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Ann. Int'l Symp. Microarchitecture,* pp. 63-74, Nov. 1994.

[32] 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.

[33] 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.

[34] J. Ruttenberg, G.R. Gao, W. Lichtenstein, and A. Stoutchinin, "Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler," *Proc. ACM SIGPLAN '96 Conf. Programming Language Design and Implementation,* pp. 1-11, 1996.

[35] J. Sanchez and A. Gonzalez, "The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures," *Proc. Int'l Conf. Parallel Processing (ICPP '2000),* pp. 555-562, Aug. 2000.
[36] P. Tirumalai, M. Lee, and M.S. Schlansker, "Parallelisation of Loops with Exits on Pipelined Architectures," *Proc. Supercomputing '90,* pp. 100-212, Nov. 1990.
[37] 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.
[38] N.J. Warter, S.A. Mahlke, W.W. Hwu, and B.R. Rau, "Reverse If-Conversion," *Proc. SIGPLAN '93 Conf. Programming Language Design and Implementation,* pp. 290-299, June 1993.

Josep Llosa received his degree in computer science in 1990 and his PhD degree in computer science in 1996, both from the Polytechnic University of Catalonia (UPC), 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.

Eduard Ayguadé received the Engineering degree in telecommunications in 1986 and the PhD degree in computer science in 1989, both from the Universitat Politècnica de Catalunya (UPC), Spain. Since 1987, he has been lecturing on computer organization and architecture and optimizing compilers. Currently, and since 1997, he is a full professor in the Computer Architecture Department at UPC. His research interests cover the areas of processor microarchitecture and memory hierarchy, parallelizing compilers for high-performance multiprocessor systems, and tools for performance analysis and visualization. He has published more than 90 papers on these topics and participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs.

Antonio Gonzalez received his degree in computer science in 1986 and his PhD degree in computer science in 1989, both from the Universitat Politècnica de Catalunya, Barcelona, Spain. He has occupied different faculty positions in the Computer Architecture Department at the Universitat Politècnica de Catalunya since 1986, with tenure since 1990, and he is currently an associate professor in this department. 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.

Mateo Valero obtained his telecommunication engineering degree from the Polytechnic University of Madrid in 1974 and his PhD degree from the Polytechnic University of Catalonia (UPC) in 1980. He is a professor in the Computer Architecture Department at UPC. His current research interests are in the field of high performance architectures, with special interest in the following topics: processor organization, memory hierachy, interconnection networks, compilation techniques, and computer benchmarking. He has published approximately 200 papers on these topics. He served as the general chair for several conferences, including ISCA-98 and ICS-95, and has been an associate editor for *IEEE Transactions on Parallel and Distributed Systems* for three years. He is a member of the subcommittee for the Ecker-Mauchly Award. Dr. Valero has been honored with several awards, including the Narcis Monturiol, presented by the Catalan Goverment, the Salva i Campillo presented by the Telecommunications Engineer Association and ACM, and the King Jaime I by the Generalitat Valenciana. He is the director of the C4 (Catalan Center for Computation and Communications). Since 1994, he has been a member of the Spanish Engineering Academy and, since January 2001, he has been an IEEE fellow.

Jason Eckhardt is currently attending Rice University, where he is pursuing a PhD degree in computer science. Previously, he spent eight years designing and developing optimizing compilers for companies such as Convex Computer Corporation, Equator Technologies, and Cygnus. His research interests include instruction scheduling, high-level loop transformations, and processor microarchitecture.