Hypernode Reduction Modulo Scheduling *

Josep Llosa, Mateo Valero, Eduard Ayguadé and Antonio González

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Campus Nord, Mòdul D6, Gran Capità s/n
08071, Barcelona, SPAIN
{josepll,mateo,eduard,antonio}@ac.upc.es

Abstract

Software Pipelining is a loop scheduling technique that extracts parallelism from loops by overlapping the execution of several consecutive iterations. Most prior scheduling research has focused on achieving minimum execution time, without regarding register requirements. Most strategies tend to stretch operand lifetimes because they schedule some operations too early or too late.

The paper presents a novel strategy that simultaneously schedules some operations late and other operations early, minimizing all the stretchable dependencies and therefore reducing the registers required by the loop. The key of this strategy is a pre-ordering phase that selects the order in which the operations will be scheduled. The results show that the method described in this paper performs better than other heuristic methods and almost as well as a linear programming method but requiring much less time to produce the schedules.

Keywords: *Instruction Scheduling, Loop Scheduling, Software Pipelining, Register Allocation, Register Spilling.*

1 Introduction

Software pipelining is an instruction scheduling technique that exploits the instruction level parallelism of loops by overlapping successive iterations of the loop and executing them in parallel. Finding the optimal schedule is an NP-complete problem and there are several works that propose and evaluate different heuristic strategies to perform software pipelining [20, 12, 11, 23, 6, 19].

The drawback of aggressive scheduling techniques, such as software pipelining, is the high register pressure. The register requirements increase as the concurrency increases [16, 15], 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 scheduling requires more registers than available, some actions, such as adding spill code, have to be performed. The addition of spill code, can degrade performance [15] due to additional cycles in the scheduling, or due to memory interferences.

The problems introduced by the high register requirements of aggressive scheduling techniques, together with the trend of increasing ILP in current microprocessors [9, 24], have led to scheduling research oriented to minimize the register requirements (in part due to the limited number of registers that existing architectures have, and in part due to the limitations in chip area and especially access time, that register files with a high number of registers will impose). In this direction there are also proposals for alternative register file organizations [22, 4, 14].

In order to achieve maximum performance, scheduling algorithms that reduce the register pressure while scheduling for high throughput are of high interest. Huff's Slack Scheduling [10] is an heuristic technique that attempts to address this concern. SPILP [8] is an integer linear programming formulation of the scheduling problem that obtains the optimal resource-constrained schedule, with minimal buffer requirements. In [7] a linear programming formulation that obtains optimum schedules with the minimum register requirements is presented. Unfortunately heuristic strategies do not always obtain the optimum results and linear programming methods require a much higher time to construct the schedules than heuristic methods.

This paper presents *Hypernode Reduction Modulo Scheduling (HRMS)*, a heuristic strategy that tries to shorten loop variant lifetimes, without sacrificing performance. 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). During the scheduling step the nodes are scheduled as soon/late as possible, if predecessors/successors have been previously scheduled.

This strategy has been tested with a set of loops taken from [8] and compared against three leading scheduling strategies. These three strategies are the previous mentioned Slack and SPILP together with

*This work has been supported by the Ministry of Education of Spain under contracts TIC 880/92 and TIC 429/95, by ESPRIT 6634 Basic Research Action (APPARC) and by CEPBA (European Center for Parallelism of Barcelona).

1072-4451/95 $4.00 © 1995 IEEE
Proceedings of MICRO-28

FRLC [23] which is an heuristic strategy which does not take into consideration the register requirements. Experimental results show that HRMS obtains better schedules than the other heuristic strategies, with a comparable scheduling time. On the other hand, HRMS produces similar results to SPILP, but requires up to 2 orders of magnitude less time than SPILP to produce the schedules. In addition, HRMS is compared against a Top-Down scheduler [15] and characterized in terms of quality of the generated schedules and the computational cost on a test-bench of over a thousand loops from the Perfect Club Benchmark Suite [3] that account for 78% of the execution time of the Perfect Club.

In Section 2 an example is used to illustrate the problems that most strategies have, and shows how our strategy shortens lifetimes, and reduces register pressure. Section 3 describes our proposal (HRMS). Section 4 presents the experiments performed, and finally, Section 5 states our conclusions.

2 Overview of software pipelining and motivating example

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 scheduling is termed the **Initiation Interval**(*II*) [20].

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*). The reader is refered to [6, 19] for an extensive dissertation of 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. Loop-invariants, have only one value for all iterations of the loop, therefore each one requires one register for all the execution of the loop irrespective 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 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. In addition, for values with a lifetime larger than the *II* new values are generated before the previous one is used, overwriting it.

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** [13], 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, renaming different instantiations of a loop-variant at execution time [5].

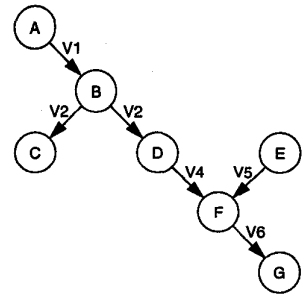

Figure 1: Dependence graph of our motivating example.

2.1 Motivating example

Consider the dependence graph of Figure 1, and an architecture where all the operations can be executed by any functional unit (i.e. general-purpose functional units). Assume that there are 4 pipelined units, and that the execution latency is 2. Since the graph in Figure 1 has no recurrence circuits, its initiation interval is constrained only by the available resources $MII = \lceil \frac{7}{4} \rceil = 2$.

In many approaches, the lifetimes of some values can be unnecessarily large. As an example, Figure 2a shows a top-down scheduling, and Figure 3a a bottom-up scheduling for the example graph.

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 soon as possible. Figure 2b shows the lifetimes of loop variants for the top-down scheduling assuming that a value is alive from the beginning of the producer operation to the beginning of the last consumer. Notice that loop variant V5 has an unnecessary 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 V5 (Figure 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 V2 has an unnecessary large lifetime due to the late placement of C.

In the strategy we propose, 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 pre-ordering 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 op-

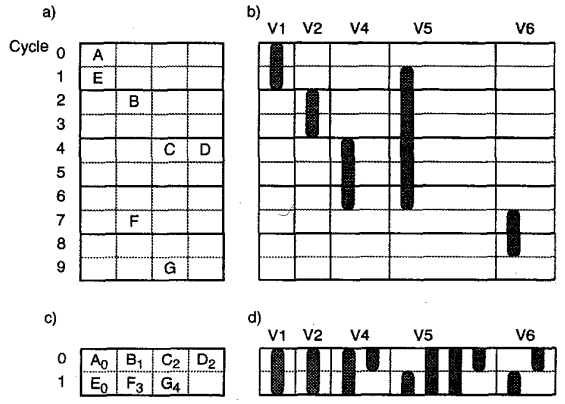

Figure 2: Top-Down scheduling: a) Schedule of one iteration, b) Lifetimes of variables, c) Kernel, d) Register requirements

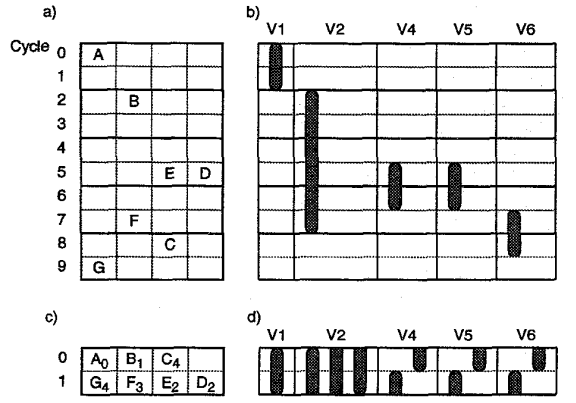

Figure 3: Bottom-Up scheduling: a) Schedule of one iteration, b) Lifetimes of variables, c) Kernel, d) Register requirements.

eration (except for the first operation to be scheduled). For instance, consider that nodes of the graph in Figure 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.

Figure 4a shows the final 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 soon 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 soon as possible in the scheduling, i.e. at cycle 9.

Figure 4b shows the lifetimes of loop variants. Notice that neither C nor E have been placed too late and too early respectively, 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 calculate a late start cycle for E. Figure 4d shows the number of alive registers in the kernel (Figure 4c) during the steady state phase of the execution of the loop. There are 6 alive registers in the first row and 5 in the second, therefore, the loop variants require only 6 registers. In contrast the top-down schedule requires 8 registers and the bottom-up schedule requires 7 registers.

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 $\delta_{(u,v)}$ 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. A recurrence circuit from an operation to an instance of itself Ω iterations later, must not be stretched beyond $\Omega \times II$. In addition, placing an operation u at a cycle t_u commits its associated resources for cycles $t_u + s \times II$, $\forall s$.

HRMS solves these problems by splitting the scheduling into two steps: A pre-ordering step that orders nodes and, the actual scheduling, that sched-

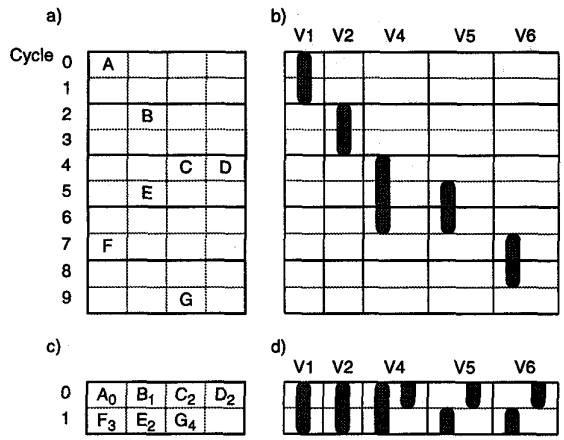

Figure 4: Bidirectional scheduling: a) Schedule of one iteration, b) Lifetimes of variables, c) Kernel, d) Register requirements.

```
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_Reduction(V',G,h);
        L' := Sort_ASAP(G');
        List := Concatenate(List,L')
return List
```

Figure 5: Function that pre-orders the nodes in a dependence graph without recurrence circuits

ules nodes (once at a time) in the order given by the pre-ordering step.

The pre-ordering 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, $G = (V, E, \delta, \lambda)$, to be ordered is a 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 pre-ordering 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 Pre-ordering 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 pre-ordering step (Figure 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*[1]. This node is inserted in the partial list of ordered nodes, then the pre-ordering algorithm sorts the rest of the nodes.

At each step, the predecessors (successors) of the *Hypernode* are obtained. Then the nodes that appear in any path among the predecessors (successors) are obtained (function *Search_All_Paths*) [2]. Once the predecessors (successors) and all the paths connecting them have been obtained, all these nodes are reduced (see function *Hypernode_Reduction* in Figure 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 algorithm we name PALA. 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 Figure 7a. Next, we illustrate the ordering of the nodes of this graph step by step.

[1]The algorithm tries to shorten lifetimes irrespective of the starting node. 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).

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

```
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'
```

Figure 6: Function *Hypernode_Reduction*

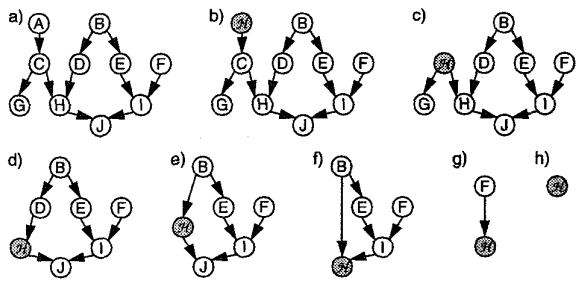

Figure 7: Sample example for reordering without recurrences.

1. Initially, the list of ordered nodes is empty ($List = \{\}$). We start by designating a node of the graph as the *Hypernode* ($\mathcal{H}$ in Figure 7). Assume that A is the first node of the graph. The resulting graph is shown in Figure 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 Figure 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 can 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 Figure 7d.

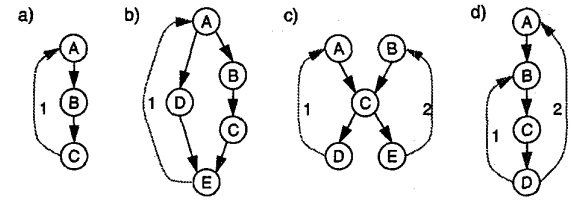

Figure 8: Types of recurrences

4. Now $\mathcal{H}$ has D as a predecessor, thus D is reduced, producing the graph in Figure 7e, and appended to the list ($List = \{A, C, G, H, D\}$).

5. Then, the successor J of $\mathcal{H}$ is ordered ($List = \{A, C, G, H, D, J\}$) and reduced, producing the graph in Figure 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 Figure 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 Figure 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 soon as possible.

3.2 Pre-ordering of graphs with recurrence circuits

In order not to degrade performance when there are recurrence circuits, the ordering step is performed giving 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 (Figure 8a).
- Recurrence circuits that share the same set of backward edges (Figure 8b). We call *recurrence subgraph* to the set of recurrence circuits that

```
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 := {h};
    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);
```

Figure 9: Procedure to order the nodes in recurrence circuits

share the same set of backward edges. In this way Figures 8a and 8b are recurrence subgraphs.

- Several recurrence circuits can share some of their nodes (Figures 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 Figure 8b are {A, D, E} and {A, B, C, E}. During the identification of recurrence circuits they are simplified so that recurrence circuits that belong to the same recurrence subgraph (i.e. that have the same set of backward edges) are stored as a single recurrence subgraph (in the worst case we can have a recurrence subgraph for each backward edge). For instance, the recurrence circuits associated with Figure 8b are stored as the recurrence subgraph {A, B, C, D, E}. After all recurrence subgraphs have been stored all the redundant nodes are removed so that a node only appears in the list associated with one recurrence subgraph . The nodes that appear in more than one recurrence subgraph are removed from all the sublists except for the most restrictive sublist in terms of *RecMII* (i.e. the first one in the list of recurrence subgraphs). For instance, the list of recurrence subgraphs associated with Figure 8c {{A, C, D}, {B, C, E}} will be simplified to the list {{A, C, D}, {B, E}}.

Once the nodes have been simplified, the actual ordering for recurrence circuits is performed. The algorithm that orders the recurrence circuits (see Figure 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 trivial recurrence circuits impose no scheduling constraints, as the scheduler previously ensured that $II \geq RecMII$. It

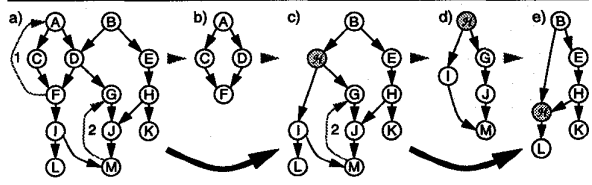

Figure 10: Example for *Ordering_Recurrences* procedure

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 Subsection 3.1. The whole subgraph is reduced to the *Hypernode*. Then, we look for all paths between the *Hypernode* and the next recurrence subgraph (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*.

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 Figure 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 that imposes most restrictions to *RecMII* . Therefore the algorithm starts by ordering it. If we isolate this subgraph and remove the backward edge we obtain the graph of Figure 10b. After ordering this graph the list of ordered nodes is ($List = \{A, C, D, F\}$). When the graph of Figure 10b is reduced to the *Hypernode* $\mathcal{H}$ in the original graph (Figure 10a), we obtain the dependence graph of Figure 10c.

2. The next step is to reduce the following recurrence subgraph {G, J, M}. For this purpose the algorithm searches 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 Figure 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 Figure 10c producing the graph of Figure 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 (Figure 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.

- 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 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:

$$Late\_Start_u = \min_{v \in PS} t_v - \lambda_u + \delta_{(u,v)} \times II$$

  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 $\min(Late\_Start_u, Early\_Start_u + II + 1)$.

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 have 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 Results

HRMS has been implemented in C++ using the LEDA libraries [17]. In this section we present some results of our experimental study. We compare HRMS with other scheduling methods using a small set of dependence graphs for which there are previously published results. In addition HRMS, has been exhaustively tested and evaluated for over one thousand loops from the Perfect Club Benchmark Suite [3]. For this loops, HRMS performance is compared with that of a Top-Down scheduler.

4.1 Comparison with other scheduling methods

We have evaluated how well our method performs compared with 3 leading methods. The selected methods are: an heuristic method that does not take into account register requirements [23] (FRLC), a life-time sensitive heuristic method [10] (Slack) and a linear programming method [8](SPILP).

We used 24 dependence graphs from [8] with a machine configuration with 1 FP Adder, 1 FP Multiplier, 1 FP Divider and 1 Load/Store unit. We have assumed a unit latency for add, subtract and store instructions, a latency of 2 for multiply and load, and a latency of 17 for divide.

Table 1 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 [8]. The number of buffers required by a schedule is defined in [8] 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 [18] it was demonstrated that the buffer requirements provide a very tight upper bound on the total register requirements.

Table 2 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 less 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% of the loops. For the remaining 66% of the loops the II is the same but in many cases HRMS requires less buffers, specially when compared to FRLC.

$S(u)$

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 1: Comparison of HRMS schedules with other scheduling 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 2: Comparison of HRMS performance versus the other 3 methods.

	HRMS	SPILP	Slack	FRLC
Compilation Time	1.45	290.72	0.93	0.71

Table 3: Comparison of HRMS compilation time to the other 3 methods.

Finally Table 3 compares the total compilation time in seconds for the four methods. Notice that HRMS is slightly slower than the two heuristic methods. But, these methods perform noticeably worse in finding a good scheduling. On the other hand, the linear programming method (SPILP) requires a much higher time to construct a scheduling that turns out to have similar 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.

4.2 Further evaluation of HRMS

In order to further evaluate HRMS, the dependence graph of all the innermost DO loops of the Perfect Club have been obtained with the ICTINEO compiler [2]. We have not measured loops with subroutine calls or with conditional exits. Loops with conditionals in their body have been converted to single basic block loops using IF-conversion [1]. A total of 1258 loops, which account for 78% of the total execution time of the Perfect Club[3], have been scheduled.

We assume a 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 2 load/store units, 2 adders, 2 multipliers and 2 Div/Sqrt units. All units are fully pipelined except the Div/Sqrt units which are not pipelined at all.

Scheduling all the loops consumed 5.5 minutes in a Sparc-10/40 workstation. Computing the recurrence circuits consumed only 3.2% of the scheduler execution time and the pre-ordering step consumed only 9% of the scheduler execution time. In contrast the scheduling step consumed 87.8% of the scheduler execution time. Notice the minimal impact of the ordering step (which is the key part of HRMS) on the overall scheduling time.

[3]Executed on an HP 9000/735 workstation

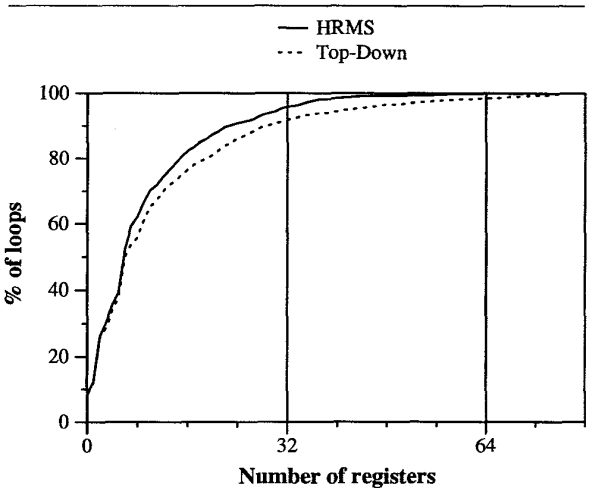

Figure 11: Static cumulative distribution of register requirements of loop variants.

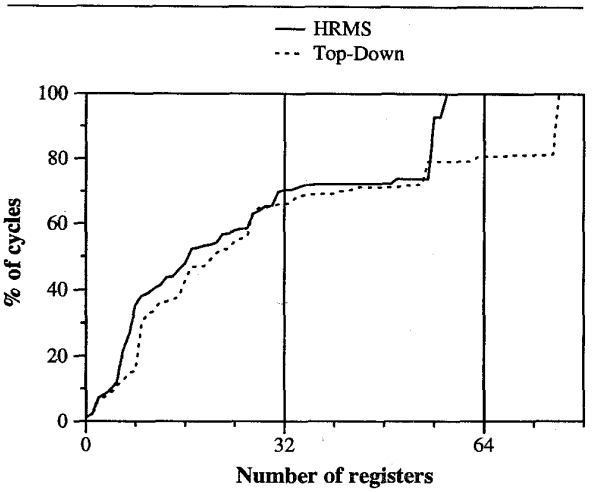

Figure 12: Dynamic cumulative distribution of register requirements of loop variants.

In order to evaluate performance, and to obtain dynamic results, 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 count iterations for the selected loops.

HRMS achieved optimal execution time ($II = MII$) for 1227 loops (97.5%). On average, the scheduler achieved an $II = 1.01 \times MII$. Considering dynamic execution time, the scheduled loops would execute at 98.4% of the maximum performance.

Once the loops have been scheduled, a lower bound on the register pressure of the loops (**MaxLive**) can be found by computing the maximum number of values that are alive at any cycle of the schedule. This section approximates the register requirements by this lower bound[4]. Lifetimes of loop variants start when the producer is issued and end when the last consumer is issued. Loop invariants are produced before entering the loop and are alive during all the execution of the loop, requiring one register each one during the execution of the loop.

Since an scheduler can only reduce the register requirements due to loop variants, Figure 11 compares the register requirements of them for both schedulers. On average HRMS requires 87% of the registers required by the Top-Down scheduler. Even though there are few loops requiring a high number of registers, loops with a high number of registers represent an important amount of the execution time of the Perfect Club (see [15]). Figure 12 shows the dynamic register requirements (i.e. each loop has been weighted by its execution time) of loop variants for both schedulers.

Most machines store loop variants and loop invariants are stored in the same register file, so their combined register pressure is also of interest. Figure 13 shows the dynamic register requirements of loop variants plus loop invariants. Notice that about 20% (it varies depending on the scheduler) of the cycles is spent in loops requiring more than 64 registers and 45% of the cycles is spent in loops requiring more than 32 registers.

Given that actual machines have a limited number of registers (generally 32), it is also of interest to evaluate the effect on performance of loop variants plus loop invariants when there is a fixed amount of available registers. Figure 14 shows the execution time of the loops scheduled with both schedulers when there are infinite, 64 and 32 registers available. When a loop requires more than the available number of registers, spill code has been added [15] and the loop has been rescheduled. Notice that the code generated by HRMS is about 43% faster in a machine with 64 registers and about 21% faster in a machine with 32 registers for the assumed architecture. We can also observe that the code generated by HRMS for a machine with 32 registers runs almost as fast as the code generated by the Top-Down scheduler for a machine with 64 registers.

5 Conclusions

This paper has presented *Hypernode Reduction Modulo Scheduling* (HRMS), a novel and effective technique for resource-constrained software pipelining. HRMS can deal with loops containing loop-carried dependences and attempts to optimize the initiation in-

[4] For an extensive discussion of the problem of allocating registers for software-pipelined loops refer to [21]. The allocation strategies presented in this paper almost always achieve the **MaxLive** lower bound. In particular, the wands-only strategy using end-fit with adjacency ordering never requires more than **MaxLive** + 1 registers.

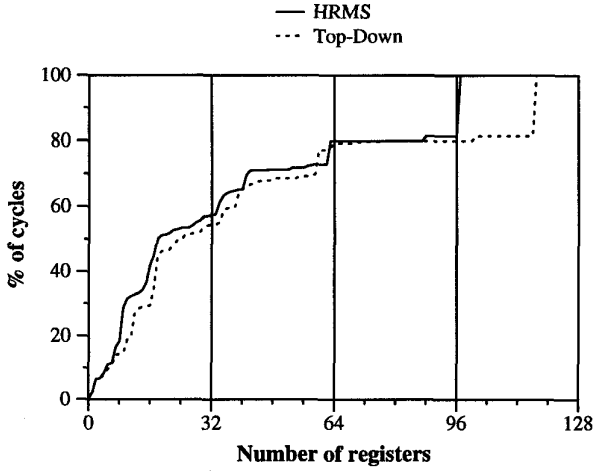

Figure 13: Dynamic cumulative distribution of register requirements of loop variants plus loop invariants.

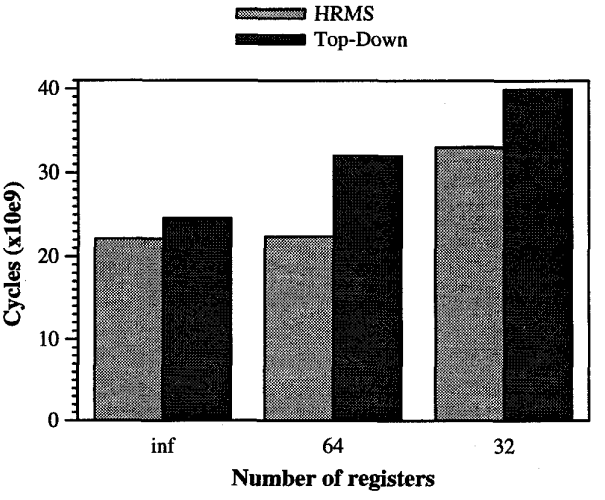

Figure 14: Cycles required to execute the loops with infinite registers, 64 registers and 32 registers.

terval while reducing the register requirements of the schedule.

HRMS pre-orders the nodes of the dependence graph before scheduling them. The ordering function gives priority to recurrence circuits, in order not to penalize the initiation interval. In addition, nodes are ordered in such a way, that when a node is scheduled, the scheduling contains at least a reference node (a predecessor or a successor). The ordering step guarantees that (except in the special case of recurrence circuits) only predecessors or successors of the current node are already scheduled, but not both of them.

Nodes are scheduled after being ordered. The scheduling step schedules a node as soon as possible if it has predecessors already scheduled, and schedules a node as late as possible if it has successors already scheduled. Scheduling nodes in this way shortens lifetimes of loop variants, and therefore reduces the register requirements of the schedule.

The usefulness of HRMS has been empirically established by applying it to several loops taken from common scientific benchmarks. We have compared our schedules with three leading methods, namely Govindarajan et al. SPILP integer programming formulation, Huff's Slack Scheduling and Wang et al. 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 as SPILP, which required up to two orders of magnitude more computing time to obtain the schedules.

Finally we provided an exhaustive evaluation of HRMS using 1258 loops from the Perfect Club Benchmark Suite. HRMS generates schedules that are optimal in terms of II for 97.4% of the loops. The preordering step has a minimal impact on the scheduling time. HRMS has been also compared with a Top-Down scheduler that does not care about the register requirements. It has been shown that, when there is a limited number of registers, HRMS has a big performance advantage.

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 [8] in order to compare our proposal with other methods. Thanks are also due to Enric Riera and the rest of the ICTINEO team for their support to this work. Finally we would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] J.R. Allen, K. Kennedy, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th annual Symposium on Principles of Programming Languages*, January 1983.

[2] E. Ayguadé, C. Barrado, J. Labarta, D. López, S. Moreno, D. Padua, and M. Valero. A uniform representation for high-level and instruction-level transformations. Technical Report UPC-DAC-95-02, Universitat Politècnica de Catalunya, January 1995.

[3] 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, November 1988.

[4] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *MICRO25*, pages 292–300, 1992.

[5] J.C. Dehnert, P.Y.T. Hsu, and J.P. Bratt. Overlapped loop support in the cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, 1989.

[6] J.C. Dehnert and R.A. Towle. Compiling for cydra 5. *Journal of Supercomputing*, 7(1/2):181–227, 1993.

[7] A.E. Eichenberger, E.S. Davidson, and S.G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proc., Internat. Conf. On Supercomputing*, pages 31–40, July 1995.

[8] R. Govindarajan, E.R. Altman, and G.R. Gao. Minimal register requirements under resource-constrained software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, November 1994.

[9] P.Y.T. Hsu. Design of the R8000 microprocessor. Technical report, MIPS Technologies, Inc, June 1994.

[10] R.A. Huff. Lifetime-sensitive modulo scheduling. In *6th Conference on Programming Language, Design and Implementation*, pages 258–267, 1993.

[11] S. Jain. Circular scheduling: A new technique to perform software pipelining. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 219–228, June 1991.

[12] M.S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

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

[14] J. Llosa, M. Valero, and E. Ayguadé. Non-consistent dual register files to reduce register pressure. In *1st Symposium on High Performance Computer Architecture*, pages 22–31, January 1995.

[15] J. Llosa, M. Valero, E. Ayguadé, and J. Labarta. Register requirements of pipelined loops and their effect on performance. In *2nd Int. Workshop on Massive Parallelism: Hardware Software and Applications*, October 1994.

[16] W. Mangione-Smith, S.G. Abraham, and E.S. Davidson. Register requirements of pipelined processors. In *Int. Conference on Supercomputing*, pages 260–246, July 1992.

[17] 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.

[18] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Conf. Rec. of the Twentieth Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 29–42, January 1993.

[19] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[20] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Microprogramming Workshop*, pages 183–197, October 1981.

[21] B.R. Rau, M. Lee, P. Tirumalai, and P. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[22] J.A. Swensen and Y.N. Patt. Hierarchical registers for scientific computers. In *Int. Conference on Supercomputing*, 1988.

[23] J. Wang and C. Eisenbeis. Decomposed software pipelining: A new new approach to exploit instruction level parallelism for loops programs. In *IFIP*, January 1993.

[24] S.W. White and S. Dhawan. POWER2: Next generation of the RISC System/6000 family. In *IBM RISC System/6000 Technology: Volume II*. IBM Corporation, 1993.