UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

2

SOFTWARE PIPELINING

2.1 INTRODUCTION

Software pipelining [Cha81] comprises a family of techniques aimed at finding a pipelined schedule of the execution of loop iterations. The pipelined schedule represents a new loop body which may contain instructions belonging to different iterations. The sequential execution of the schedule takes less time than the sequential execution of the iterations of the loop as they were initially written.

In general, a pipelined loop schedule has the following characteristics[1]:

- All the iterations (of the new loop) are executed in the same fashion.
- The initiation interval (II) between the issuing of two consecutive iterations is always the same.

Figure 2.1 shows an example of software pipelining a loop. The DDG representing the loop body to pipeline is presented in Figure 2.1(a). The loop must be executed ten times (with an iteration index $i \in [0, 9]$). Let us assume that all instructions in the loop are executed in a single cycle and a new iteration may start every cycle (otherwise the dependence between instruction A from

[1] We will assume here that the schedule contains a unique iteration of the loop.

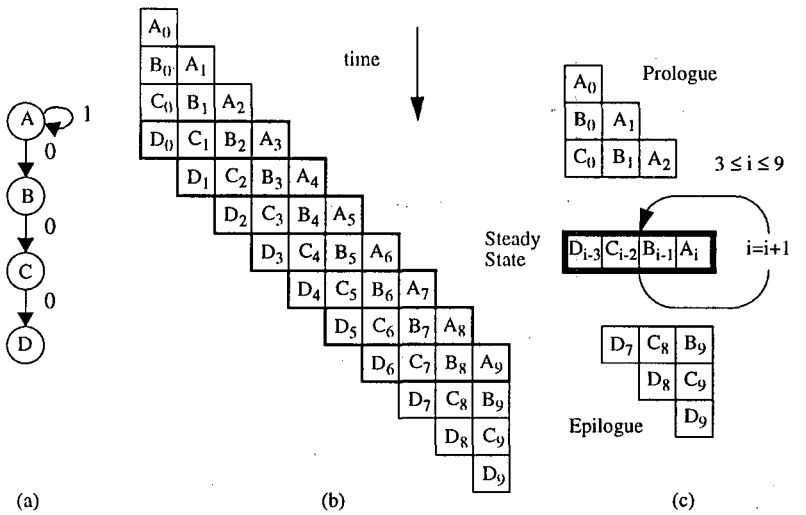

Figure 2.1 Software pipelining a loop
(a) DDG representing a loop body
(b) Parallel execution of the loop
(c) New parallel loop body

iterations i and $i+1$ will not be honored). With this assumption, the loop can be executed in a more parallel fashion than the sequential one, as shown in Figure 2.1(b). The execution time of the sequential loop is 40 cycles, whereas the execution time of the pipelined loop is only 13 cycles. The execution of an instruction X belonging to the iteration i is denoted by X_i in Figure 2.1. Note that an execution pattern is repeated from the issuing of iteration third until the issuing of iteration ninth ($i \in [3, 9]$). Such a pattern is called the *steady state* or *pipeline*. Instructions A_i, B_{i-1}, C_{i-2} and D_{i-3} from iterations $i = 3$ to $i = 9$ are executed at a time in the *steady state* (we assume that the architecture has sufficient resources to execute the four instructions in parallel). In order to maintain the semantics of the loop execution, a piece of code must be executed before the *steady state*. This code is denoted as the *prologue*. In the same way, a piece of code, denoted as *epilogue*, must be executed after the last iteration of the *steady state*. By using the previous description, a new more parallel loop body (formed by the *steady state*) can be built, as shown in Figure 2.1(c). The new loop is preceded by a *prologue* and followed by an *epilogue*, and it is executed seven times instead of ten.

2.2 STATE OF THE ART

2.2.1 Notation and classification

This section describes some software pipelining approaches. Software pipelining has been widely studied in recent years. It is suitable for the high-level synthesis (HLS) of VLSI circuits, as well as for compilers for parallel architectures (superscalar and VLIW processors). Several software pipelining techniques have been proposed for these two areas in recent years. The terminology

used in the papers describing such techniques is quite wide, and usually different terms are used to describe the same concept. In order to use the same terminology for all methods, we will firstly perform some brief definitions which will be used in the rest of this chapter to describe the different techniques.

Let us call *initial loop* the loop as it was initially written, and *pipelined loop* the new loop obtained after being software pipelined. The initiation interval (II) denotes the number of cycles elapsed between the issuing of two consecutive iterations of the pipelined loop (and therefore the time required to execute an iteration). The II is usually shorter than the time required to execute an iteration of the initial loop. When a lower bound on the initiation interval is computed, it is called the *minimum initiation interval*, and denoted by *MII*. The time required to execute an iteration of the initial loop in the pipelined loop is called the *iteration time*. The number of iterations skewed for each instruction in the steady state is called the *iteration index* or *folding index*. The *iteration indices* are always positive integers normalized to zero. In the example from Figure 2.1(c), the *initiation interval* is 1, the *iteration time* is 4 and the *iteration indices* for instructions A, B, C and D are respectively $3, 2, 1$ and 0, indicating that instructions $A_{j+3}, B_{j+2}, C_{j+1}$ and D_j are executed in parallel in the steady state.

In general, software pipelining approaches represent a loop by means of a *data dependence graph*, which is a directed graph $DG(V, E)$. Instructions of the loop are represented by the set of nodes V, whilst the set of edges E represents the *data dependences* or *precedence constraints* between instructions (in general, compilers for parallel architectures consider the nodes of the DG as instructions, while systems for HLS consider nodes as single operations. The main difference among them is that an instruction may contain multiple operations and use several resources). When a subset of edges forms a cycle in the DG, it is called *a recurrence*. Two kind of data dependences are in general considered: *intra loop dependences* (ILDs) between instructions belonging to the same iteration, and *loop carried dependences* (LCDs) between instructions belonging to different iterations. A sequence of edges (with their corresponding nodes) in the DG is called a *path*. The node which has no predecessors in the *path* is called the *head* of the *path*. In the same way, the node which has no successors is called the *tail* of the path. A *path* in the DG requires a certain execution time in order to honor all its dependences. Such execution time is called the *length* of the *path*. The *path* with the largest *length* is called the *critical path* of the graph.

We will classify software pipelining approaches into three main categories:

1. Techniques which do not calculate a lower bound on the II (*MII*). These techniques usually try to find a pipelined schedule quickly. The II of the schedule is then successively decreased until it cannot be further reduced. These techniques cannot guarantee that an optimal schedule is found, even when they find it, because they do not calculate *MII*.

2. Techniques which make an estimation of the *MII*. In general, these techniques make an estimation of the *MII* precompacting the loop by assuming infinite resources. Next, they try to find a pipelined schedule in the previously compacted number of cycles by considering resource constraints. As in the techniques of category 1, these kind of techniques cannot guarantee that an optimal schedule is found, since *MII* is estimated.

3. Techniques which analytically compute the *MII*. We classify these techniques into three categories, according to which factors they take into account for calculating the *MII*:

   (a) Techniques which take only resources into account for calculating *MII*.

(b) Techniques which take only recurrences into account for calculating *MII*.

(c) Techniques which take both the resources and the recurrences into account for calculating *MII*.

These techniques try to find a pipelined schedule in the previously computed *MII* cycles. If no schedule is found, the expected initiation interval is increased and the methodology is applied again. The main advantage of these techniques is that they can guarantee that an optimal schedule is found when the *II* of the schedule is the previously calculated *MII*.

2.2.2 Approaches which do not calculate MII

Perfect pipelining [AN88b]

In perfect pipelining (*PP*) [AN88b], the loop is compacted, unrolled and pipelined, searching for an emerging pattern. The instructions may be moved independently after precompaction.

A set of local primitive transformations, involving only adjacent nodes of the *DG*, are the building blocks of *PP*. They are called the *core transformations*, and were defined for *percolation scheduling* [Nic85]. The four *core transformations* are the following:

- *DELETE*: removes a node from the *DG* if the node is empty (contains no instructions) or unreachable. A node may become empty or unreachable as a result of other transformations.
- *MOVE-OP*: moves an assignment x from a node u to a node v through an edge (u, v), provided no conflict exists between x and the instructions in v, and x does not kill any value alive in v.
- *UNIFY*: moves a single copy x of identical assignments from a set of nodes $\{v_j\}$ to a common predecessor node u. This is performed if no dependence exists between x and the instructions in u, and x does not kill any value alive at u.
- *MOVE-TEST*: moves a test x from a node v to a node u through an edge (u, v), provided that no dependence exists between x and the instructions in u.

Once the loop has been compacted, it is unrolled an unspecified number of times. The result is further compacted by using the *core transformations* and assuming unlimited resources. In order to make the formation of a pattern easier, conditions are added to limit the distance between the first and the last scheduled instructions from an iteration. When the pattern produced after unrolling K times the loop is equivalent to the pattern produced by unrolling the loop $K - 1$ times, the algorithm halts.

Although code space is increased, an advantage of *PP* face to other approaches is that the code may be compacted more tightly. Another advantage is that the target machine is assumed to have multi-way branching. This is a powerful feature which compares directly to other approaches whose basic blocks are ended by a single branch, and where only basic blocks are considered. A disadvantage of *PP* is that unlimited resources are initially assumed. The way to solve resource

constraints is by rescheduling the pattern found for unlimited resources with the given constraints. This produce sub-optimal results in several cases.

Software retiming [PR91]

[PR91] and [PR94] present a methodology for software pipelining based on four transformations: *retiming*, *associativity*, *commutativity* and *inverse element law*. The approach is denoted as *software retiming* (SR). The proposed transformations work as follows:

- *Retiming*: As defined by Leiserson in [LRS83], retiming uses the distributivity of the delay operator D over most other operators: $D(a)\Theta D(b)$ is equivalent to $D(a\Theta b)$ and vice versa (Θ being an arbitrary operator).
- *Associativity*: Associativity postulates that, in the set over an algebraic structure defined using an operation Θ, for every a, b and c which are elements of the set, it holds that $a\Theta(b\Theta c) = (a\Theta b)\Theta c$ [Van50].
- *Commutativity*: Commutativity states that, in the set over an algebraic structure defined using an operation Θ, for every a, b which are elements of the set, it holds that $a\Theta b = b\Theta a$ [Van50].
- *Inverse element law*: If a is in the the set over an algebraic structure defined using an operation Θ, then there is some element b in the set, called an inverse of a, such that $a\Theta b = b\Theta a = e$.

Software retiming is a fast iterative improvement probabilistic algorithm that uses the previous transformations. SR focuses on solving the following problem: "given a DG of a loop and a set of resource constraints, apply retiming, associativity, commutativity and inverse element law in such a way that the resource utilization of the obtained DG is maximized". Note that *software retiming* maximizes the resource utilization achievable by a DG, and not the resource utilization of a schedule. Therefore, *software retiming* makes an estimation of the resource utilization, instead of an exact measurement. The resource utilization for a resource r, U_r, is defined as the ratio of the number of cycles in which the resource is used to the available number of cycles. The total resource utilization of a loop, U, is defined as the weighted sum of the resource utilization of the set of resources (R): $U = \sum_{r \in R} w_r \cdot U_r$. The weights w_r are proportional to the hardware cost of the resource r.

In order to lead the transformations, *software retiming* uses an objective function highly correlated to the final (unknown) hardware utilization. The function is based in the following observations:

- Timing constraints on nodes that are not strict make it easier to achieve a high resource utilization.
- Nodes vying for the same resource must be distributed over the time.
- The critical path must be shorter than the available time.
- The number of variables which are alive at the same time must be smaller than the number of available registers.

Software retiming uses an iterative algorithm to find a solution. Two classes of movements are defined: *retiming* and *generalized associativity movements*. For each feasible movement α, an inverse transformation β exists. Movement α is denoted as a *forward movement*, and movement β is denoted as a *reverse movement*. The iterative algorithm is organized in two phases:

1. In the first phase, the solution space is scanned in an organized fashion in order to detect areas where the objective function has a small value. At every point in the optimization process, a movement is selected in a probabilistic fashion.
2. The areas selected in the first phase are used in the second phase as the starting points for a more elaborate search. The movement offering the best decreasing in the objective function is automatically performed. For each starting point, the search is concluded when a local minimum is reached. The best of those minima is selected as the final solution.

Since the objective function is parameterized, users may guide the search towards their own objectives. However, a high CPU-time is required to execute the iterative algorithm, especially for large *DG*s. One problem is that, since the points found in the first phase do not guarantee a near proximity to the best solution, phase 2 can derive to local minima, moving away from the best solution.

Rotation scheduling [CL92]

An approach for software pipelining based on retiming [LRS83], called *rotation scheduling*, is presented in [CL92] and [CLS93]. *Rotation scheduling* reduces the iteration time of the pipeline schedule after finding the schedule.

The *rotation* technique repeatedly transforms a schedule into a more compacted one, increasing the resource utilization. An existent schedule is partially rescheduled by *rotation* to obtain a shorter and valid schedule with resource constraints. The result of *rotation* implicitally retimes the $DG(V, E)$ to naturally produce a pipeline schedule. The state of a sequence of *rotations* is recorded by a simple retiming (node-labelling) function. Therefore, *rotation* and *folding* are similar concepts.

After a sequence of *rotations*, the iteration time may be too long. In this case, it must be reduced. *Rotation scheduling* (RS) uses a simpler *integer linear programming formulation* [Sch86] to find a retiming with the smallest iteration time such that the given schedule is a valid schedule.

The algorithm to find a loop schedule by *rotation* with a short iteration time for a given schedule is as follows:

1. *Find a down-rotatable set of nodes*: A set X of nodes is *down-rotatable* if and only if every path from $V - X$ to X contains at least one LCD.
2. *Rotate the found set down*: A *down-rotation* of a node implies retiming the node (in terms of folding, it is equivalent to increasing the folding index of the node). In order to select a node to be *down-rotated*, *rotation scheduling* uses a *list scheduling* algorithm [DLSM81] which uses the number of successors of a node as priority function.

3. *Reschedule the set of nodes*: Each node is initially rescheduled at the last possible cycle of the schedule while resources are available. After that, the node is pushed across the schedule according to the data dependences in the *DG*. After any *down-rotation*, there always exists a schedule which is at least as short as the original one.

An advantage of *rotation scheduling* is that it is intuitively easy to understand. However, since the *MII* is not computed, *rotation scheduling* cannot guarantee that an optimal solution has been found (except when all the resources are fully used). The final step, reducing the iteration time, may correct excessive retiming done by the rotation algorithm (since heuristics have been used to select a node to be down-rotated). However, since such a step is performed by using an *integer linear programming formulation*, it may consume much CPU-time.

2.2.3 Approaches which estimate the MII

URPR [SDX86]

The *URPR* algorithm (UnRolling, Pipelining and Rerolling) [SDX86] first compacts the loop body without taking LCDs into account, and following this the precompacted loop is pipelined. The compaction is performed to find an upper bound on the steady state length. The new instructions (composed of several of the original instructions) found after the initial compaction are called *microcode composites* (*MC*s). The initiation interval expected for the pipelined loop is computed as follows:

- The number of cycles that each MC_i from different iterations must be offset to honor all ILDs, $\delta(MC_i)$, is computed.
- The expected initiation interval is the maximum of all the $\delta(MC_i)$.

Once the initiation interval (*II*) has been computed, *URPR* unrolls the loop $K = \frac{L}{II}$ times, L being the length of the precompacted loop body. K is the maximum number of iterations that *URPR* can execute concurrently under ideal conditions. The unrolled loop is scheduled as follows:

- Each new compacted loop body is scheduled from *II* cycles after the starting of the last loop body.
- When *MC*s from different compacted loop bodies are placed in the same cycle, there must be no resource conflicts. Moreover, the pipeline must honor all LCDs.
- If placing an *MC* violates an LCD, it is moved down in the schedule until all dependences are honored.

After the loop has been unrolled and pipelined, it is rerolled. Rerolling is the process of creating a prologue, a steady state and an epilogue. The steady state is created from a set of adjacent cycles of minimum length which contains all the *MC*s in the original loop body. Since the steady

state may contain more than one copy of a given MC, all duplicated MCs are deleted. This is performed to ensure an integer number of original loop iterations within the steady state. For greater simplicity, the algorithm further restricts the integer number of loop iterations to one.

In order to calculate the initiation interval, the *URPR* algorithm only takes dependences which cross one iteration into account. However, this process should be performed for all dependences in the loop. The *URPR* algorithm has the advantage of being simple. Working with a precompacted loop makes it easy to pipeline without taking ILDs into account. However, this can also be considered as a disadvantage, since the precompaction may impose unnecessary constraints on the pipelining process, often restricting the achievable parallelism.

Sehwa [PP88]

Sehwa [PP88] is a system for the synthesis of pipelined data paths. Shewa can find the minimum cost design, the highest performance design and other designs between these two in the design space.

Sehwa contains a set of fast scheduling procedures which are iteratively executed. In order to maximize the performance of the design, these procedures perform resource allocation at the same time as scheduling. Each scheduling iteration is guided by the performance and cost estimation of the previous schedule. Sehwa optionally uses an exhaustive algorithm for optimal scheduling of operations while satisfying the cost constraints. Such exhaustive scheduling algorithm takes the best result of polynomial time-scheduling algorithms as a good lower bound on performance to prune the search space. The algorithm to perform cost-constrained synthesis is as follows:

1. Compute the minimum possible initiation interval such that the cost for the minimum required set of resources is within the cost constraint. The number of resources of each type i required for an expected initiation interval II, N_i, is computed as $N_i = \lceil \frac{V_i}{II} \rceil$, V_i being the number of nodes representing operations which require a resource of type i.
2. For all possible cycles, performing scheduling of the loop operations while resource constraints are satisfied.
3. For all schedules find at step 2:
   - compute the total cost of the design by adding the latch and multiplexer cost to the operator cost.
   - compute the performance of the design considering the resynchronization rate.
4. If any solutions are found satisfying the cost constraints:
   - select the fastest design.
   - check if the performance of the fastest design can be increased by adding resources while the cost constraint is satisfied. Select a type of resource to be increased and execute step 2 again. Otherwise, execute step 6.
5. If no solutions are found satisfying resource constraints, choose another set of resources. Increment II in one unit and recompute the minimum number of required resources. Then, go to step 2.

6. Report the fastest design found.

Sehwa works with conditional jumps. Software pipelining is achieved by considering *functional pipelining* in the data path. That is, a new datum may be issued before the previous datum gives a result. Although *functional pipelining* was initially proposed to work with an algorithm represented by a basic block, it works quite well with loops. In fact, *functional pipelining* n the data-path is specially useful to pipeline loops represented by acyclic *DG*s.

Since Sehwa is an algorithm based on successive refinements of the found solutions, its computational complexity is higher than the computational complexity of other approaches. Sehwa usually finds good designs because the search space is widely explored.

Loop folding [GVD89]

Loop folding (LF) [GVD89] is a control-flow transformation that is equivalent to *functional pipelining* in the data-path. *Loop folding* has been implemented in the CATHEDRAL II compiler integrated in the ATOMICS system [GRVD87].

The goal of *loop folding* is to optimize loop organization by introducing partial overlaps between the schedule of consecutive loop iterations. Like the remaining software pipelining techniques, *loop folding* reduce the average time required to execute an iteration by parallelizing the execution of operations belonging to different iterations. As a side effect, the iteration time may increase. In terms of data-path pipelining, the number of cycles elapsed between the supply of an input data and the associated output may be increased.

Loop folding can be realized by moving (or *folding*) operations between loop iterations before scheduling the loop. The *folding index* of an operation in a certain loop organization is the (integer) number of loop iterations over which the operation has been moved with respect to the original loop organization. In the original loop organization, all operations have a *folding index* of zero. If the *folding index* of an operation u is increased (decreased) in a *DG*, the *folding indices* of all the successors (predecessors) of u should be increased (decreased) with at least the same amount.

ATOMICS optimizes the loop control-flow iteratively. In successive *folding-iteration* steps, some operations are *folded*. The obtained *DG* is scheduled by using *list scheduling* [DLSM81] and taking resource constraints into account. In order to schedule the *DG*, ATOMICS uses a two-level iteration mechanism.

- First of all, an estimation of the II is calculated, and the *DG* is scheduled in II cycles. II is assumed to be the length of the critical path plus one. If no schedule in II cycles is found, the estimated II is increased and the process is repeated again until a schedule is found.
- The *folding* may be adjusted in a next iteration step, in order to reduce the obtained II. The goal of *folding-iteration* step j is simply to find a schedule with a length II_j equal to $II_{j-1} - 1$.

The heuristics used to fold a loop are described below. As a basic principle, *folding indices* of operations will never be decreased. In every *folding-iteration* step j, the schedule obtained from

step $j-1$ is examined. A group of nodes, for which an additional *folding* will most probably result in a faster schedule, is identified:

- The *folding index* of all nodes that have been scheduled prior to cycle II_j is increased by one.
- In order for a schedule of length II_j to exist, the *DG* should not contain any path longer than $II_j + 1$. If such path(s) occurs, the *folding index* of all nodes belonging to the excessive parts of such path(s) is increased by one. This action is repeated until no such path(s) remains.

2.2.4 Approaches which analytically calculate MII

Modulo scheduling [RG81]

Modulo Scheduling (*MS*) [RG81] is probably the best-known software pipelining approach. In fact, *MS* is a framework within which a wide variety of algorithms and heuristics may be defined for scheduling a loop. The objective of *MS* is to engineer a schedule for one iteration of the loop such that, when this same schedule is repeated at regular intervals of *II* cycles, no ILD nor LCD is violated, and no resource usage conflict arises between instructions of either the same or different iterations.

MS consists of the following steps:

1. Compute a lower bound on the initiation interval (*MII*). The number of available resources and the recurrences (cycles formed by data dependences in the DG) on the loop are taken into account to compute *MII*.
2. If *MII* is not an integer, and if the percentage degradation in rounding it up to the next larger integer is unacceptably high, the body of the loop may be unrolled prior to scheduling.
3. Schedule a single iteration of the loop in any desired manner while resource and dependence constraints are being fulfilled. The different heuristics to determine the order of scheduling instructions configurate the different *MS* algorithms. The instructions are scheduled *as soon as possible*.
4. If no schedule has been found by step 3, increase the expected *II* by one unit and execute again step 3. Some approaches have proposed increasing the initiation interval by more than one unit in order to avoid spending an excessive amount of time compiling large complex loops [Huf93].
5. If rotating registers [RYYT89, BYA93] are absent, the steady state is unrolled to enable *modulo variable expansion* [Lam88].
6. Generate the appropriate prologue and epilogue code sequences.

The schedule found by following the previous steps allows us to issue a new iteration each *II* cycles. Initially, the instructions were stored in a reservation table with a length of *IT* cycles, *IT* being the iteration time. The space required to store such a schedule is proportional to the iteration

time of the loop. In order to reduce such a space, and given that the expected II of the loop is known in advance, current approaches of MS use a *modulo reservation table* (MRT) [Lam88] to store the schedule. The MRT is a reservation table with a length of II cycles. Therefore, the space required to store the schedule is proportional to II (II is, in general, less than the iteration time). Each row of the MRT represents a cycle for issuing an instruction, and each column represents a resource. Operations are assigned to the MRT by fulfilling the *modulo constraint*:

- An instruction u which is scheduled at cycle C_u is assigned to cycle $C_u \bmod II$ in the MRT, with an iteration index $\left\lfloor \frac{C_u}{II} \right\rfloor$.
- No other instruction v can be scheduled at cycle C_v such that $(C_u \bmod II) = (C_v \bmod II)$ if there are not sufficient resources to execute u and v at the same time.

The algorithm stops when all instructions have been assigned to the MRT or an instruction exists which cannot be assigned without violating any dependence nor modulo constraint.

In MS approaches, if an instruction is scheduled in an inappropriate cycle, scheduling the remainder instructions of the loop may be impossible, even though a feasible schedule may exist. Since no heuristic guarantees a correct selection of instructions, the only way to solve this drawback is to introduce backtracking when an instruction cannot be scheduled in the MRT. Several approaches have been recently proposed by using this idea [Huf93, Rau94].

MS approaches are easy to understand and to implement. Moreover, their low computational complexity [Rau94] makes them appropriate for compilers for parallel architectures. In fact, to our knowledge, at least two current compilers [Ram92, DT93] have already incorporated *modulo scheduling* algorithms.

Optimal loop parallelization [AN88a]

The approach developed in [AN88a] is based on some results in compaction-based software pipelining from perfect pipelining [AN88b]. We will call this approach OLP. OLP is based on the following idea: "scheduling a large portion of the loop's execution history should reveal some repeating behavior, which can be used to obtain a good schedule for the loop". Such a repeating behavior is denoted as the *pattern* of the loop.

In order to find the *pattern*, OLP examines a partial execution history of the loop, denoted as the first i iterations. After that, the statements of those i iterations are scheduled *as soon as possible*. That is, if the longest chain of dependences on which statement u depends has length j, then u is scheduled at cycle j. This is called a *greedy* schedule.

First of all, OLP calculates a lower bound on the initiation interval (MII) of the loop by taking the recurrences of the loop into account. After that, OLP searches for a pattern with length MII. To do this, statements not belonging to the critical path are rescheduled so that they have the same initiation interval as statements on the critical path. The critical path can be scheduled exactly in MII cycles. The result is a very compact pattern for the entire loop.

OLP divides the schedule of an iteration in *regions*. A *region* in the schedule of an iteration is a continuous interval of cycles in which each one contains some statement from the iteration. *OLP* searches for the *maximal* (longest) *regions* of an iteration. Between two adjacent *maximal regions* there must be at least one cycle (gap) with no statement from that iteration. *OLP* looks for consecutive iterations i and $i+1$ with the same *maximal regions* and where the gap between the *maximal regions* from $i+1$ is larger than or equal to the gap between the *maximal regions* from i. Reducing the gap of both iterations by delaying the *maximal region* which is first scheduled does not increase the execution time of the loop, but it makes it easier to find an execution pattern.

As in *perfect pipelining*, when the number of resources is constrained, *OLP* reschedules the pattern found for infinite resources with the given constraints. Therefore, sub-optimal results are found (note also that resource constraints are not taken into account to compute *MII*). *OLP* is an approach intuitively easy to understand and to program with a low computational complexity, as shown in [AN88a].

Percolation-based synthesis [PLNG90]

The Percolation Based Synthesis (*PBS*) [PLNG90] is a software pipelining approach which works with conditional jumps and multi-cycle pipelined instructions. *PBS* is based on percolation scheduling [Nic85] and loop unrolling.

Percolation scheduling is a system of semantic-preserving transformations which converts an original *DG* into a more parallel one. Nodes in the new *DG* may contain several instructions which may be executed in parallel. They are also called *states*. The core of *PBS* consists in the same four transformations as *perfect pipelining*; namely, *move-op, move-test, delete* and *unify*. Repeatedly applying this transformations allows data-independent instructions to *percolate* towards the top of the *DG* from the different parts of the code. The core transformations are proven to be complete with respect to the set of all possible local, dependence-preserving transformations.

PBS performs the following five steps:

1. *Find the optimal schedule*: the first step consists of finding an optimal schedule without taking resource constraints into account. Such a schedule is found by using the *OLP* approach.
2. *Find each instruction's mobility and reorder instructions*: if the number of instructions in one state exceeds the number of available resources, some instructions must be delayed. *PBS* uses the *mobility* [PG87] as criterion to defer instructions. Operations with the highest *mobility* are first deferred because their delay will not necessarily stretch the schedule, whereas deferring an instruction with mobility 0 will certainly make the schedule longer.
3. *Make reservations*: when the functional units are not pipelined, a new instruction must wait for another one to be completed if both execute in the same functional unit. This fact causes the reservation problem. In the presence of resource constraints, assigning the optimal schedule to the available resources is not always possible. Therefore, *PBS* reserves states so that the execution time of the functional units is respected for any instruction. Empty states are added when necessary.

4. *Select and adjust state i*: this step selects a state i and delays instructions from i due to resource constraints. An instruction which has to be deferred is moved to the next available state in the program.

5. *Percolate instructions from i's successors*: after deferring some instruction from state i, there is a possibility that some of the instructions from i's successors will percolate up. This percolation of instructions is due to the addition of new states between the original i and its successors. Operations are moved up while preserving data dependences, and only if there are available resources in earlier states. Steps 4 and 5 are repeated while there are resource conflicts.

The advantage of PBS is that conditional jumps and multi-cycle pipelined instructions are taken into account. The disadvantage is that the loop is unrolled only until a pattern is found, and only one iteration of the loop is considered. This reduces the possibility of percolating instructions. Moreover, since the initial optimal schedule only takes recurrences (and not resource constraints) into account, the schedule found may be sub-optimal.

Unrolling-based software pipelining [BC90]

In [BC90], a loop optimization approach for horizontal microcode machines, is proposed. This approach is denoted as *unrolling based software pipelining*, and it will be shortened here by $UBSP$.

First of all, $UBSP$ calculates a lower and an upper bound on the initiation interval. The lower bound is computed by taking only resources into account. The upper bound is the length of a non-pipelined schedule. The expected initiation interval of the schedule may range between the previous two bounds.

The basic idea of $UBSP$ is to search for a repetitive pattern in the scheduling of the previously unrolled (and compacted) loop. Such a pattern may contain more than one iteration. Since unrolling is considered, $UBSP$ enables several iterations to have a different scheduling, unlike techniques which do not take loop unrolling into account. The algorithm used for $UBSP$ is as follows:

1. Initiate the unrolling degree[2] i ($i = 1$).
2. Unroll the loop $i + 1$ times and schedule (by compaction) the unrolled loop.
3. Search for a pattern in the schedule. This pattern may contain several iterations of the loop.
4. If no pattern is found, increment i by one unit and go to 2. If a pattern is found, construct a new equivalent loop containing the following: a prologue, the found pattern repeated a certain number of times, and an epilogue.

The main problem of the algorithm is to ensure that the compaction algorithm used at step 2 is really bounded. In order to guarantee this, the algorithm has a mechanism which restrains the

[2] The unrolling degree is the number of instances of the loop body. Therefore, an unrolling degree two means that two loop bodies are considered.

length of the scheduling of the iterations. It prohibits instructions of an iteration to be scheduled too early in the unrolled loop.

As can be seen, the algorithm is quite similar to the proposed by *OLP*. The main difference is that resource constraints are taken into account from the beginning, whilst *OLP* considers resource constraints at the end of the process. However, the calculation of the lower bound on *II* does not take recurrences into account. Therefore, the algorithm can start searching for impossible schedules, which increases its execution time. One advantage of *UBSP* compared with some previous described techniques is that a pattern containing several iterations of the loop is considered. This may significantly increase the throughput of the schedule, as will be shown in following chapters.

Lam's algorithm [Lam88]

The Lam's algorithm (*LA*) [Lam88] is one of the most well-known software pipelining approaches. In fact, the term *software pipelining* comes from this algorithm. *LA* creates a schedule (by compacting the loop body) for a single iteration of the loop which remains valid when it is overlapped in a pipeline with initiation interval *II*. Such a schedule is longer than *II* cycles (the length is the *iteration time*), but it will form a steady state with a length of *II* cycles when it is overlapped in the pipeline. This type of pipeline is called a *regular pipeline*. All iterations in the loop will be identically scheduled. In the *regular pipeline* there are no resource conflicts if a new iteration of the compacted loop starts every *II* cycles. Moreover, all data dependences in the loop are honored.

The initial compacted loop (for cyclic graphs) is found as follows:

1. Find the strongly connected components in the graph [Meh84], and compute the closure of the precedence constraints in each connected component by solving the all-points longest path problem for each component [Flo62, DBR67].
2. The connected components are first individually scheduled. The original *DG* is reduced by representing each connected component as a single vertex. Edges connecting nodes from different connected components are represented by edges between the corresponding vertices. This *reduced graph* is acyclic.
3. The nodes belonging to a connected component are scheduled (by using *list scheduling*) in a topological order by taking only ILDs into account.
4. The algorithm to schedule the *reduced graph* is also a *list scheduling*.
5. If a node cannot be scheduled in *II* consecutive cycles due to resource conflicts, it will not fit in anywhere within the current schedule. When this happens, the attempt to find a schedule for the given initiation interval is aborted and the scheduling process is repeated with a greater *II* value.

The software pipelining algorithm differs from traditional list scheduling in that a resource reservation table with *II* cycles, similar to the one used in *modulo scheduling* algorithms, is used to determine if there is a resource conflict.

LA calculates a lower and an upper bound on the initiation interval. The upper bound is the iteration time of the loop. The lower bound (*MII*) is calculated by taking resource constraints and recurrences into account. *LA* produces a loop body which will form a *regular pipeline*, which simplifies the creation of the prologue, steady state and epilogue.

Decomposed software pipelining (FRLC) [WE93a, WE93b]

In *decomposed software pipelining* (*DESP*) [WE93a, WE93b], software pipelining is considered as an instruction-level transformation from a vector of one-dimension to a matrix of two dimensions. Rows in the matrix represent cycles of the schedule; columns represent the iteration which instructions belong to. For example, if instruction u is in the element (i, j) of the matrix, this represents that instruction u from the iteration j (u_j) is executed at cycle i of the schedule. If two instructions have the same row number (rn), they will be executed at the same cycle. If two instructions have the same column number (cn), the instance of both instructions comes from the same iteration. The column number and the row number are defined such that $S(u_1) = rn(u) + II \cdot (cn(u) - 1)$ and $S(u_i) = S(u_1) + II \cdot (i - 1)$. Therefore, the software pipelining problem is decomposed into two subproblems: determining the row and the column numbers of instructions in the matrix. Some constraints must be fulfilled:

- Resource constraints: two instructions with the same row number cannot use the same resource (or resource stage for pipelined resources).
- Dependence constraints: all data dependences must be honored.
- Cyclicity constraint: a new iteration (with the same schedule) can start every II cycles without resource conflicts.

The basic idea of *DESP* is very simple. The loop is pipelined in two steps: one is to determine the row numbers and the other to determine the column numbers. Two algorithms are proposed: one to compute the row numbers before the column numbers (*FRLC*), and the other to compute the column numbers before the row numbers (*FCLR*). We will only describe here *FRLC*. The *FRLC* (*first row last column*) algorithm is as follows:

1. Find the strongly connected components (SCC) of the *DG* [Meh84].
2. Remove some edges from these SCCs such that the modified SCCs become acyclic and, in the absence of resource constraints, list scheduling may be used to get the row numbers for the instructions of the SCCs with the minimum possible II.
3. Remove all edges which are not included in these SCCs.
4. Use *list scheduling* [DLSM81] to determine the row numbers by taking resource constraints into account.
5. Determine column numbers by taking dependences into account

Pipeline optimization based on iterative refinements [MD90]

An algorithm for the generation of pipelined designs developed for use in an interactive behavioral synthesis system is presented in [MD90]. The algorithm (*IRIS*) is based on the iterative refinement of initial solutions, and works only with acyclic graphs. The main difference among *IRIS* and other approaches based on iterative refinements (such as *loop folding*, for example), is that the user is able to interrupt the algorithm during the optimization phase, obtaining a partially optimized solution. *IRIS* is divided into three main phases:

1. *Statement of design goals.* The statement of design goals is made by the designer. This takes the form of either a resource constraint or a target initiation interval to be achieved by the final solution. When resource constraints are given, *IRIS* computes first the *MII* of the target design by only taking resources into account. When an initiation interval is given, *IRIS* determines the minimum resource requirement for each type of resource.
2. *Derivation of a base solution.* Unlike *loop folding*, *IRIS* does not have to carry out an iterative scheduling process to determine II, since it has been determined in the first phase. In order to find the base solution, *IRIS* schedules the operations in the loop in a topological order, *as soon as possible* and modulo II. That is, operations are assigned from cycle 0 to cycle $II-1$. When an operation must be assigned to cycle II, it is actually assigned to cycle 0. Therefore, operations to be assigned to cycle C are assigned to cycle C mod II. The folding index of an operation to be assigned to cycle C is $\lfloor C/II \rfloor$.
3. *Iterative optimization.* The iteration time can often be improved without increasing either the number of resources or the initiation interval. Achieving this improvement is the goal of the optimization phase. The optimization is based on swapping folded operations with others of the same class, but with a lower folding index. In particular, *IRIS* strategy consists of swapping an operation with the nearest suitable operation.

Loop winding [HHL91]

By using a *functionally-pipelined data path*, the processing of a sample datum can be started before the completion of the previous one. The idea of *loop winding* is similar to that of *functional pipelining*, except that it is applied to a loop rather than an overall algorithm. In fact, *functional pipelining* is a special case of *loop winding*. The approach proposed in [HHL91] (*LW*) is based on *loop winding*. *LW* optimizes both the initiation interval of a schedule and the iteration time. It is divided into two phases:

- the construction phase, in which a schedule with minimum initiation interval is found.
- the refinement phase, in which the iteration time is reduced.

In order to pipeline the execution of a *DG*, the *DG* is partitioned horizontally into pieces, which are then wound to form a shorter loop DG_S. DG_S contains one instance for each operation of the loop, and it is executed in the *execution window*. The *execution window* is a portion of the

schedule of the overall loop whose length is II cycles (II has been previously determined). The DG formed by the hitherto unscheduled operations is called the *remaining graph*, and is denoted by DG_R. During each step of the algorithm, a set of operations is selected from DG_R. The scheduled graph DG_S is constructed iteratively by adding the operations just selected from DG_R. The algorithm terminates when DG_R becomes empty.

Two different approaches are proposed, depending on whether the loop has LCDs or is without them.

1. When no LCDs exist in the DG, a minimum initiation interval MII is determined by taking resources into account. The length of the *execution window* is MII. Then, the operations in the DG are incrementally partitioned into blocks. During the scheduling of a block, operations in DG_R are scheduled into the *execution window as soon as possible* by using *urgency* [PP88] as priority function to select operations (*forward scheduling*). After that, the already scheduled operations are pulled down so that the operations in DG_R can be scheduled into the earlier cycles during the next pass (*forward scheduling*).

2. When LCDs exist in the DG, the solution space is explored by successively estimating a new initiation interval. A pipelined schedule is performed for each estimation until a feasible solution is found. The estimation starts with the *minimum initiation interval*, and ends with a *maximum initiation interval*. MII is calculated by taking recurrences and resources into account. The *maximum initiation interval* is the number of cycles required for a list scheduling [DLSM81] to schedule an iteration of the loop. The priority function used to select an operation for scheduling first selects nodes belonging to strongly connected components. The algorithm for finding a pipelined schedule of a loop with LCDs first schedules the loop without taking LCDs into account. Following this, it makes a pre-assignment of those operations that produce data for operations in DG_S. Finally, DG_S is reorganized until a feasible schedule is found (*iterative folding*). The *iterative folding* ends when a feasible schedule is found or $MII + 1$ iterations are done.

After a schedule is found, the iteration time is iteratively reduced. A cycle is reduced at each iteration by performing *backward scheduling* with the operations with the greatest *folding index* and by reducing the length of the critical path. This is done by *folding* the tail of a critical path.

Theda.Fold algorithm [LWGL92]

An approach of software pipelining also based on *loop folding* is presented in [LWGL92, LWLG94]. The algorithm is denoted as *Theda.Fold* (TF). TF solves the following problem: "given a loop, a target initiation interval and a set of resource constraints, schedule the loop in a pipelined fashion such that the iteration time of executing an iteration of the loop is minimized". The target initiation interval is initially the MII, calculated by taking resources and recurrences into account. The algorithm consists of two phases:

1. *As soon as possible pipelined scheduling.* In the first phase, the resource constraints are ignored, and only the data dependences and the target II are taken into account. The result is a schedule that has the shortest possible iteration time. The operations in the DG are

scheduled *as soon as possible* by using a *loop folding* schedule similar to the one explained in [GVD89]. If no schedule exists in the expected II cycles, the algorithm halts (II must be increased). Otherwise, the expected iteration time is set to the iteration time of the found schedule.

2. *Rescheduling*. In the second phase, some operations scheduled at cycles such that resource constraints are violated are rescheduled to other cycles. The selection of operations for rescheduling is based on a priority function called *total difference* (also called *variability* in [LWGL92]). *Total difference* is calculated in three steps. In the first step, a look-ahead schedule is built for each candidate operation. In the second step, all the found look-ahead schedules are scored. Finally, the operations are selected according to the score of the produced look-ahead schedules. If no schedule is found, the expected iteration time is increased by one unit and scheduling is performed again.

The two-phase approach in TF is similar to PBS [Nic85], but TF requires less execution time and less memory storage to work (for example, TF only needs a single copy of the DG during loop folding, while PBS needs to unroll the loop until a pattern emerges).

Multidimensional loop folding [Rim93]

Multidimensional Loop Folding ($MDLF$) [Rim93] is an approach aimed at executing a loop with the maximum performance under some given constraints. Three loop transformations are considered, namely *loop unrolling* [DH79], *loop expansion* [RJ94] and *tree height reduction* [HC89]. Software pipelining is performed by means of *loop folding* [GVD89]. If *loop folding* cannot achieve the desired performance due to insufficient (fine-grain) parallelism, loop transformations are used to enhance and extract additional (coarse-grain) parallelism. The loop is then scheduled using conventional *loop folding* techniques.

Loop expansion is a loop-optimizing transformation aimed at software pipelining a nested loop. When the previously described techniques must be applied to a (perfect) multiple-nested loop, software pipelining is performed in the innermost loop, ignoring the remaining loops. The idea of *loop expansion* is different, and it is similar to the one proposed by *Unroll and Jam* [CCK87, Car93]. It consists of unrolling an outer loop and fusing back together the resulting inner loops. The effect is the same as unrolling the innermost loop with respect to the index of an outer loop. Once the loop has been unrolled, it is pipelined. $MDLF$ works as follows:

1. *Select a loop transformation (loop unrolling, loop expansion or tree height reduction)*. In order to select which transformation to apply, $MDLF$ performs an estimation of the expected *minimum initiation interval* of the loop by using the different transformations. MII is analytically computed by taken resource constraints and recurrences in the loop into account.

2. *Select the number of loop instances per iteration after the transformation*. This calculation is also analytically performed (it is not performed if the selected transformation is *tree height reduction*).

3. *Perform the selected loop transformation.*

4. *Schedule the transformed loop.* *MDLF* uses a scheduling approach called *RECALS II*. *RECALS II* is a list scheduling algorithm [DLSM81] based on a priority function called $PALAP\ (pseudo - ALAP)$. The difference between *ALAP* and *PALAP* is that *ALAP* is computed without taking resource constraints into account, whereas *PALAP* is computed by considering a subset of resource constraints. *PALAP* assigns higher priorities to nodes with successors vying for the same resources.

5. *Perform loop folding with the found schedule.*

The loop-optimizing techniques used by *MDLF* are not always useful, and in general, applying such techniques increases the cost (code length, registers requirement) while improving the performance of the schedule.

2.2.5 Linear programming approaches

Several *Linear Programming* (LP) approaches have been proposed for software pipelining in the recent years. Instead of giving heuristic algorithms, as the previously explained methods, LP approaches begin with a mathematic description of the scheduling objectives and constraints, which can easily be translated into *linear programming formulations* [Sch86]. The final objective is to minimize a user-defined cost function. We will describe here some of these techniques. They belong to the categories 1 and 3.

ALPS [HLH91] is an *integer lineal programming* (ILP) approach which tries to reduce the CPU-time by reducing the solution space of the scheduling problem. *ALPS* arranges the data dependence relationships in the formulation, such that the cost function is transformed into a linear function. The search space for each operation (time frame for scheduling) is restricted by a previous calculation of the first and the last cycle at which each operation may be issued. The search space for resources is reduced by introducing a lower and an upper bound in the number of resources of each type. *ALPS* solves *time-constrained scheduling* and *resource-constrained scheduling*. The software pipelining formulation is built based on the concepts from *loop folding* [GVD89]. *ALPS* reduces the number of registers required after a schedule is found.

[VGN92] and [GNV93] present another ILP approach (R-PER) to the software pipelining problem. It is based on the idea that the schedule of all loop iterations do not have to be *regular*. That is, not all the iterations have to be executed in the same manner. The requirement that only *regular* schedules are considered may exclude some optimal solutions [JA90]. A software pipelining improvement may be achieved if previous loop unrolling is considered [JA90, JC92a]. This ILP approach presents a single mathematical formulation which considers both software pipelining and loop unrolling, but does not take resource-constraints into account. The schedule of two consecutive iterations may not be the same, but it will repeat every r iterations. For this reason, the schedule is said to be *r-periodic*.

Another ILP approach (OASIC) is presented in [GE90, GE91, GE92]. Three problems are solved: (i) simultaneous scheduling and functional unit allocation (ii) simultaneous scheduling and register allocation and (iii) simultaneous scheduling and functional unit and register allocation. The approach considers multi cycle pipelined FUs, as well as maximum and minimum timing constraints between pairs (or groups) of operations. The ILP model presented could not be solved by using

general branch and bound techniques due to their size. Therefore, polyhedral theory [NW89] is used to synthesize a solution.

[GAG94] presents SPILP, an ILP approach oriented towards minimizing register requirements under resource constraints. SPILP starts computing the *MII* of the loop by taking resources and recurrences into account. Following this, an ILP formulation tries to find a schedule in the minimum initiation interval by using the minimum number of *buffers* (a concept similar to registers). If no schedule is found, the expected initiation interval is increased and the process is repeated again. The ILP formulation considers multi-operation pipelined functional units and functional pipelining. The formulation is quite general in that it can be used for architectures with homogeneous or heterogeneous FUs. The approach seems to be very powerful, since it achieves optimal results in a few seconds.

[EDA95] presents an approach that schedules the loop operations for the highest steady state throughput and minimum register requirements. The approach determines optimal register requirements for machines with finite resources and for general dependence graphs. Loop iterations with multiple basic blocks are IF-converted. The scheduling method satisfies arbitrary resource and dependence constraints and minimizes the maximum number of live values at any cycle. Although the algorithm finds very promise results, it is too expensive to be integrated in a compiler.

ILP approaches find optimal solutions for the defined cost function. However, when the cost function is a heuristic, the results may not be optimal. For example, *ALPS* reduces register requirements by reducing variable lifetimes. Although in most cases a reduction in variable lifetimes leads to a reduction in the number of required registers, this is not always true. Therefore, an "inappropriate" cost function may guide the ILP approach to a sub-optimal solution. The main disadvantage of such approaches is that, usually, they consume much CPU-time.

2.2.6 Comparisons among the approaches

This Section compares the software pipelining approaches explained at the previous Section by using different features. We will use the following characteristics to compare the methodologies:

1. How the final schedule is built:
   (a) By unrolling the loop and searching for an emergent pattern.
   (b) By scheduling and transforming the loop instruction to instruction.
   (c) By previous precompaction of the loop body by using a scheduling algorithm. The pipelined schedule is found by using the previously compacted loop (usually by means of transformations in the loop schedule).
   (d) By using ILP formulation to find the schedule.
2. Consider (or not) previous unrolling of the loop to increase the throughput (when *MII* is not an integer). This consideration produces a schedule containing several iterations of the loop. In such a schedule, all the iterations are not necessarily executed in the same manner.
3. How resource constraints are considered:
   (a) Consider resource constraints from the beginning.

   (b) Find a schedule by considering unlimited resources, and next consider resource constraints.

4. Consider (or not) multi-cycle pipelined functional units.
5. Consider (or not) conditional jumps.
6. Consider (or not) functional pipelining between the scheduling of consecutive iterations.
7. Consider optimizations on:
   (a) Iteration time.
   (b) Register requirements.

By using the previously enumerated features, table 2.1 presents a comparison among the software pipelining approaches. The last row, labelled as *UNRET*, corresponds to the methodology proposed in this work. Note that there is no mark at point 1. This is because we use an approach different from the previous ones. Our approach transforms the overall loop before being scheduled. This allows us to use any scheduling algorithm proposed in the literature. We optimize both the iteration time and the number of required registers (although they are strongly related, optimizing iteration time does not always guarantee optimizing the number of registers).

The approaches are ordered according to the classification done in Section 2.2.1:

1. Techniques which do not calculate *MII*.
2. Techniques which make an estimation of the *MII*.
3. Techniques which calculate the *MII*:
   (a) By taking resources into account.
   (b) By taking recurrences into account.
   (c) By taking both resources and recurrences into account.

2.3 TECHNIQUES PROPOSED IN THIS WORK

This work proposes methodologies to solve two different problems: *resource-constrained software pipelining* and *time-constrained software pipelining*. Both methodologies are based on the same principles, and the loop pipelining algorithm used is the same.

First, we will briefly show how the approach proposed to solve *resource-constrained software pipelining* works. Henceforth, it will be denoted as *UNRET*. We will also show how other methodologies solve the same example. *Time-constrained software pipelining* will be addressed at the end of this section.

Our first objective is, for a given a set of resource constraints, to determine analytically the minimum number of times a loop must be unrolled in order to obtain the minimum execution time. Such a number is called the optimal unrolling degree.

Category/ Approach		1				2	3		4	5	6	7	
		a	b	c	d		a	b				a	b
1	PP	•					•			•			
	SR			•			•		•				
	RS			•			•		•		•	•	
	ALPS				•		•		•		•		•
	OASIC				•		•		•		•		•
2	URPR			•		•		•					
	Schwa			•			•		•	•	•		•
	LF			•			•		•		•		
3(a)	UBSP	•				•	•		•				
	IRIS		•				•		•			•	
3(b)	OLP	•						•					
	PBS	•						•	•	•			
	R-PER				•	•			•				
3(c)	MS		•				•		•		•		
	LA			•			•		•		•		
	DESP			•			•		•				
	LW			•			•		•		•	•	
	TF			•				•	•		•	•	
	MDLF			•		•	•		•		•		
	SPILP				•		•		•		•		•
	UNRET					•	•		•		•	•	•

Table 2.1 Comparison among different software pipelining approaches

Computing the theoretical optimal unrolling degree is always possible. However, finding a schedule of the unrolled loop with the expected initiation interval is not always possible. Figure 2.2 shows an example. Dependence (E, A) traverses three iterations, and thus E_i must be executed to completion before A_{i+3} starts. The remaining dependences exist within each iteration.

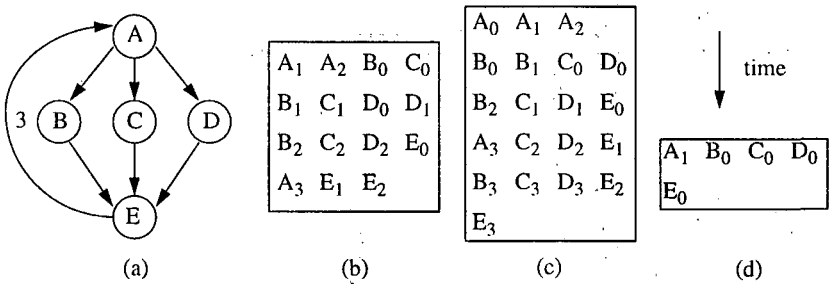

Figure 2.2 Example of *DG* and schedules for 4 adders
(a) Example of *DG* with 5 additions
(b) Schedule found by *UNRET*
(c) Schedule found by using the optimal unrolling degree
(d) Schedule found without previous unrolling

Scheduling without previous unrolling

Let us assume that the loop depicted in Figure 2.2(a) is executed in an architecture with 4 adders which add in a single cycle. A schedule in two cycles (II=2) can easily be found by software pipelining the loop, as shown in Figure 2.2(d). The throughput (average number of iterations executed per cycle) of such a schedule is $Th = \frac{1}{2}$. All the methodologies described in Section 2.2 which do not consider loop unrolling may find such a schedule (or a similar one with the same throughput).

Scheduling by considering previous unrolling

Unrolling the loop may increase the throughput of the schedule. The *minimum initiation interval* of the loop in Figure 2.2(a) is $MII = \frac{5}{4}$, indicating that 4 iterations can be initiated every 5 cycles, with a maximum throughput $Th = \frac{4}{5}$ iterations/cycle[3]. The optimal unrolling degree is therefore 4, and a time-optimal schedule of the unrolled loop must delay $II = 5$ cycles. Unfortunately, none of the current software pipelining approaches has found such a schedule (we challenge the reader to find it).

- *What do other techniques propose?*

  As shown in Section 2.2, when a schedule with the expected II is not found, current approaches [SDX86, RG81, Huf93, Rau94, BC90, PP88] increase the expected II and try to find a longer schedule by using the same unrolling degree.

  Therefore, the loop (unrolled four times) is software pipelined again, looking for a schedule in 6 cycles. Figure 2.2(c) shows the found schedule, with a throughput $Th = \frac{2}{3}$. All the previous techniques that consider loop unrolling find such a schedule as the best one (or a similar one with the same throughput). This schedule is far from the theoretical optimal-time schedule, but it is 1.33 times faster than the schedule found without previous unrolling (see Figure 2.2(d)).

- *What does this work propose?*

  1. **Exploration of the iteration space**

     The strategy followed by *UNRET* when a time-optimal schedule of the unrolled loop is not found is quite different. In order to find a schedule which maximizes the execution throughput, *UNRET* attempts to explore other unrolling degrees instead of increasing the expected II while maintaining the optimal unrolling degree. Thus, *UNRET* explores pairs (II_K, K) in decreasing order of expected throughput. K represents the unrolling degree of the loop, and II_K represents the expected II of a schedule of the loop unrolled K times. Chapter 6 explains how this exploration is done.

     The initial pair to explore, representing an optimal-time schedule, is (II_K, K)=(5,4). As no schedule for 4 iterations in 5 cycles is found, *UNRET* computes the next pair, obtaining (II_K, K)=(4,3). Following this, the loop is unrolled 3 times with a target $II_K = 4$. The unrolled *DG* is software pipelined and a schedule in 4 cycles is found, as shown in Figure 2.2(b). The throughput of such a schedule is $Th = \frac{3}{4}$. Although

[3] The way to compute such a bound will be described in Chapter 3.

this schedule is not time-optimal (for the lower bound computed), it is better than that obtained by any of the approaches described in Section 2.2. A significant speedup is obtained due to the more exhaustive exploration of the unrolling degree of the loop: 1.125 with respect to techniques which previously unroll the loop, and 1.5 with respect to techniques which do not unroll the loop.

2. **Software pipelining**

   The pipelined schedule is found by using an approach different from that previously proposed in the literature. The loop body is first overally transformed (by means of retiming). Since the expected initiation interval is known in advance, the transformed loop body may be scheduled in a *modulo reservation table* [Lam88] by using any known scheduling algorithm. Therefore, different algorithms can be used for scheduling the same (transformed) loop. The treatment of complex operations which use several functional units (FUs) is hidden into the scheduling algorithm, as well the use of pipelined multi-operation FUs and functional pipelining. Other features, such as operation chaining or bus constraints, can be easily included. We use a modified *list scheduling* as scheduling algorithm. Chapters 4 and 5 explain the basis of our algorithm and how it works.

3. **Register optimization**

   As in some techniques described in Section 2.2 [PP88, HLH91, GAG94], *UNRET* also reduces the number of registers required after a schedule is found. This is done in two steps: (i) by reducing the iteration time (*SPAN*) and (ii) by rescheduling some operations. Chapter 7 explains this step in detail.

4. **Time-constrained loop pipelining**

   Finally, time-constrained loop pipelining is considered in Chapter 8. By using some of the features of *UNRET* , we have developed a new algorithm to find the cheapest pipelined loop schedule when the input constraint is a maximum number of cycles (T_{max}). First of all, a pipelined schedule which uses the minimum number of resources is found for a given T_{max}. Then, the throughput of the schedule is increased by using the found set of resources. Finally, the number of required registers is reduced.

2.4 SUMMARY

This chapter describes the basic software pipelining idea. Previous work in software pipelining is also presented. Techniques are divided into three categories: approaches which do not calculate the *MII*, approaches which perform an estimation of the *MII* and approaches which analytically compute the *MII*, Software pipelining techniques are proposed for compilers for parallel architectures (superscalar and VLIW processors), as well as for the high-level synthesis of VLSI circuits. Therefore, the previous work here described includes techniques proposed by communities working in both subjects.

The effectivity of the methodology proposed in this work is shown by using a simple example of resource constrained software pipelining. The example illustrates how our methodology may improve the results obtained by current software pipelining approaches.