UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

1

INTRODUCTION

1.1 MOTIVATION OF THIS WORK

Developing efficient programs for many of the current parallel computers is not easy due to the architectural complexity of those machines. The wide variety of machine organizations often makes it more difficult to port an existing program than to reprogram it completely. Therefore, powerful translators are necessary to generate effective code and free the programmer from concerns about the specific characteristics of the target machine. This work focuses on techniques to be used by an important class of translators, whose objective is to transform sequential programs into equivalent more parallel programs. The transformations are performed at instruction level in order to exploit low level parallelism and increase memory locality.

Most of the current applications are programmed in languages which do not allow us to express parallelism between high-level sentences (as Pascal, C or Fortran). Furthermore, a lot of applications written ten or more years ago are still used today, and it is not feasible to rewrite such applications for many reasons (not only technical reasons, but also economic ones). Translators enable programmers to write the application in a familiar sequential programming language, without concerning their selves with the architecture of the target machine. Current compilers for parallel architectures not only translate a program written on a high-level language to the appropriate machine language, but also perform some transformations in the final code in order to execute the program in a more parallel way. The transformations improve the performance in

the execution of the program by making use of the knowledge that the compiler has about the machine architecture. The semantics of the program remain intact after any transformation.

In general, a program specifies a certain sequence of actions to be performed by the computer. A *restructuring* compiler tries to find groups of those actions such that:

- the actions in a group can be executed in parallel
- two groups can be executed independently of each other
- the executions of two groups can be overlapped or
- a combination of the previous execution schemes can take place

In general, these groups take the form of basic blocks, instruction sequences with no branches into or out of the block. Experiments show that limiting parallelization to basic blocks not included in loops limits maximum speedup to bbetween only two and four in all type of programs [Lil94]. This is because loops often comprise a large portion of the parallelism available to be exploited in a program. For this reason, a lot of effort has been devoted in the recent years to parallelize loop execution.

Several parallel computer architectures and compilation techniques have been proposed to exploit such a parallelism at different granularities. Multiprocessors exploit coarse-grained parallelism by distributing entire loop iterations to different processors. Systems oriented to the high-level synthesis (HLS) of VLSI circuits, superscalar processors and very long instruction word (VLIW) processors exploit fine-grained parallelism at instruction level.

This work addresses fine-grained parallelization of loops. We will briefly describe in Section 1.2 the topics towards which this work is focused: HLS of VLSI circuits and compilers for superscalar and VLIW architectures. Section 1.3 describes how a loop can be represented by means of a directed graph, a usual representation for loops. Sections 1.4 and 1.5 presents an overview of the techniques proposed in the literature for coarse-grained and fine-grained parallelization respectively. Finally, Section 1.6 shows how the algorithms proposed in this work will be presented, and how their complexity will be calculated.

1.2 HIGH-LEVEL SYNTHESIS AND PARALLEL ARCHITECTURES

1.2.1 High-level synthesis

Henceforth, a system to perform the high-level synthesis of VLSI circuits will be called a *HLS system*. High-level synthesis (also called behavioral synthesis) is the task of converting a behavioral description of a digital system into a register-transfer level (RTL) design implementing that behavior.

The behavioral description is usually presented as an algorithmic description given in a procedural programming language (as Pascal [Tri87] or ADA [Gir84]) or in a hardware description language (as

VHDL [IEE88], HardwareC [KD90a], Verilog [TM91] or Silage [Hil85]). The description specifies the functionality of the digital system at a high level, and therefore it does not contain structural information such as component types and their interconnections.

The RTL design consists of a data path and a control unit. The data path contains arithmetic logic units (ALUs), application specific functional units (adders, multipliers, etc.), interconnection units (multiplexers or buses of addresses and data) and storage units (memories or registers). The control unit contains the hardware required to sequence the operations in the data path, and the interconnection lines between the data path and the control unit (bus of control).

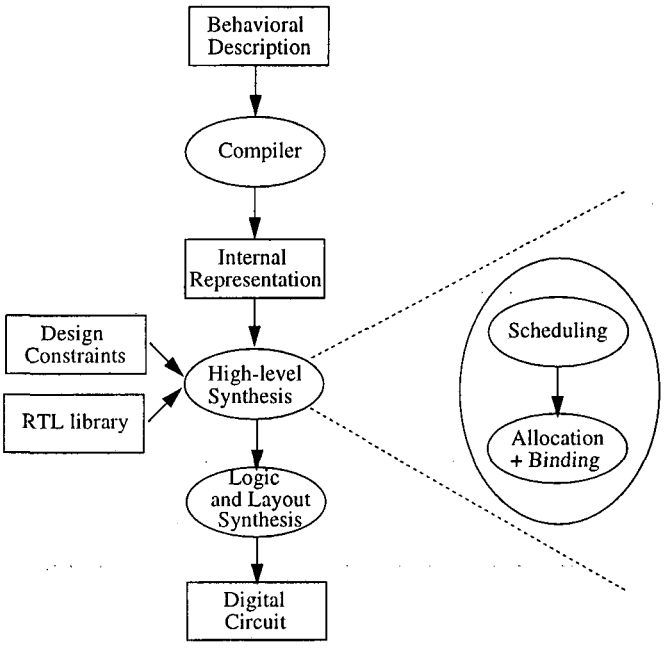

Figure 1.1 High-level synthesis system

Figure 1.1 shows a generic HLS system. The *compiler* transforms the behavioral description into an internal representation (usually a data-flow graph or a parse tree). The *RTL library* contains the physical models of the components to be used. The *design constraints* are in the form of limitations on cycle time, total time to process a datum, area requirements or power consumption.

The two main steps in high-level synthesis are *scheduling* and *allocation*. They are closely interrelated. *Scheduling* (see [WC95] for an overview) consists of assigning operations to cycles. *Allocation* [TS86, ST91] consists of assigning the operations to specific hardware. Once each operation has been assigned to a type of FU, *module binding* is performed to assign operations to specific FUs and variables to specific storage units. The aim of *scheduling* is to minimize the amount of time (or the number of cycles) required to execute the program for the given constraints. In *allocation*, the problem is to minimize the amount of hardware required.

Once the schedule and the data path have been chosen, it is necessary to synthesize the control unit. This can be done in different ways. If hardwired control is chosen, a cycle corresponds to a state in a controlling finite state machine, which can be synthesized by using known methods [San90]. If microcoded is chosen instead, a cycle corresponds to a microprogram step. Finally, the design must be converted into real hardware. Lower level tools such a *logic synthesis* and *layout synthesis* complete the design.

1.2.2 Superscalar processors

Instructions performing scalar computations in a program are called *scalar* instructions. This is the type of instructions typically found in general-purpose microprocessors. A *scalar* processor is a computer implementation able to improve performance by concurrent execution of *scalar* instructions.

The *time taken* by a processor to complete a program is determined by three factors [Joh90]:

- The number of instructions required to execute the program.
- The average number of processor cycles required to execute an instruction.
- The processor cycle time.

Processor performance is improved by reducing the time taken, and therefore one or more of the above factors. Scalar processors reduce the time taken by reducing the average number of processor cycles required to execute an instruction.

The execution of an instruction is divided in three main steps, as shown in Figure 1.2(a):

- Fetch of the instruction.
- Decode of the instruction and assemble operands.
- Execute the instruction and write results.

Scalar processors overlap the steps of different instructions, using a technique called *pipelining* [Kog81], as shown in Figure 1.2(b). The steps do not necessarily take the same amount of time (in fact, instructions take a long time to fetch, compared to the execution time of decode or execute) [Joh90]. Pipelining reduces the average number of cycles required to execute an instruction by permitting the processor to handle more than one instruction at a time. In order to pipeline the execution of instructions, *scalar processors* require both several FUs and specific hardware to detect when two instructions may overlap their execution. The term *superscalar processor* denotes a processor able to issue more than one instruction at a time, as shown in Figure 1.2(c).

1.2.3 VLIW processors

The *instruction parallelism* of a program is defined as the average number of instructions that are executed at a time. Superscalar processors take advantage of instruction parallelism to reduce

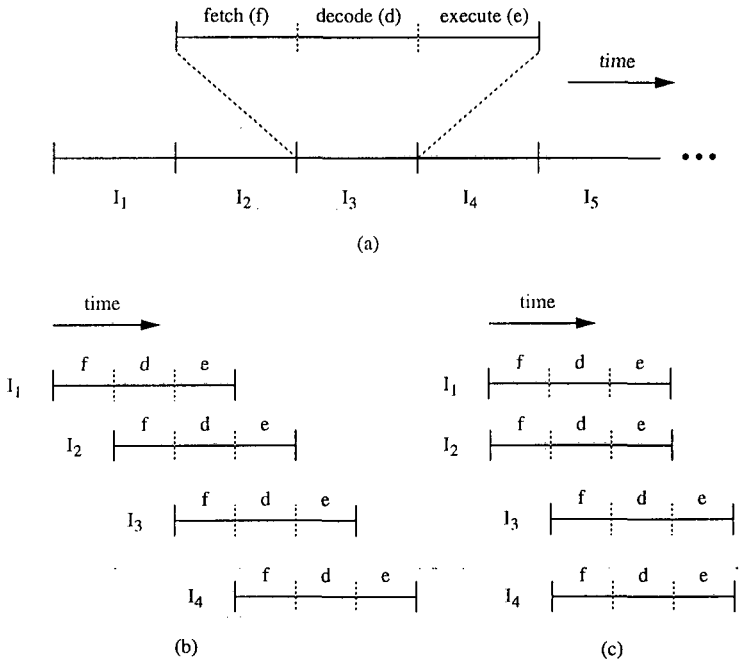

Figure 1.2 Execution of instructions in different processors
(a) Sequential execution
(b) Pipelined execution in a scalar processor
(c) Pipelined execution in a superscalar processor

the average number of cycles per instruction. VLIW processors take advantage of instruction parallelism to reduce the number of instructions.

In a VLIW processor, a single instruction specifies more than one concurrent operation. For this reason, VLIW processors reduce the number of instructions of a program in comparison to superscalar processors. However, the operations specified by a VLIW instruction must be independent of one another in order that the processor may sustain an average number of cycles per instruction comparable to that of the superscalar processor. Figure 1.3 shows an example of execution of instructions in a VLIW processor in which each single instruction performs two different operations.

The instruction of a VLIW processor is normally quite large, taking many bits to encode multiple operations. VLIW processors rely on software techniques as *compaction* [Fis81, Fis83] to pack the collection of operations representing a program into instructions. The more densely the operations can be compacted, the better the performance, and therefore the better the encoding efficiency.

VLIW processors present two main problems:

- They are not software compatible with any general-purpose processor.
- The density of instruction compaction depends on the instruction parallelism.

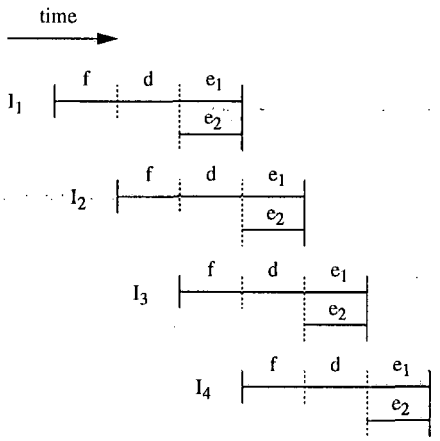

Figure 1.3 Execution of instructions in a VLIW processor

Unlike *superscalar* processors, a VLIW processor does not require special hardware to detect instruction parallelism, because the parallelism is inherently specified by the instruction. The compiler is the only one responsible for detecting which operations may be compacted into a single instruction.

1.3 INTERNAL REPRESENTATION OF LOOPS

1.3.1 Program dependences

The parallelism available in a program is limited by the dependences existing between its instructions. Three types of dependences can be distinguished:

- *Resource dependences* between two statements, consequence of the limited hardware available in the architecture. A *resource dependence* occurs between two instructions when they require the same resource (or resource stage for pipelined resources) at the same cycle.
- *Data dependences* between two statements, which occur when they reference the same memory location or access the same physical registers. There are three types of data dependences:
  - *Flow dependences* (also called *read-after-write hazard*). A flow dependence exists between two consecutive statements, S_1 and S_2, when S_2 requires a value produced by S_1 before starting its execution.
  - *Output dependences* (also called *write-after-write hazard*). These dependences exist between two statements which write a result into the same storage location.
  - *Antidependences* (also called *write-after-read hazard*). An antidependence occurs between two consecutive statements, S_1 and S_2, when S_2 writes a result in a storage location used as input operand by S_1.

```
Q := 0;
for i:=1 to N do
    Q := Q + Z[i] * X[i];
endfor
```

Figure 1.4 Source code for the Livermore Fortran Kernel 3 (Inner product)

Flow dependences are the only *true* dependences in which a result produced by the first statement is used as an input value to the second statement. Both *antidependences* and *output dependences* occur when the programmer or the compiler reuse storage locations. Variable renaming [Tom67] can be used to eliminate these two types of dependences.

- *Control dependences* consequence of the control structures of the programming language (*if-then-else* structures and conditional and unconditional jumps used to implement loop structures).

1.3.2 Data dependence graph

In general, only control and data dependences are taken into account to represent a loop. The representation is in the form of a *control-data flow graph* (CDFG). Nodes in the CDFG represent statements from the loop body, and two different type of edges are considered: edges representing data dependences and edges representing control dependences.

Loops not containing conditional sentences can be represented by a *data dependence graph* (DDG), which is a directed graph representing the data dependences existing among the statements of the loop body. Each vertex of the DDG represents a statement of the loop. An edge between two vertices (statements) indicates a data dependence between the two statements. Edges are labelled with an integer, indicating the number of iterations traversed by the dependence. Symbolic data dependence tests [Ban89] can determine whether data dependences exist between program statements or not.

Let us show an example. Figure 1.4 shows the source code of the *Livermore Fortran Kernel* [McM86] number 3, the *inner product*. In order to execute this loop in a parallel architecture, it is necessary to *compile* it into the appropriate assembly language. For the sake of clarity, we will *compile* it into a pseudo-assembly language. We will use lower case names to represent registers, and upper case names to represent address constants. Array components initialized before the first iteration use a subscript 0; otherwise, the index is the iteration number where they were generated. Compilation of the loop iteration control has been suppressed for simplicity. Figure 1.5 shows the compiled *inner product* by using the previous assumptions.

Figure 1.6 depicts the DDG corresponding to the loop body of the inner product. Dependence from z to u is labelled with an 1 because the result write by z at iteration i is used by u at iteration $i+1$. Dependence from u to w is labelled with a 0 because the result write by u at iteration i is used by w at the same iteration i. A similar reasoning can be used for the rest of the dependences.

```
z_0 = &Z[1]          ; initialize address of the first elements
x_0 = &X[1]
q_0 = 0              ; initialize sum

for i:=1 to N do
   u_i = *z_{i-1}    ; load Z[i]
   v_i = *x_{i-1}    ; load X[i]
   w_i = u_i · v_k   ; multiply Z[i] · X[i]
   q_i = q_{i-1} + w_k ; accumulate product in Q
   z_i = z_{i-1} + 4 ; increment Z[i] address
   x_i = x_{i-1} + 4 ; increment X[i] address
endfor
```

Figure 1.5 Inner product compiled into a pseudo-assembly language

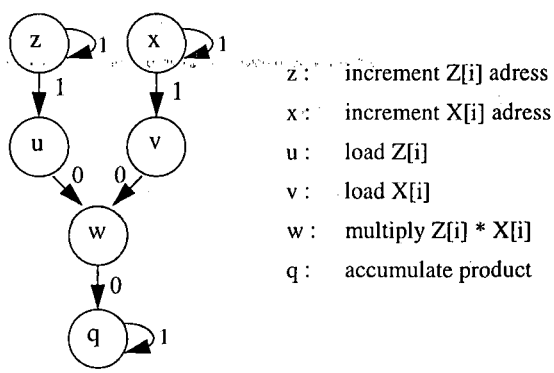

Figure 1.6 DDG of the inner product

1.4 COARSE-GRAINED PARALLELIZATION

Loops are traditionally classified into two categories, according to their parallelism:

- *Doall loops* are loops in which no data dependences exist between instructions belonging to different iterations. In these kind of loops, all the iterations can start their execution simultaneously since no iteration depends on any other [Dav81].
- *Doacross loops* are loops which contain cross-iteration dependences. Therefore, the execution of an iteration requires as input data results computed by previous iterations [Pad79].

Coarse-grained architectures are architectures based on multiprocessors. They exploit parallelism by scheduling entire iterations (and not individual instructions) on separate processors. The *loop scheduling strategy* determines which iterations will be executed on which processor at what time. Two important factors determine the quality of the concurrent code: the balance of processor load and the amount of processor idle time due to synchronization.

For *doall loops*, scheduling strategies can be classified as either static or dynamic, depending on when the task assignment decisions are made.

- *Static parallel loop scheduling*, also called *prescheduling*, assigns iterations to specific processors at compile time. In a real architecture, with P available processors, the iterations can be folded onto the processors in one of two ways [PW86]:
  - The compiler can preschedule the iterations on the loop onto the P processors in contiguous blocks. If the loop has K iterations, processor i executes iterations $(\frac{1+(i-1)\cdot K}{P} \cdots \frac{i\cdot K}{P})$
  - The compiler can preschedule the iterations on the loop by assigning every Pth iteration to the same processor
- *Dynamic loop scheduling*, also called *self-scheduling* [KW84, TY86], can be used to move scheduling decisions from compile time to runtime by making each processor responsible for allocating its own work. At the end of executing an iteration, the processor enters a critical section of code to determine what iteration of the loop it should execute next. *Self-scheduling* works well when the workload for each iteration is relatively large, but may vary between different iterations (for example, due to the existence of conditional code).

For *doacross-loops*, *doacross scheduling* [Cyt84] can be used to distribute consecutive loop iterations onto separate processors. In order to prevent dependence violations, explicit synchronization forces each iteration to start when data input are available. The execution of a doacross-scheduled loop can be modeled as shown in Figure 1.7 [Cyt84]. The total execution time of the loop is:

$$Time = (N-1)\cdot d + N \cdot \text{time to execute an iteration}$$

With infinite resources, each iteration executes on separate processors. In this assumption, the delay d depends on the data dependences of the loop and the time required by the functional units to execute the instructions on the loop. In a real machine with P processors, d also depends on P, because the time due to the synchronization among processors varies according to P.

```
For i := 1 to N do
    delay d · (I − 1);
    execute loop body for iteration I;
Endfor
```

Figure 1.7 Model for doacross scheduling

The compiler can sometimes reorder statements to reduce the effect of the required synchronizations. Weak data-dependence tests may add some synchronizations that are not necessary, so good dependence testing is critical for good performance [Ban89]. The avoidance of redundant control and data dependences may reduce not only the number of synchronization instructions, but also the complexity of the Boolean expressions in some of the resulting *if* statements [BENP93].

In the last three decades a large number of compiler transformations for optimizing programs have been described and implemented. We enumerate some of them below. A good overview and a more complete list can be found in [BGS93].

Loop unswitching [AC71] is applied when a loop contains a conditional with a loop-invariant test condition. The loop is then replicated inside each branch of the conditional, saving the overhead of conditional branching inside the loop, reducing the code size of the loop body, and possibly enabling the parallelization of a branch of the conditional. *Loop-invariant code motion* [CS70] is applied to a computation inside a loop whose result does not change between iterations in order to moving the computation outside of the loop. *Loop peeling* removes dependences created by the first or last few loop iterations. It simply breaks a loop into sections without changing the iteration order. *Loop spreading* [GP88] takes two serial loops and moves some of the computation from the second one to the first one so that the bodies of both loops can be executed in parallel. *Scalar expansion* [PKL80] transforms scalar variables in a loop into temporary arrays. In some cases this allows the elimination of data dependences. *Array expansion* [Fea88] is an extension of *scalar expansion*. It is based on expanding the array by adding an additional dimension to it. *Strength reduction* of induction variable expressions [All69] can be applied to induction variables (variables whose value is determined only by the iteration of the loop) in order to replace a expensive operation with a cheaper one. *Induction variable elimination* [All69] can be used to eliminate the original induction variable once *strength reduction* has been performed. *Loop distribution* (also called *loop fission* or *loop splitting*) [Mur71, AKL81] breaks a single loop into several adjacent loops. It is used to enhance memory hierarchy performance. *Loop Fusion* [Ers66] transforms two adjacent loops into a single loop. *Loop coalescing* [Pol87] combines a loop nest into a single loop, with the original indices computed from the resulting single induction variable. This can improve the scheduling of the loop on a parallel machine, and may also reduce loop overhead. *Loop collapsing* [AC71] transforms two nested loops into a single loop. It is a simpler, more efficient, but less general version of *coalescing* in which the number of dimensions of the array is actually reduced.

Loop reordering transformations change the order in which the iterations of a (multiple-nested) loop are executed. *Loop interchanging* [AK84] is used to interchange the order of two loops. *Loop reversal* [Wed75] changes the direction in which the loop traverses its iteration range. *Loop blocking* (also called *loop tiling*) [AKL81] improves memory locality, primarily the cache, by partitioning the

execution of the innermost loop into blocks. *Strip mining* [Lov77] is used for memory management. It transforms a singly nested loop into a doubly nested one. The outer loop steps through the index set in blocks of some size (according to the characteristics of the target machine), and the inner loop steps through each block. *Inverse strip mining* [HKT91] may decrease the size of the unit of work. It is useful, for example, for distributed-memory processors. In these machines, sending messages that exceed the hardware buffer size may cause the processor to block until the receiving processor acknowledges receipt of the first buffer.

The parallelization of the loop body can also be achieved by transforming the iteration space in a way such that the amount of parallelism available on each iteration is increased. Several techniques have been devised to such purpose [Pol88, Mur71, Lam74].

Loop skewing [Mur71, Lam74] is useful when data dependences exist among instructions belonging to the same iteration. It was invented to handle wavefront computations, so called because the updates in an array propagate like a wave across the array. *Loop skewing* transforms the loop body by changing the expressions involving the loop index. It also changes the loop-counter boundaries.

Cycle shrinking [Pol88] is a transformation which can expose relatively fine-grained parallelism. It is based on increasing the size of the loop body by means of a partial unrolling, and then compacting the loop. The simplest case arises when there are no dependences between instructions belonging to different iterations, and therefore the amount of parallelism available in the loop body is increased proportionally to the number of times the loop is unrolled. This proportional increase also happens when the number of iterations traversed by the dependences is greater than or equal to a certain integer $d > 1$, and the loop is unrolled d times or less.

1.5 FINE-GRAINED PARALLELIZATION

Fine-grained architectures exploit loop parallelism at instruction level by executing several instructions (or operations) per cycle. This work is focused towards the parallelization of loops for these architectures.

In superscalar processors, dependences between instructions must be checked either statically (by the compiler) or dynamically (by the hardware) to ensure that only independent instructions are issued simultaneously. In VLIW processors the compiler checks dependences and reorders the instructions, creating new composed instructions (containing several of the initial instructions of the loop that can be executed in parallel). Each new instruction is executed according to an execution pattern, and no special hardware is required to check for dependences violation (the compiler ensures that dependences will be honored).

Little instruction-level parallelism is found within a basic block because, in general, basic blocks have a small size (typically about 5-20 instructions on the average) [RF93]. Instructions from multiple basic blocks must be concurrently executed if the parallel architecture is to be fully used. Since the branch condition, which determines which block is to be executed next, is often resolved at the end of a basic block, it is necessary to continue the execution along one or more paths before it is known which way the branch will go. This is called *speculative execution*.

Several dynamic schemes for *speculative execution* have been proposed in the literature [HP86, HP87, SP88, SV87]. All this can be expensive in hardware. The alternative is to perform *speculative code motion* at compile time, i.e. move instructions from subsequent blocks up past branch instructions into preceding blocks. Such code motion is fundamental to *global scheduling* schemes such as *trace scheduling* [Fis81] and *superblock scheduling* [HCC89].

Trace scheduling [Fis81] was the first technique able to operate across conditional jumps and jump targets. *Trace scheduling* uses information on the probability that the program would follow a given branch of a conditional jump (*branch prediction*). These probabilities may be computed heuristically or based on profiling information. The most probable trace through the code is selected and parallelized (by compaction), subject only to the constraints imposed by the data dependences. In cases in which control enters or leaves the trace, incorrect results may be produced. In order to resolve this situation, a *recovery* code is introduced at each entry and exit point of a basic block, so that all instructions that would be executed in the original program are also executed in the compacted program. The process is repeated by choosing the most likely (non-overlapping) trace and compacting it.

Superblock scheduling [HCC89] separates the trace selection and code replication from the scheduling. This is done to facilitate the task of incorporating profile-driven global scheduling into more conventional compilers. Instructions can only be moved up above branches, never down. To make this possible, *superblock-scheduling* outlaw control flow into the interior of a trace by means of *tail duplication* (by creating a copy of the trace below the entry point and redirecting the incoming control flow path to that copy). Once this is done for each incoming path, the resulting trace consists of a sequence of basic blocks with branches out of the trace but no incoming branches except to the top of the trace. This is a *superblock*, also known as *extended basic block* in compiler literature.

Trace scheduling and *superblock-scheduling* work with acyclic flow graphs. In cyclic flow graphs (loops), instruction-level parallelism is obtained by overlapping the execution of multiple basic blocks (in loops, the multiple basic blocks are the multiple iterations of the same piece of code). The natural extension of the previous global scheduling ideas to loops is to unroll the body of the loop a number of times and then performing *trace scheduling* over the unrolled loop body [FLS81]. A drawback of this approach is that no overlap is sustained across the back edge of the unrolled loop.

Loop unrolling has also be applied in conjunction with *statement substitution* [KKP+81] to increase the parallelism of the loop body. Given an assignment $x = expression$, *statement substitution* replaces some or all the occurrences of v on the right-hand sides of assignment statements with *expression*. *Statement substitution* increases the length of the right-hand side of assignment statements, and usually enhances the opportunity for parallelization, especially if *tree-height reduction* [Kuc78, HC89] is used. *Tree-height reduction* techniques use associativity, commutativity and distributivity to decrease the *height* of an expression tree and therefore decrease the best parallel execution time of the expression. Nowadays, *statement substitution* is used mostly to help expose the nature of the arrays subscripts in order to allow a more accurate dependence analysis.

The most popular techniques to exploit fine-grained parallelism are known as *software pipelining*. The objective of such techniques is to overlap the execution of consecutive iterations of the loop in such a way that the parallelism of the loop execution increases, making use of the ability of the processor to execute several instructions at a time. *Software pipelining* techniques have also been

proposed for HLS systems and, in fact, its name comes from the work of designers of pipelined functional units. Their objective was to sustain the initiation of successive function evaluations before prior ones had been completed. Since this style of computation is termed *pipelining* in the hardware context, it was denoted *software pipelining* in the programming domain [Cha81].

This work proposes a new *software pipelining* technique. Therefore, in Chapter 2 we will present an extensive overview of the *software pipelining* techniques proposed in the literature. Such techniques have been devised to parallelizing code for superscalar and VLIW processors, as well as to the HLS of VLSI circuits.

1.6 REPRESENTATION OF ALGORITHMS IN THIS WORK

In this work, we will describe algorithms as programs written in a *pseudocode*. What distinguishes *pseudocode* from *real* code is that in *pseudocode* we employ whatever expressive method is the clearest and most concise to specify a given algorithm. In our pseudocode, indentation indicates block structure. Using indentation instead of conventional indicators of block structure, such as *begin* and *end* statements, greatly reduces clutter while preserving, or even enhancing, clarity.

In describing the running time of the algorithms, we will measure the size of the input graph in terms of the number of vertices ($|V|$) and the number of edges ($|E|$). For the sake of simplicity, henceforth we will adopt a common notational convention for the parameters $|V|$ and $|E|$. When using asymptotic notation (such as O-notation), and only when using such a notation, the symbol V will denote $|V|$ and the symbol E will denote $|E|$. This convention makes the running-time formulas easier to read and avoids any ambiguity.

We have programmed in C++ all the algorithms proposed in this work by using the library LEDA (Library of Efficient Data Types and Algorithms) [MN89, Nae93, MN95]. LEDA represents a graph by using double adjacency lists. An adjacency-list representation consist of an array of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list of u contains (pointers to) all the vertices v such that there is an edge $(u, v) \in E$. A double adjacency list consists of two lists, one pointing the successors of every node and the other one pointing its predecessors.

1.7 SUMMARY

In this chapter, we present the motivation that gives rise to this work. First of all, the three types of architectures towards which this work is addressed are introduced. These architectures are:

- Application specific data paths designed by using high-level synthesis tools.
- Superscalar processors.
- Very Long Instruction Word processors.

We also present a brief description of the types of dependences that may exist among the instructions in a program, namely:

- Resource dependences (due to the target architecture).
- Data dependences, which are classified into *flow* dependences, *output* dependences and *antidependences*
- Control dependences (due to the control sentences in the program).

Most of the time required to execute a program is spent within the loops. Dependences among the instructions in a loop are usually represented by a directed graph. We have presented an example.

Dependences in a loop limit the maximum parallelism achievable during the execution of the loop, and therefore they also limit the parallelism of the program. The parallelism of a loop can be exploited at different granularities:

- Coarse-grained parallelization is based on scheduling entire iterations on separate processors.
- Fine-grained parallelization exploits loop parallelism at the instruction level by executing several instructions per cycle.

We enumerate some approaches which exploit parallelism at both granularities. Finally, we describe how the algorithms proposed in this work and their complexity are presented.