Multiprocessor Scheduling of Fine-Grain Iterative Data-Flow Graphs using Genetic Algorithms

Master's Thesis

Erwin R. Bonsma

University of Twente
Department of Electrical Engineering
Laboratory for Network Theory and VLSI design
Enschede, The Netherlands
University of Twente

Supervisors:	Prof. Dr. Ing. O.E. Herrmann
	Dr. Ir. S.H. Gerez
	Dr. Ir. S.M. Heemstra de Groot
	Ir. J. Hofstede
Date:	May 29, 1997
Report Code:	EL-BSC-018N97

Abstract

This thesis presents a method that schedules fine-grain iterative algorithms onto multiprocessor architectures. For every operation in the algorithm, a processor and a start time are chosen. In order to exploit as much parallelism as possible, the scheduling method can generate overlapped schedules. The scheduling method takes communication delays into account. Special care has been taken to design the method so that it is of practical use: the basic hardware model can be extended to include detailed features of the multiprocessor architecture. This is illustrated by implementing a hardware model that requires routing the data transfers over a communication network with a limited capacity. The scheduling method consists of three layers. In the top layer a genetic algorithm takes care of the optimization. It generates different permutations of operations, that are passed on to the middle layer. There, the global scheduling heuristic makes the main scheduling decisions based on a permutation of operations. Details of the hardware model are not considered in this layer. This is done in the bottom layer by the black-box heuristic. It completes the scheduling of an operation and ensures that the detailed hardware model is obeyed. Both heuristics can insert cycles in the schedule to ensure that a valid schedule is always found quickly. The results show that the scheduling method is able to find good quality schedules in reasonable time.

Contents

1

Introduction

Digital signal processing (DSP) is becoming more and more common. The progress in VLSI technology has resulted in an enormous increase in the hardware execution speed. This has caused that a wide variety of signals can now be processed by digital circuits. Not too long ago the processing of digital signals was restricted to low-bandwidth signals as for instance speech; DSP applications have, however, nowadays been extended to include real-time processing of high quality audio and even moving images.

The latter signals require much more computational power than early DSP applications did. Additional processing power is partly provided by the advance in technology and by using DSP processors that are specially designed to suit the characteristics of DSP algorithms. Still, a further increase in computational power is desirable. Therefore, it is very useful to consider executing algorithms on multiple processors in parallel. These "multiprocessors" require, however, a careful scheduling of the algorithm. Every operation must be assigned to a processor and it must must be decided when it is executed. This problem is already difficult on its own, but gets even more complicated when communication delays or other aspects of the multiprocessor architecture are taken into account.

This thesis presents a multiprocessor scheduling method that is able to use a detailed hardware model. It is based on a general hardware model, which already takes communication delays into account. The hardware model can be extended to include more specific characteristics of the multiprocessor. A hardware model that requires routing data transfers over communication links with limited capacity, shows the feasibility of the chosen approach.

Since DSP algorithms are generally executed repetitively, the scheduling method considers iterative algorithms in particular. It can produce overlapped schedules to fully exploit the parallelism that exists in these algorithms.

This thesis is structured as follows. Chapter 2 provides the necessary background. It presents the most important scheduling theory, gives an overview of related research and examines the available multiprocessors. Finally, it gives a detailed formulation of the scheduling problem that is considered.

The scheduling method that is proposed, consists of three layers that each contain a single algorithm. Chapter 3 gives an overview of how these algorithms are connected and explains the most important ideas. After that, the next three chapters discuss each of the algorithms in

full detail. The "global scheduling heuristic" is presented in Chapter 4. This heuristic is based on a simple hardware model but nevertheless makes the main scheduling decisions. Chapter 5 discusses the "black-box scheduling heuristic". This heuristic ensures that every schedule obeys the detailed hardware model. The third algorithm is a genetic algorithm that takes care of the main optimization. It is presented in Chapter 6.

In order to examine whether the scheduling method can be improved upon by making little changes to it, several experiments have been carried out. Chapter 7 presents these experiments and gives recommendations based on the results. Subsequently, Chapter 8 shows the performance of the scheduling method by testing the method on various scheduling problems. Finally, Chapter 9 summarizes the most important conclusions. It also gives recommendations for future research.

One appendix has been included. Appendix A gives a list of the notations that are used in this thesis.

2

The Scheduling Problem

The scheduling problem that is considered in this thesis is the scheduling of fine-grain, iterative algorithms on a multiprocessor architecture. The scheduling method is able to construct overlapped schedules. Overlapped schedules and other concepts from scheduling theory are discussed in Section 2.1. After that, Section 2.2 presents existing scheduling methods that are related to the scheduling problem considered here. Section 2.3 continues with a discussion of currently available multiprocessors suitable to execute fine-grain algorithms. This is useful because it gives an idea on what the requirements are for the multiprocessor model on which the scheduling method will be based. Finally, Section 2.4 ends the chapter by giving a detailed problem formulation.

2.1 Scheduling Theory

The next three sections give an outline of the theory that is relevant for the scheduling problem that is considered in this thesis. A scheduling method requires a specification of the algorithm that it must schedule. A representation that can be used is the data-flow graph, which is discussed in Section 2.1.1. After that, Section 2.1.2 explains the most important basic concepts and terminology of scheduling theory. Finally, Section 2.1.3 gives the theoretical performance bounds that are most commonly used.

2.1.1 The Data-Flow Graph

The representation that is usually used to specify the algorithm that must be scheduled, is the *data-flow graph* (DFG). It consists of a vertex set V and an edge set E. The vertex set contains different types of vertices (or nodes): $V = C \cup I \cup O$ (C, I and O are pairwise disjoint). I and O are respectively the set of input and the set of output nodes. These nodes are used to indicate where the DFG consumes input data and where it produces output data. C is the set of nodes associated with the operations in the algorithm. The duration of each operation $c \in C$ is specified by $c.duration$. It is expressed in *time units* (TU). This unit is related to the global clock signal in a multiprocessor architecture. A time unit corresponds to a single clock cycle of the global clock.

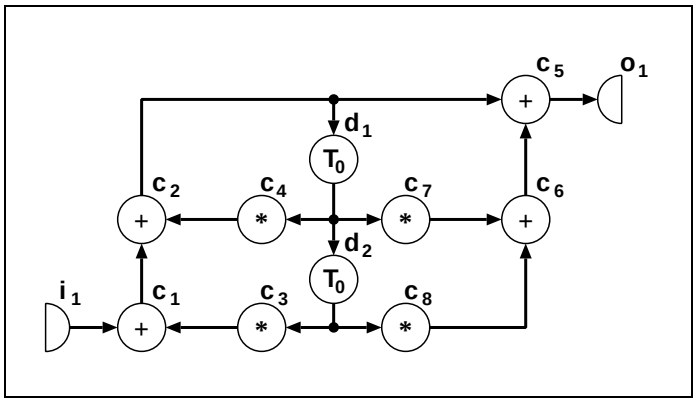

Figure 2.1: *The IDFG for a second-order filter.*

Data dependencies between the nodes in V are represented by directed edges $e \in E$. An edge from node $c_1 \in V \setminus O$ to node $c_2 \in V \setminus I$ implies that c_2 must be executed after the execution of c_1 has been completed because c_2 uses the data produced by c_1.

To schedule an algorithm at compile time, the algorithm must not contain data-dependent conditionals. These algorithms can be described by a *synchronous* DFG. Synchronous DFGs do not contain any data dependencies [HdG90].

DSP applications often execute an algorithm again and again, constantly processing new input samples and producing new output samples. Algorithms that are iterated many times are called *iterative algorithms*. The algorithm is executed periodically. The repetition period is called the *iteration period* and is denoted by T_0. It equals the period with which input samples are consumed and output samples are produced. The value of the iteration period is often expressed in time units.

Iterative algorithms can have data dependencies between operations from different iterations. *Iterative data-flow graphs* (IDFGs) can be used to specify these algorithms. The vertex set V is extended to include the set of delay nodes D: $V = C \cup D \cup I \cup O$ (C, D, I and O are pairwise disjoint). Delay nodes, $d \in D$, are used to represent dependencies between nodes from different iterations. In a figure a delay node is denoted by T_0. Every delay node has associated an iteration offset *d.multiplicity*. This offset indicates the number of iterations that a delay node holds the data it receives before it releases the data at its output. IDFGs can contain directed cycles. However, every cycle must contain at least one delay node as the algorithm can not be scheduled otherwise. An example of an IDFG is shown in Figure 2.1. It is a representation of a second-order digital filter.

The *granularity* of an algorithm represented by a DFG depends on the complexity of the operation nodes. An operation can be *atomic*. An atomic operation is indivisible for the processing element that must execute it and therefore gives the lowest level on which parallelism can be exploited. Operations can also be more complex: an operation node on its turn can be described by a DFG. A DFG in which all operations are atomic is called an *atomic DFG*. It gives a *fine-grain level* description of the algorithm. Another notion that is also used is the *fully specified flow graph* (FSFG). An FSFG is a generic flow graph in which the nodal operations are constrained to be the atomic operations of the constituent processors on which the algorithm will be implemented [GB93].

2.1.2 Some Basic Concepts on Scheduling

When the operations that have to be scheduled and the precedence relations are known beforehand, which generally holds for DSP applications, the scheduling can take place at compile time. This is also known as *static scheduling*. Static scheduling is opposed to *dynamic scheduling*, which schedules operations at run time. This thesis only considers static scheduling.

Another characteristic of a scheduling method is whether or not it allows operations to be interrupted once their execution has begun. If it is possible and interrupted operations can be resumed at a later moment, the scheduling is called *preemptive scheduling*. In contrast, *nonpreemptive scheduling* requires that operations are executed without interruption.

When an algorithm is scheduled for execution on a multiprocessor, several optimization goals can be chosen. It is possible to minimize the *throughput delay* or *latency*. It is the time between the consumption of an input sample and the production of the corresponding output sample. Secondly, for iterative algorithms the iteration period can be minimized.

These two optimization goals are typical for *resource-constrained scheduling*. For resource-constrained scheduling the hardware is specified and it can not be changed by the scheduling method. In contrast, *time-constrained scheduling* tries to use as little hardware as possible when an execution speed is given. The number of processors that is required can for instance be minimized.

Scheduling methods exploit the parallelism that exists in the algorithm between operations from the same iteration (*intra-iteration parallelism*). However, next to that, iterative algorithms often contain parallelism between operations from different iterations (*inter-iteration parallelism*). Scheduling algorithms for iterative algorithms can also exploit this parallelism by executing operations from different iterations in parallel. The schedules that are then produced are called *overlapped schedules*. These schedules are opposed to *nonoverlapped schedules*, where for every iteration period only operations belonging to that iteration are executed. The difference between those two types of schedules is illustrated in Figure 2.2 (a) and (b). The notation for each operation task in the schedule is as follows: the base symbol refers to a specific operation in the DFG, the superscript indicates the iteration that the execution belongs to. It can be noted that both schedules are valid schedules for the second-order filter that was given in Figure 2.1.

Cyclo-static schedules form a special class of overlapped schedules. In a cyclo-static schedule an operation does not have to be executed on the same processing element for every iteration. Schedules corresponding to subsequent iterations can have a constant displacement in processor space. This is illustrated in Figure 2.3 (a). Cyclo-static schedules are opposed to *fully-static schedules*. The latter require that each operation is assigned to the same processing element for all iterations.

Within the class of cyclo-static scheduling several other classes of schedules can be distinguished, for instance: skewed single instruction multiple data (SSIMD) schedules and parallel skewed single instruction multiple data (PSSIMD) schedules [CM94]. In SSIMD a schedule consists of a single row. All operations from the same iteration are executed by a single processor. The next iteration is executed by a different processor and starts a number of time units later. PSSIMD schedules resemble SSIMD schedules. The difference is that a group of

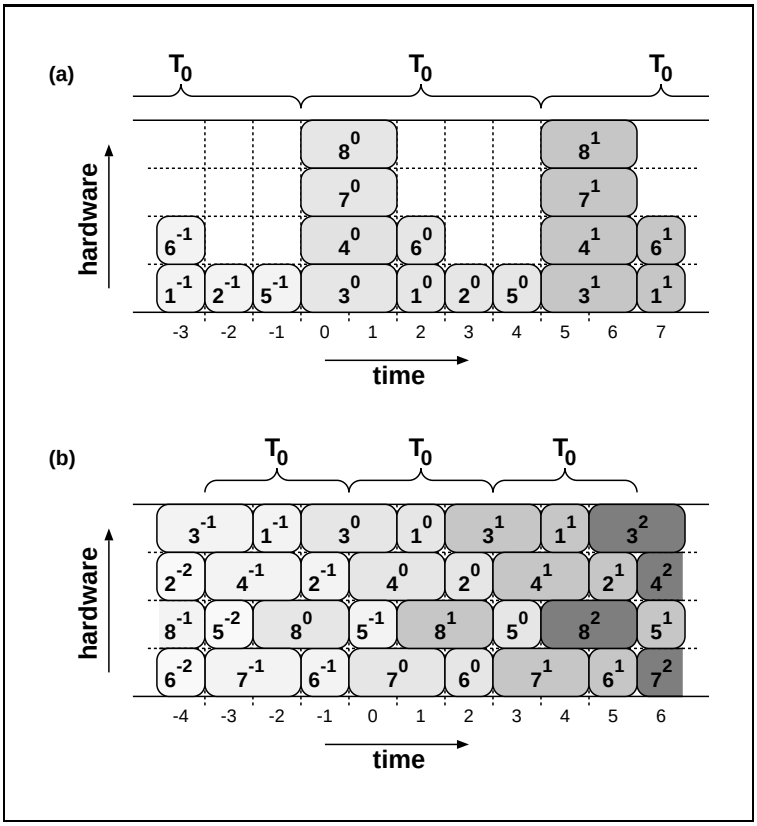

Figure 2.2: *(a) A nonoverlapped schedule, and (b) an overlapped schedule.*

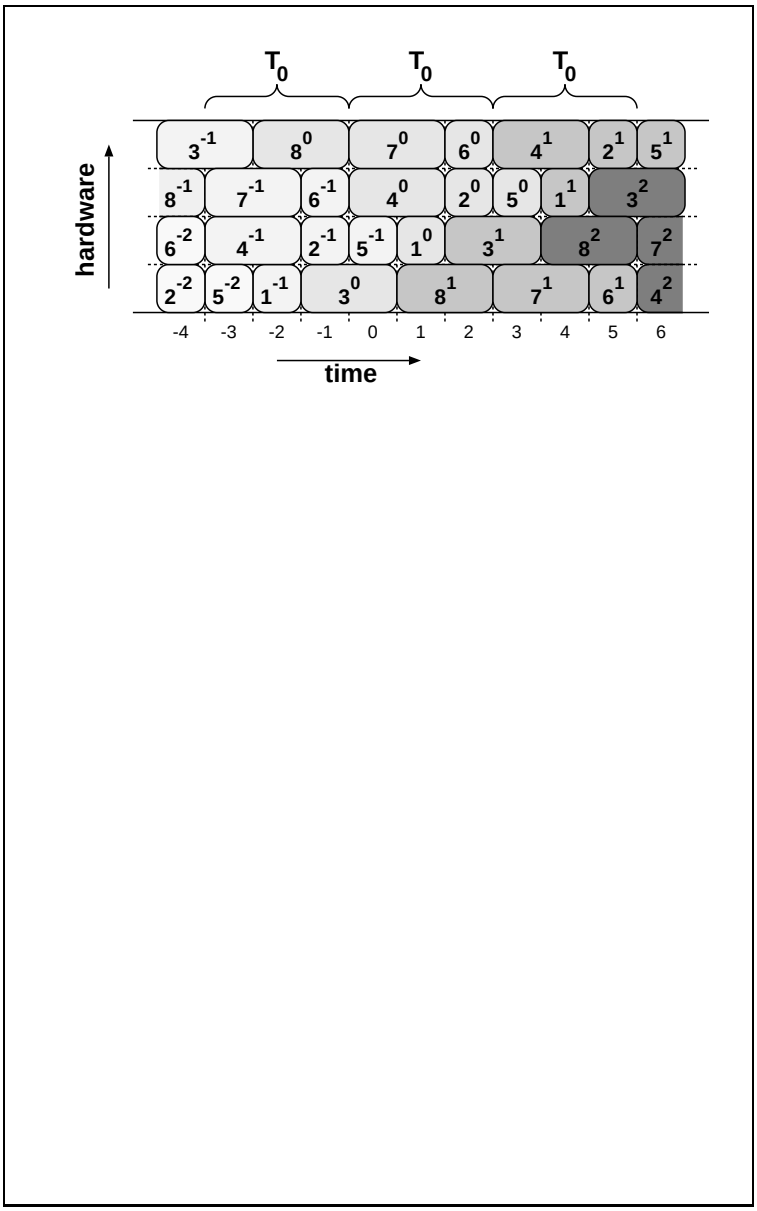

Figure 2.3: *(a) A cyclo-static schedule, and two special cases: (b) a SSIMD schedule, and (c) a PSSIMD schedule.*

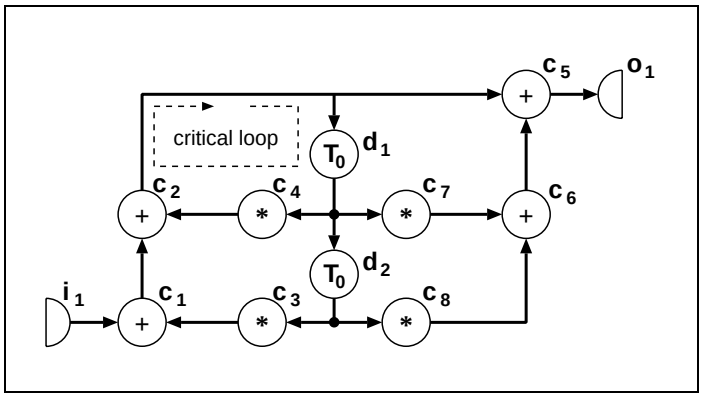

Figure 2.4: *The critical loop in the second-order filter.*

processors executes operations from one iteration instead of a single processor. Examples of these two schedule types are shown in Figure 2.3 (b) and (c).

The multiprocessor model can consist of *homogeneous* or *nonhomogeneous processors*. When processors are homogeneous they all have the same external behaviour. Nonhomogeneous processors however, do not have an equivalent behaviour. They can for example differ in execution speed or they can differ because they can not all handle the same operations.

2.1.3 Performance Bounds

There are some performance bounds for multiprocessor scheduling problems. These bounds give minimum values for some of the optimization goals that can be chosen. These bounds can not always be achieved for every multiprocessor configuration. Still, they provide a means to estimate the quality of schedules found by a scheduling method. Furthermore, they can be used by the scheduling method to guide its search for a good schedule. Some bounds that are commonly used (see for instance [CM94] or [HdG90]) will be given here.

The *iteration period bound* (IPB) gives a lower bound on the iteration period when unlimited hardware is available. When the DFG contains no loops, the iteration period can be made arbitrarily small. Otherwise IPB is calculated as follows:

$$IPB = \max_{l \in L} \left\lceil \frac{\sum_{c \in C \cap n(l)} c.duration}{\sum_{d \in D \cap n(l)} d.multiplicity} \right\rceil \tag{2.1}$$

with L the set of loops in the DFG, and $n(l)$ the set of nodes in loop l. Note that $C \cap n(l)$ is the set of all computational nodes in loop l and similarly, $D \cap n(l)$ is the set of delay nodes in the loop. The loop l that results in the maximum value that leads to the value of IPB, is called the *critical loop* of an IDFG.

To give an example, consider the second-order filter that is given in Figure 2.1. When an addition requires 1 TU to execute and a multiplication 2 TU, IPB can be calculated. Figure 2.4 shows the critical loop for the second-order filter. It easily follows that the minimal iteration period is three, for $IPB = (2 + 1)\,\mathrm{TU}/1 = 3\,\mathrm{TU}$.

Closely related to the concept of the critical loop is the *slack time* [HdG90]. The slack time of a loop l is given by:

$$slacktime = \sum_{d \in D \cap n(l)} d.multiplicity \times T_0 - \sum_{c \in C \cap n(l)} c.duration \tag{2.2}$$

with $n(l)$ the set of nodes contained in loop l.

When $T_0 = IPB$, the slack time of the critical loop of a schedule is usually zero[1]. A slack time of zero implies that when one operation in the loop is scheduled, there is no freedom of choice for the start times of the other operations in the loop. These are then all fixed. When the slack time of a loop is larger than zero, there is more freedom of choice to schedule the operations in the loop[2].

Another bound is the *periodic delay bound* (PDB). PDB gives a lower bound on the latency of the schedule when the iteration period is equal to IPB. The bound is given by:

$$PDB = \max_{p \in P}(\sum_{c \in C \cap n(p)} c.duration - IPB \times \sum_{d \in D \cap n(p)} d.multiplicity) \tag{2.3}$$

with P the set of paths from input to output and $n(p)$ the set of nodes in a path p. The path $p \in P$ that determines the value of PDB (in other words, the path responsible for the maximum value) is called the *critical path*.

The third bound is the *processor bound* (PB). It gives the minimum number of processors that are necessary to have a schedule with $T_0 = IPB$.

$$PB = \left\lceil \frac{\sum_{c \in C} c.duration}{IPB} \right\rceil \tag{2.4}$$

Related to this bound is the *iteration period bound for a fixed number of processors* (IPBFP), as given in [HdG90].

$$IPBFP = \left\lceil \frac{\sum_{c \in C} c.duration}{N_P} \right\rceil \tag{2.5}$$

Where N_P is the number of processors that is available. The maximum value of the two values IPB and IPBFP can be used to estimate a minimum iteration period when the number of processors is fixed.

[1] The slack time does not always have to be zero. It can be larger than zero when the denominator in Equation 2.2 is larger than one.

[2] At least, when the rest of the schedule is ignored. Operations can be part of more than one loop.

2.2 Related Scheduling Methods

This section discusses related scheduling methods that have been reported in literature. In Section 2.2.1 scheduling methods that can schedule iterative algorithms are distinguished from those that are restricted to noniterative algorithms. Section 2.2.2 then highlights some of the scheduling methods for iterative algorithms. Section 2.2.3 continues by discussing scheduling methods that take communication delays into account. Subsequently the next two sections are devoted to scheduling methods that make use of general optimization techniques. Section 2.2.4 gives scheduling methods that use nonlinear and linear integer problem formulations. The application of genetic algorithms to the scheduling problem is discussed in Section 2.2.5. Finally, Section 2.2.6 presents scheduling methods for iterative algorithms that do not ignore communication delays and that furthermore consider the DSP architecture.

2.2.1 Iterative Algorithms versus Noniterative Algorithms

A large part of the available literature on scheduling methods does not consider iterative algorithms but concentrates on scheduling methods that generate efficient and short schedules for a single execution of an algorithm. Scheduling methods for noniterative algorithms try to minimize the total execution time by exploiting the available parallelism in the DFG as much as possible. As it has already been mentioned, scheduling methods for iterative algorithms can also exploit inter-iteration parallelism by generating overlapped schedules.

This difference causes that the scheduling problem for iterative algorithms, where minimization of the iteration period is often more important than minimizing the throughput delay, can not be solved efficiently by scheduling methods for noniterative algorithms. However, these methods can still be used to schedule iterative algorithms. When there are dependencies between different iterations, the corresponding delay nodes must be converted to pairs of input and output nodes. The resulting graph (without cycles) can then be scheduled using the scheduling method for noniterative algorithms. However, the resulting schedule will be nonoverlapped and inter-iteration parallelism will not be exploited.

A technique known as *direct blocking* (see for instance [HdG90]) can be used to improve the throughput. This technique concatenates several iterations of the algorithm to create a new DFG. This larger DFG is then scheduled instead of the original DFG. A disadvantage of direct blocking is that the resulting schedules are longer. When a scheduling method does not make use of direct blocking, *unblocked* schedules are generated. The same schedule is used for all iterations of the algorithm.

2.2.2 Scheduling Methods for Iterative Algorithms

To exploit inter-iteration parallelism, scheduling methods have been designed that specifically consider iterative algorithms. Well-known examples are a cyclo-static scheduling method that uses exhaustive search [Sch85], the maximum-spanning tree method and optimum unfolding [PM91]. Short descriptions of these scheduling techniques and a discussion of their advantages and disadvantages can be found in [MC94], [GB93] and [HdG90]. This will not be repeated here, because all these methods have in common that they do not consider communication delays

nor any other aspects of the DSP architecture. These scheduling methods are not particularly relevant to the scheduling problem considered in this thesis.

Another scheduling method for iterative algorithms can be found in [GB93]. A method is given that generates optimal schedules. The scheduler performs an exhaustive search and finds a schedule that satisfies the iteration period bound, the processor bound and the periodic delay bound if such a schedule exists. If no large bottlenecks are present in the FSFG (and the calculated bounds can be met) there is hardly any backtracking. Therefore the average-case time-complexity is polynomial. However, the scheduling method has an exponential worst-case time-complexity. The method does not directly consider communication delays or the DSP architecture. Only after a schedule has been found these are taken into account during the processor assignment and code generation stage. However, because of this approach it is likely that the final implementation is not optimal. The deviation from the optimal solution is probably even considerable, as is explained briefly later on.

The scheduling-range-chart method, presented in [HdG90, HdGGH92], is another scheduling method for iterative algorithms. This heuristic repetitively determines for each operation the time-span within which the operation can be scheduled (the *scheduling range* of the operation). The scheduling ranges are used to schedule an operation at every step of the scheduling method. This method can consider nonnegligible communication delays. However, the assignment algorithm which takes the communication delays into account is not completely specified. Furthermore, it is very hard to extend the method so that it also considers the architecture of the DSPs.

2.2.3 Nonnegligible Communication Delays

There is a reasonable amount of research done on multiprocessor scheduling methods that do not ignore communication delays. The majority of this research does not address iterative algorithms, but attempts to find efficient schedules for a single execution of an algorithm. An example can be found in [Kaw92]. A heuristic is presented that schedules noniterative tasks on a message passing multiprocessor system. The main effort is focused on a routing algorithm that finds a transmission pattern that results in the minimum communication delay for a given message, when its sender and receiver processor and a start time are given and some communications have already been scheduled. The scheduling method is a list scheduler and it uses the proposed routing algorithm. At each step of the scheduling method a task is chosen depending on its priority and it is assigned to a processor. All incoming messages for the task are scheduled using the routing algorithm. After all messages have been routed the start time of the task is set and a new task is chosen to be scheduled.

In [SSRM94] another heuristic is presented that schedules noniterative precedence-related tasks onto a multiprocessor with nonnegligible intertask communication in order to minimize the completion time. The heuristic is based on list scheduling combined with a routing algorithm. The main goal was to improve existing heuristics by exploiting schedule holes.

2.2.4 (Non)linear Integer Problem Formulation

None of the scheduling methods mentioned so far consider the architecture of the processors that are used. The scheduling method presented by [KKT90] looks at a small part of the DSP architecture by taking memory constraints into account. Communication delays are also considered. However, the granularity of the algorithms that are scheduled by the proposed method is not fine grain. A node in the DFG represents for instance a 1024-point FFT calculation. Block-type and stream-type tasks are distinguished in order to enhance parallelism by using pipelining when this is possible. These concepts are, however, not relevant for fine-grain level algorithms. Schedules are generated by solving a nonlinear integer problem using a branch and bound algorithm.

Integer linear programming (ILP) can be used to solve the scheduling problem. In integer linear programming, a problem is written as a set of linear equations involving integer variables. There are several dedicated tools available that solve these equations efficiently. [CHK94] use ILP to model the scheduling and partitioning problem for noniterative algorithms onto multiprocessor systems. They present an exact ILP formulation that performs scheduling and partitioning simultaneously. The model supports processors with more than one functional unit and it considers the timing of communications precisely. Several communication architectures are supported. The flexibility of the ILP formulation makes it possible to consider several communication architectures. They present exact formulas for buses (nonbuffered communication) and FIFO queues (which implement buffering). The formulation is too complex for algorithms with a large number of operations. Therefore they also present a simplified ILP and an iterative partitioning heuristic. Large algorithms should be scheduled using a combination of these three algorithms. Each algorithm should be used at a different stage.

An advantage of ILP is the flexibility and the ease to support different features of the DSP architecture (multiple functional units per processor, limited memory, etc.) or to model the communication network. A disadvantage is that the time complexity is exponential. Especially when the model becomes sophisticated, the size of the algorithms that can be scheduled is restricted.

2.2.5 Genetic Algorithms

Another approach to obtain optimized schedules when the DSP architecture and the architecture of the communication network is relevant, is the use of genetic algorithms. [DAA94] propose a problem-space genetic algorithm to solve the multiprocessor scheduling problem. They consider scheduling noniterative precedence-related tasks with nonnegligible inter-task communication. Their approach is based on the combination of a genetic algorithm with a simple list-scheduling heuristic. The heuristic is designed to schedule algorithms when communication delays are negligible. So, when it would be used on its own, the schedules that are generated when communication delays are nonnegligible, can probably be improved upon considerably. The genetic algorithm optimizes the original schedules by distorting the task-priorities that are used by the list-scheduling heuristic. The advantage of this approach is that schedules with nonnegligible communication can be obtained relatively simply (for the scheduling method does not have to consider the complications of nonnegligible communication directly). Additionally, the heuristic sees to it that the solution space is searched in an

intelligent way.

The above is an example of a *hybrid genetic algorithm*. The main characteristic of these type of genetic algorithms is that they make use of problem-specific knowledge in an intelligent way. This can be done by using a problem-specific heuristic, but also by using a genetic encoding and genetic operators and that are tailored to the problem. One of the advantages is that in this way the creation of infeasible solutions can be avoided. This is important when the scheduling problem is solved by a genetic algorithm. The reason is that precedence constraints and limitations imposed by the hardware, cause that it is very hard or even impossible to design a direct encoding of solutions that is able to generate feasible schedules after crossover has been applied.

Recently, Heijligers also reported good results in the application of hybrid genetic algorithms to the scheduling problem [Hei96]. He considered a high-level synthesis problem where fine-grain algorithms had to be scheduled and assigned to a set of hardware resources. Operations are scheduled by a topological-permutation-based heuristic which maintains time ranges for all operations to guide its scheduling decisions. The scheduling method supports iterative algorithms because it includes a strategy to create pipelined schedules. However, the hardware model did not include communication delays.

The good results that have been obtained by hybrid genetic algorithms and the flexibility that they offer, is the reason that the proposed scheduling method makes use of a genetic algorithm. Genetic algorithms are discussed in more detail in Chapter 6.

2.2.6 Communication Delays and DSP Architecture

There is not yet much research done regarding scheduling methods for iterative algorithms that consider communication delays and the architecture of the DSPs. That this is certainly worthwhile is shown by Curtis and Madisetti [CM94]. Their objective is to use realistic structural and behavioral level descriptions of DSPs to find rate optimal and processor optimal schedules. They developed the DSMP-C1 method for this purpose. This method considers the location of the operands, the number of accumulators and registers, the size of on-chip and external memories, the size of communication buffers, the inter-processor communications and pipelining. The DSMP-C1 method is computationally intensive and is only practical for small FSFGs. However, it clearly shows the speed-up that can be gained by considering the DSP architecture. The iteration period of the schedule obtained with the DSMP-C1 method is on average more than twice as short as the iteration period of schedules obtained by scheduling methods which do not directly consider the DSP architecture.

Curtis and Madisetti also give another approach for the scheduling of iterative algorithms without ignoring communication delays or the DSP architecture [MC94]. They present integer linear programming formulas to model the internal processor structures and the external interconnection network. These models are solved by 0-1 integer linear programming to obtain an optimal schedule for the chosen optimization goal. Different interconnection structures are considered ranging from a fully connected cost model to a randomly connected cost and capacity model. Furthermore, DSPs that consist of multiple functional units are supported and memory and registers of the DSPs are taken into account. The majority of the models that are presented, perform scheduling and mapping independently and generate only near-optimal schedules. One

model is given that combines scheduling and processor mapping (and thus generates a global optimal schedule), but solving this problem takes considerably longer. An advantage of this method is again its flexibility. Besides, an optimal solution is guaranteed. A disadvantage is that the method is computationally intensive. The time complexity is exponential.

2.3 Multiprocessors

This section gives some background on the multiprocessor model that will be used. It consists of three sections. Section 2.3.1 presents some existing multiprocessor architectures that are suitable to exploit fine-grain parallelism. Subsequently, Section 2.3.2 explains why it is recommendable that a scheduling method uses a detailed multiprocessor model. Finally, Section 2.3.3 summarizes the previous two sections by given two important requirements for the multiprocessor model.

2.3.1 Available Multiprocessors

In order to execute fine-grain algorithms efficiently, the communication delay between different processors in a multiprocessor must be very short (approximately the same as the instruction execution time). Most multiprocessor architectures have a communication setup time that is much too high to meet this demand. There are currently only a few multiprocessor architectures suitable for the implementation of fine-grain algorithms. Three of those will be mentioned briefly.

Barnwell and Madisetti present experimental laboratory digital signal multiprocessors that have been designed at Georgia Tech to provide experimental verification of their scheduling research [BM93]. They give the following recommendations for architectural changes to digital signal processors to make them more suitable for fine-grain parallel signal processing:

- Change from multiply-accumulate (MAC) to multiply-add-to-multiported-register-file (MAMPORT) architecture.
- Enable sharing of multiported register files between processors.
- Allow communications and processor operations in parallel without the introduction of wait states.
- Reorganize on-chip pipelining to minimize delay between data input and result storage.

They note that some of these features are beginning to appear in commercially available DSP chips.

Another multiprocessor is the PADDI [CR92]. It is a field-programmable multiprocessor IC that has been designed for the rapid prototyping of high-speed data paths typical to real-time DSP applications. A sophisticated crossbar network connecting the execution units ensures the fast communication that is necessary to implement fine-grain algorithms.

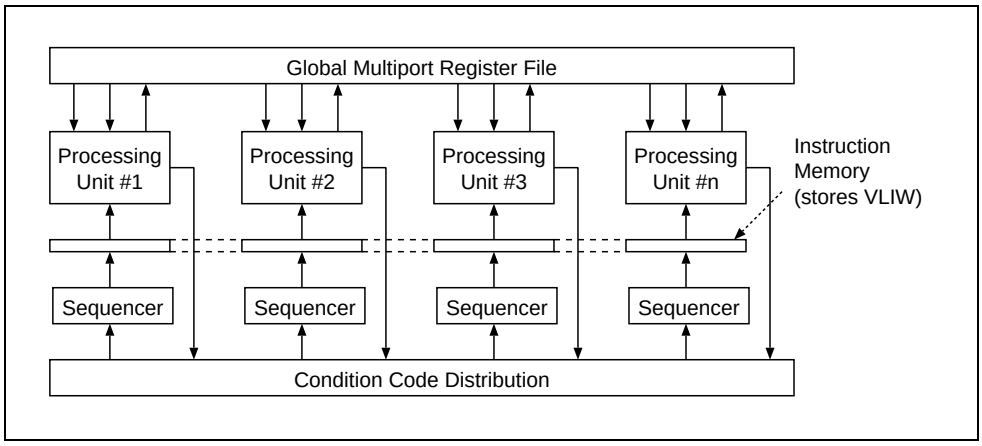

Figure 2.5: *An example of a VLIW architecture: the XIMD architecture.*

The third architecture that is mentioned here is actually not a multiprocessor. However, it is included because it can also be used as target hardware for fine-grain algorithms. It is the set of *very long instruction word* (VLIW) architectures. A VLIW architecture is a processor that contains a considerable number of functional units that are tightly synchronized. [Böc95] reports several VLIW architectures. One of those is shown in Figure 2.5. It can be seen that the architecture supports fine-grain parallelism.

Because there are currently only a few multiprocessors available that can exploit fine-grain parallelism, it is wise not to design a scheduling method that depends too much on a certain multiprocessor model. The small number of suitable multiprocessors and the differences between them would make it hard to set up a good model. Additionally, it is questionable whether the scheduling method can then be applied to future multiprocessor architectures.

2.3.2 A Realistic Hardware Model

However, the above does not mean that the scheduling method should not use a detailed realistic model of the multiprocessor. When a scheduling method does not consider a complete and realistic model of the multiprocessor architecture, the resulting schedules (although close to optimal with respect to the underlying model) will result in inefficient implementations. [CM94] shows that it is certainly worthwhile to explicitly consider the multiprocessor architecture. For then schedules are produced that are on the average more than twice as short as schedules that are based on a less realistic model (and therefore had to be modified before they could be executed on the actual hardware).

So, it is desirable that a multiprocessor scheduling method is based on a realistic model of the multiprocessor. A realistic model can for instance support: communication delays, contention of communication links, the structure of the datapath in the processor, allocation of registers, the restrictions imposed by the instruction set, pipelining, etc.

It is unlikely that a scheduling method based on a complex hardware model will always produce schedules that are close to optimal. It should, however, be noted that this is not necessary to outperform the results obtained by less realistic scheduling methods.

2.3.3 Summary

The previous two sections have led to two requirements that were used during the design of the proposed scheduling method:

- *The scheduling method should be flexible with respect to the multiprocessor architecture.*
- *It is preferable that the scheduling method creates schedules that can be implemented directly on a realistic multiprocessor architecture.* The schedule method should at least be designed in such a way that the method can be extended to create schedules that are more realistic.

2.4 Problem Formulation

This thesis considers overlapped fully-static scheduling of iterative algorithms. The scheduling problem is resource constrained. The optimization goal is to minimize the iteration period. Next to that, the latency is minimized. However, minimization of the iteration period is always valued more.

The algorithm is specified by an IDFG. The IDFG consists of operation nodes $c \in C$, delay nodes $d \in D$, input nodes $i \in I$ and output nodes $o \in O$. These nodes are connected by directed edges $e \in E$ that represent the data dependencies.

Each operation c has a fixed length $c.duration$ that is independent of the hardware on which it is executed. However, different operations can have different execution lengths. Different *operation types* can be distinguished, for example: addition, multiplication and subtraction. The type of each operation is given by $c.type$.

A hardware configuration is also specified. The atomic hardware processing elements that can execute the operations are called *functional units* or FUs in short. This name is chosen because the hardware model can support multiprocessor configurations that have processors with more than one FU (e.g. a multiplier and an ALU).

The set of functional units is given by F. The number of FUs is $|F|$. The FUs can be *nonhomogeneous* because not every FU has to support the same set of operation types. The operations supported by each FU are specified by the set $fu.optypes$, $fu \in F$.

All FUs are synchronized and use a single global clock. The execution of an operation always starts at beginning of a time unit and ends at the end of a time unit. Preemptive scheduling is not allowed.

The hardware model includes communication delays. The scheduling method must obey the *hardware distance matrix* $\mathbf{D}_h$. The subscript h refers to hardware. $\mathbf{D}_h(fu_1, fu_2)$ with $fu_1, fu_2 \in F$, specifies the minimal communication delay that occurs when data is transferred from fu_1 to fu_2. It is required that $\mathbf{D}_h(fu, fu) = 0, \forall fu \in F$.

The above is the basic hardware model that is used by the scheduling method. However, the scheduling method can be extended to support more sophisticated models. At a later moment in

this thesis the model is expanded to include a communication network with a limited capacity. The scheduling method must then also route the data transfers. The scheduling problem is further complicated because contention in the network can result in larger communication delays than those specified by the distance matrix $\mathbf{D}_h$. Chapter 5 discusses this more detailed hardware model.

The scheduling method assigns every operation $c \in C$ to a FU and sets *c.fu* to reflect this assignment. It also specifies when the operation is executed by setting *c.starttime* to the start time of the operation for iteration 0. The scheduling method produces a valid schedule: all precedence constraints in the IDFG and the entire hardware model, are obeyed.

3

The Scheduling Method

This chapter consists of four sections that jointly give an outline of the proposed scheduling method. Section 3.1 gives an overview of the structure of the scheduling method and mentions the three layers that can be distinguished. Section 3.2 then follows by explaining why this approach is chosen. The last two sections briefly discuss two layers of the scheduling method in more detail: Section 3.3 discusses the global scheduling heuristic and Section 3.4 the black-box scheduling heuristic.

3.1 Overview

The scheduling method consists of three layers, as illustrated in Figure 3.1. A *genetic algorithm* in the upper layer takes care of the main optimization goal. It is a hybrid genetic algorithm because it uses two problem-specific heuristic algorithms located in the two lower layers to generate schedules. The genetic algorithm is able to influence the schedules that are produced by these heuristics because it provides the order in which the operations are scheduled. The genetic algorithm tries to optimize the schedules that are produced by varying this order.

The middle layer consists of a *global scheduling heuristic* (or global heuristic in short) that is based on a general multiprocessor description. This description contains all the available functional units with the set of operations they support and a simple model of the communication network (namely, the minimal communication delays between each pair of functional units). The global heuristic ignores more detailed characteristics of the multiprocessor model when making scheduling decisions.

To make sure that the resulting schedule can be executed on the multiprocessor, the global heuristic cooperates with a *black-box scheduling heuristic* (or black-box heuristic in short) that can be found in the third layer. It is called "black box" because the global heuristic sees the heuristic as a black box; i.e. the global heuristic communicates with the black-box heuristic in a fixed way and the global heuristic does not need to be adapted for different behaviours of the black-box heuristic. The black-box heuristic uses a detailed description of the multiprocessor architecture and schedules each operation completely. Depending on the multiprocessor model it may be necessary to move input operands to the inputs of the functional unit, to route interprocessor data transfers, to allocate registers, etc.

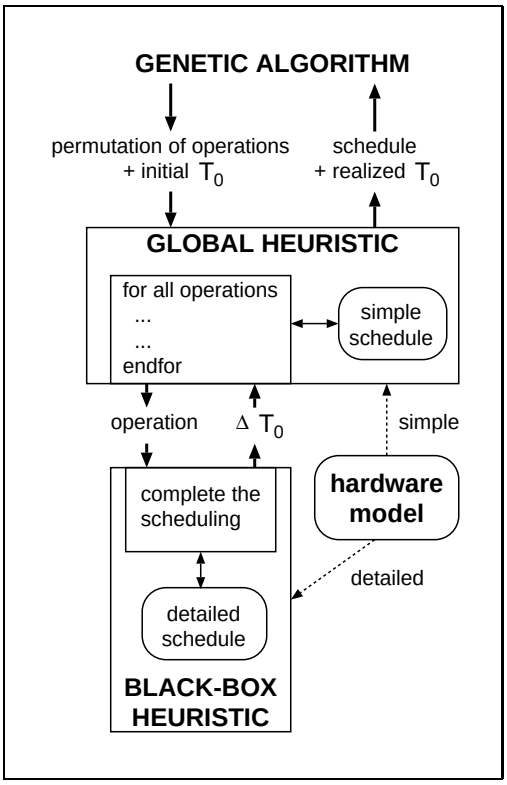

Figure 3.1: *Structure of the proposed scheduling method.*

3.2 Main Idea

As was mentioned in the previous section, a genetic algorithm is used on top of a global scheduling heuristic. This has an important consequence. The global scheduling heuristic does not have to consider the entire realistic hardware model explicitly when it has to make scheduling decisions. For it is expected that the genetic algorithm is able to implicitly optimize for features of the architecture that are not considered by the global heuristic. The main purpose of the global heuristic is to help the genetic algorithm to search the solution space fast and efficiently.

The fact that the global heuristic is not based on a complete and realistic multiprocessor model has a number of advantages. Two advantages will be mentioned here. Firstly, because the model is very general, the global heuristic can be used for a wide range of multiprocessor architectures. A second advantage is that the global scheduling heuristic does not have to be too complex, which implies that it is easier to design a good heuristic.

A detailed and realistic model of the multiprocessor is only used in the black-box heuristic. This has the advantage that in order to support new multiprocessor architectures, only the black-box heuristic has to be extended[1]. The black-box heuristic schedules one operation at a time and uses a greedy heuristic to do so. This has the advantage that the heuristic can be fast and does not need to be too complex. Because the genetic algorithm can also influence the

[1]Object-oriented programming is expected to be very suitable to allow for extensions and changes of the black-box heuristic in order to support a wide variety of multiprocessor architectures.

scheduling results of the black-box heuristic, it is expected that this local approach can still give good results.

3.3 The Global Scheduling Heuristic

A detailed description of the global scheduling heuristic is given in Chapter 4. Some features of the scheduling heuristic are now mentioned briefly. Firstly, the order in which operations are scheduled is determined by a permutation of the operations. This allows the use of a genetic algorithm which will search for the optimal permutation. Secondly, the heuristic is able to exploit inter-iteration parallelism because the schedule is folded to a single iteration period and operations from other iterations are considered. Thirdly, the heuristic considers the effects of precedence constraints between the operations. This ensures that operations are never scheduled such that direct or indirect precedence constraints are violated. However, it can not be guaranteed that there are always enough free time slots in the schedule left when an operation has to be scheduled. A fourth feature of the heuristic overcomes this problem. Empty cycles can be inserted in an intermediate schedule.

It may happen that the quality of the schedule deteriorates when extra cycles are inserted during scheduling, in particular when cycles are inserted at a final stage of the scheduling heuristic. However, inserting new cycles ensures that the heuristic always produces a feasible schedule. This characteristic has a positive effect on the efficiency of the genetic algorithm. Furthermore, an initial iteration period can be given to the scheduling heuristic. When this initial iteration period is chosen larger than IPB, but still lower or equal to the practical minimum iteration period for the given scheduling problem[2], a possible deterioration of the quality of the schedules can be reduced. The genetic algorithm must then provide the initial iteration period.

3.4 The Black-Box Scheduling Heuristic

The global heuristic assigns an operation to an FU and chooses a certain start time. In doing so, it takes precedence relations and minimal communication delays into account. The global heuristic, however, ignores any additional restrictions imposed by the multiprocessor architecture. To ensure that these restrictions are also considered, the global heuristic calls the black-box heuristic, which uses an accurate multiprocessor model. The black-box heuristic completes the scheduling of the operation. It for instance schedules all communications between the current operation and its predecessor and successor operations that have already been scheduled.

The black-box heuristic must abide by the way the operation is scheduled by the global heuristic. It can not schedule the operation on a different FU when that turns out to be more convenient. The black box heuristic is however allowed to alter its own internal and more

[2]The minimum iteration period for a given scheduling problem can be higher than the theoretical iteration period bound IPB, for the calculation of IPB is merely based on simple characteristics of the DFG and the hardware configuration. Chapter 5 gives an example of a scheduling problem where $T_0 = IPB$ can not be met. Chapter 8 presents much more examples.

detailed schedule. This may be necessary in order to guarantee that the operation can be scheduled. Whether this may happen depends on the multiprocessor model that the black-box supports.

The black-box heuristic must *always* be able to schedule the operation and the resulting schedule should also be valid of course. When the black-box heuristic would not be allowed to insert cycles in the schedule, this can not be guaranteed. Therefore the black-box heuristic is also allowed to insert cycles in the schedule. This may for example be required when there is contention in the communication network. When it has finished scheduling an operation, it returns the number of cycles that it inserted in the schedule to the global heuristic. The global heuristic can then determine whether to schedule the operation at the given position in the schedule (start time and FU) or to continue searching to see if there is another (better) possibility to schedule the current operation.

Chapter 5 discusses the black-box heuristic in more detail.

4

The Global Scheduling Heuristic

The global scheduling heuristic is described in this chapter. First, Section 4.1 explains the main ideas behind the global heuristic. After that, the algorithm of the heuristic is given in Section 4.2. Some characteristics of the algorithm are discussed in more detail in Section 4.3. Finally, Section 4.4 illustrates how the global heuristic functions by means of a detailed example.

4.1 Main Ideas

The operations are scheduled one by one. For every operation a start time and a FU are chosen. No backtracking is used, so once an operation has been scheduled, its position in the schedule will not change. This has the advantage that the execution speed of the global scheduling heuristic is fast.

The scheduling method requires a special representation of time because cycles can be inserted in the schedule. Section 4.1.1 explains how times are represented efficiently. Distance matrices are used to ensure that precedence relations are not violated and all the scheduling ranges for the still unscheduled operations remain non-empty. Section 4.1.2 discusses this in detail. However, these distance matrices cannot guarantee that operations can always be scheduled. Why this is the case, is explained in Section 4.1.3. Nevertheless, it is desirable that the heuristic will always produce a valid schedule. To make this possible, the heuristic is able to insert extra cycles in the schedule. Section 4.1.4 discusses the insertion of cycles in detail.

4.1.1 Representation of Time

Times need to be represented and manipulated by the scheduling method. An obvious example is that the start time of every operation needs to be stored.

The fact that cycles can be inserted in the schedule by the scheduling method, has its effect on how times can be represented efficiently. A one-dimensional time scale can be used:

$$time = column\_no + schedule\_no \times T_0 \tag{4.1}$$

Where *column_no* is the ranking number of the column in the schedule. Columns in the schedule are numbered consecutively from left to right, starting at 0 (see Figure 4.1). The value *schedule_no* is a ranking number indicating which T_0-sized interval or schedule the time lies in. The numbering is such that the time scale is a linear sequence of integer numbers. It is not recommended to use this time scale to store times because the iteration period T_0 changes when a cycle is inserted in the schedule. This means that every time variable that is stored by the scheduling method needs to be updated to reflect this change.

It is therefore useful to consider a two-dimensional representation of time. For instance the following one where time is represented by a pair of values:

$$time = T_r(column\_no, schedule\_no) \tag{4.2}$$

This representation of time is called *time pair with a relative column index* and the identifier T_r is used to indicate that this representation is used.

The representation has the advantage that when the iteration period changes, all times remain valid. However, when cycles are inserted in the schedule, it can happen that new columns are inserted in front of existing columns. So although (the start of) each operation stays fixed to the same column, the *column_no* of that column can change. Which means that it is still necessary to update the times that are stored by the scheduling method.

So a further improved representation of time is:

$$time = T_s(column\_id, schedule\_no) \tag{4.3}$$

The representation is called *time pair with symbolic column index* and it is identified by T_s. The index of a column is symbolic because *column_id* is a symbol that uniquely identifies a column in the schedule. It does not change when new columns are inserted in the schedule. This has the advantage that after insertion of cycles all time variables stored by the scheduling method remain valid.

A disadvantage is that it is more complicated to manipulate times, for instance to increase a time by one time unit or to compute the difference of two times. However, when object-oriented programming is used, it is very easy to separate the additional time-manipulation code from the actual code of the scheduling algorithm. Furthermore, for the sake of efficiency it is then easy to support all three time representations and make it possible to switch between them. For instance, a variable of time that is actively used and for which it is known that during its use no cycles will be inserted in the schedule, can more efficiently be represented by a time pair with a relative column index.

Figure 4.1 and Table 4.1 illustrate these three different ways to represent times. The figure shows a schedule before and after a cycle has been inserted in front of the second column. The table shows how this affects the start time of each of the three operations.

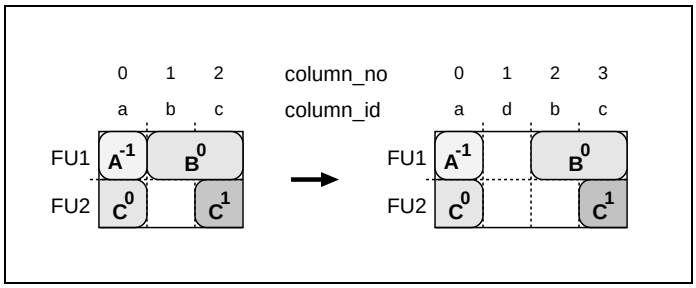

Figure 4.1: *A cycle is inserted before the second column.*

	start time (before)				start time (after)		
A	$T_s(a,1)$	$= T_r(0,1)$	$= 3$	$\rightarrow$	$T_s(a,1)$	$= T_r(0,1)$	$= 4$
B	$T_s(b,0)$	$= T_r(1,0)$	$= 1$	$\rightarrow$	$T_s(b,0)$	$= T_r(2,0)$	$= 2$
C	$T_s(c,-1)$	$= T_r(2,-1)$	$= -1$	$\rightarrow$	$T_s(c,-1)$	$= T_r(3,-1)$	$= -1$

Table 4.1: *The start times of the operations before and after an extra cycle was inserted (see Figure 4.1).*

4.1.2 Precedence Relations

When an operation is scheduled, the precedence relations between the operations as specified by the DFG, are taken into account. Not only will the heuristic make sure that the precedence relations for the operations that have already been scheduled are not violated, it also respects the precedence relations for the operations that have not yet been scheduled. It makes sure that their scheduling ranges will all remain nonempty. This ensures that when they have to be scheduled, a start time can be found that does not violate any precedence constraints either.

Operation Distance Matrix

In order not to violate precedence constraints, the heuristic makes use of an *operation distance matrix* $\mathbf{D}_c^{T_0}$. This matrix specifies the maximum path lengths between every pair of operations (in both [HdG90] and [Hei96] such a matrix is used, although under different names). The length of a path is constructed by summing all the durations of the operations in the path. These lengths can be negative when the path contains delay nodes, because they contribute $-T_0 \times d.multiplicity$ to the path. When there is no connection between two nodes, the distance will be set to $-\infty$. Because the distance matrix depends on the iteration period, and the latter can change during the run of the heuristic, several distance matrices (one for every allowed iteration period) are required: $\mathbf{D}_c^{T_0}[c_1, c_2]$, with $T_0 \in [T_{0,min}, T_{0,max}]$ and $c_1, c_2 \in C$. Where $T_{0,min}$ and $T_{0,max}$ are respectively the minimum and maximum values that can occur during a run of the global heuristic. $T_{0,min}$ is generally set to IPB. The value of $T_{0,max}$ can be an upper limit specified by the user. An alternative implementation is to adjust it dynamically throughout the run of the heuristic algorithm.

$\mathbf{D}_c^{T_0}[c_1, c_2] = d$ means that with an iteration period T_0, the start time of operation c_2 must be at least d time units later than the start time of operation c_1. When d is negative, $-d$ gives an upper limit on the number of time units that operation c_2 can be scheduled earlier than operation c_1. Note that d does not give an upper limit on the number of time units that

operation c_2 can be scheduled later than operation c_1. This limit is given by $-\mathbf{D}_c^{T_0}[c_2, c_1]$.

As an example, consider once again the second-order filter given in Figure 2.1. When $T_0 = 3\,\text{TU}$, a multiplication requires 2 TU and an addition 1 TU, the operation distance matrix is as follows (all values are in TU):

$$\mathbf{D}_c^{T_0} = \begin{bmatrix} 0 & 1 & -4 & -1 & 2 & 1 & -1 & -4 \\ -3 & 0 & -5 & -2 & 1 & 0 & -2 & -5 \\ 2 & 3 & 0 & 1 & 4 & 3 & 1 & -2 \\ -1 & 2 & -3 & 0 & 3 & 2 & 0 & -3 \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 1 & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 3 & 2 & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & 3 & 2 & -\infty & 0 \end{bmatrix}$$

The calculation of a distance matrix can be done in $O(N_c^3)$ by using the Floyd-Warshall algorithm [HdG90].

Valid Start Times Ignoring Communication Delays

When an operation has to be scheduled, the distance matrix $\mathbf{D}_c^{T_0}$ is used to calculate ranges of valid start times for the operation. Every range is specified by a minimum start time and a maximum start time.

First the range of start times is calculated when communication delays are ignored:

$$R_{nc,min}(c_{cur}, T_0, S_c) = \max_{c \in S_c} \Big(c.starttime + \mathbf{D}_c^{T_0}[c, c_{cur}] \Big) \tag{4.4}$$

$$R_{nc,max}(c_{cur}, T_0, S_c) = \min_{c \in S_c} \Big(c.starttime - \mathbf{D}_c^{T_0}[c_{cur}, c] \Big) \tag{4.5}$$

Here c_{cur} is the current operation to be scheduled and S_c is the set of scheduled operations. These two range limits will be called $R_{nc,min}$ and $R_{nc,max}$ respectively; the parameters will be omitted for the sake of convenience. It should not be any source for confusion, for all three parameters are part of the state of the scheduling method and therefore have a fixed and known value at any given moment. So whenever the parameters are missing, read for them the values as given by the current state of the scheduling method. Another convention is that the lower and upper limit make up a range, which is referred to as R_{nc}.

When the initial iteration period was chosen sufficiently large, and previous operations were scheduled within their range R_{nc}, the range R_{nc} will always be nonempty.

Valid Start Times Including Communication Delays

The global scheduling heuristic also calculates the valid start times for an operation when communication delays are not ignored. The length of the communication delays depends on

the FU to which the operation is assigned. Therefore more than one range is calculated. Namely one for every FU:

$$R_{c,min}(c_{cur}, fu, T_0, S_c) = \max_{c \in S_c} \Big(c.starttime + \mathbf{D}_c^{T_0}[c, c_{cur}] + \mathbf{D}_h[c.fu, fu] \Big) \tag{4.6}$$

$$R_{c,max}(c_{cur}, fu, T_0, S_c) = \min_{c \in S_c} \Big(c.starttime - \mathbf{D}_c^{T_0}[c_{cur}, c] - \mathbf{D}_h[(fu, c.fu] \Big) \tag{4.7}$$

Here c_{cur} is the current operation to be scheduled and fu the FU for operation c_{cur}. S_c is again the set of scheduled operations. $\mathbf{D}_h$is the hardware distance matrix as defined in Section 2.4. These range limits will be called $R_{c,min}(fu)$ and $R_{c,max}(fu)$ respectively and the range they define $R_c(fu)$, with $fu \in F$. The entire set of these ranges will be called R_c.

Even when the ranges R_c are constantly obeyed when operations are scheduled, it is possible that when a new operation is scheduled, $R_{c,min}(fu)$ is larger than $R_{c,max}(fu)$ for a given fu. In other words, the range $R_c(fu)$ can be empty. This can happen because the communication delays are not included in the operation distance matrix. They can not be included because it would require that all operations are assigned to a FU beforehand.

It can even occur that *all* ranges $R_c(fu), fu \in F$, are empty. This is illustrated in the following example.

Example

To see how distance matrices are used to calculate the ranges of valid start times, consider the following example. The data-flow graph and an intermediate schedule are shown in Figure 4.2. The multiprocessor configuration consists of only two FUs. The hardware distance matrix is as follows:

$$\mathbf{D}_h = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

It can be seen that it takes 1 TU to transfer data from one FU to the other.

The operation distance matrix when $T_0 = 3\,\mathrm{TU}$ can be derived from the DFG:

$$\mathbf{D}_c^{T_0} = \begin{bmatrix} 0 & -\infty & 1 & 2 \\ -\infty & 0 & 1 & 2 \\ -\infty & -\infty & 0 & 1 \\ -\infty & -\infty & -\infty & 0 \end{bmatrix}$$

The table in the figure gives the values of all the time ranges for operation C. To see how these values can be calculated, consider the following calculation:

$$\begin{aligned} &R_{c,min}(\mathrm{C}, \mathrm{FU1}, 3\,\mathrm{TU}, \{\mathrm{A}, \mathrm{B}, \mathrm{D}\}) \\ &\quad = \max(T_r(0,0) + 1 + 0, T_r(0,0) + 1 + 1, T_r(2,0) - \infty + 0) = T_r(2,0) \end{aligned}$$

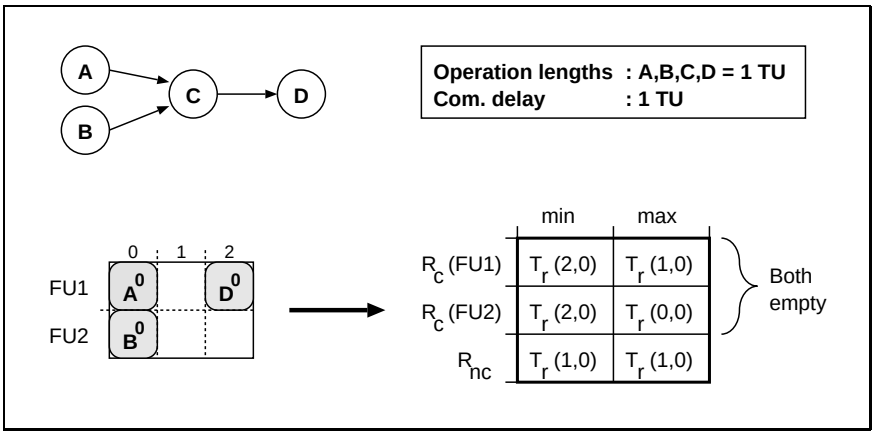

Figure 4.2: *Ranges of valid start times for operation C. Note that all ranges R_c are empty.*

The calculation of the other bounds of the time ranges is similar (using Equations 4.4 to 4.7). As can be seen, both ranges R_c are empty, whereas the range R_c is not. □

4.1.3 Schedule Instants

Even when the heuristic uses the distance matrix $\mathbf{D}_c^{T_0}$ every time it schedules an operation, it can happen that given a partial schedule a still unscheduled operation can not be scheduled. The first reason has already been given at the end of the previous section. The operation distance matrix does not include communication delays. It is therefore possible that given a partial schedule, the communication delays for an operation can not be satisfied (which is the case when all the ranges $R_c(fu), fu \in F$ are empty).

A second reason is that the scheduling method is resource constrained. It can therefore happen that given a partial schedule, an operation can not be scheduled because the resource it needs (a FU to be executed on[1]) is not available. Inserting cycles in the schedule can be used to solve the problem (this is explained in Section 4.1.4).

To describe how the heuristic algorithm works, it is useful to distinguish "base schedule instants" and "valid schedule instants". These terms are explained below.

Given a partial schedule and a still unscheduled operation, a *base schedule instant* is a proposed way to schedule the operation. It contains a start time for the operation and a FU to which it is assigned. However, it does not necessarily have to be completely specified. Allocation of other hardware resources (e.g. communication links) may be excluded. Nor does the base schedule instant have to represent a possible or valid way to schedule the operation. For it is never used directly to schedule an operation, instead it is used to construct a valid schedule instant.

A *valid schedule instant* specifies a valid way to schedule the operation given the precedence relations, the hardware model and the current allocations of the hardware.

[1]When the black-box heuristic is added to the scheduling method, additional resources are taken into consideration, as for instance communication links.

For the global heuristic, a base schedule instant is given by a FU fu and a start time t for an operation. fu must be a FU that can handle the type of operation. The start time is within the range R_{nc}. The latter ensures that precedence relations are satisfied. It is possible that minimum communication delays are not satisfied or that the FU is not free at the specified start time. Therefore, a base schedule instant is always converted to a valid schedule instant. If the base schedule instant is not yet valid, extra cycles are inserted in the schedule. A valid schedule instant in the heuristic therefore includes a specification of the number of cycles to insert n_{insert}, and the column in the schedule col_{insert} where to insert these cycles.

One of the reasons that base schedule instants are considered next to valid schedule instants is that they provide a systematic and straightforward manner to generate all valid schedule instants that are sensible. With sensible it is meant, that valid schedule instants that are known not to be optimal beforehand (for instance, because they insert more cycles in the schedule than is necessary to remedy the constraints that are violated) do not have to be generated.

Furthermore, the same approach is used later on when the global heuristic is connected to the black-box heuristic. Valid schedule instants for the global heuristic are then used as base schedule instants for the black-box heuristic. The black-box heuristic then converts those to valid schedule instants with respect to the more detailed hardware model.

4.1.4 Inserting New Cycles

Inserting an extra cycle in the schedule creates for every resource a new free time unit in the schedule. Therefore it can be used when there was not a resource available for the operation. Furthermore, inserting extra cycles in the schedule can increase the time gap between two operations. So, inserting extra cycles in the schedule can also be used when communication delays are not yet satisfied. Notice that a side effect is an increase of the iteration period.

When new cycles are inserted in the schedule, the number of extra cycles n_{insert} and the column where to insert these cycles col_{insert} have to be specified. The cycles are inserted immediately before col_{insert}. Because the scheduling is nonpreemptive, after a cycle is inserted in the schedule every operation should still execute uninterruptedly and be allocated to a continuous series of cycles in the schedule. There are more ways to accomplish this. The proposed way is to move operations such that every operation still starts in the same column as it did before the cycles were inserted. This is illustrated in Figure 4.3 (a).

However, this is not the only way to insert cycles in the schedule. It can be seen that as long as one part of every operation executes at the same column as it did before insertion of cycles in the schedule, the resulting schedule will be valid. With valid it is meant that the precedence relations between the operations are obeyed. As long as the precedence relations are satisfied before cycles are inserted in the schedule, they will also be satisfied afterwards. Furthermore, the scheduling ranges for the operations that have not yet been scheduled are not affected negatively. The size of these ranges stays equal or increases when new cycles are inserted in the schedule. Therefore a different approach to insert cycles is to fix the end time of every operation. This way of inserting cycles is illustrated in Figure 4.3 (b).

The reason that it was decided to fix the start times of every operation is that it is more "natural" to implement it efficiently. For the second way to insert cycles requires storing the

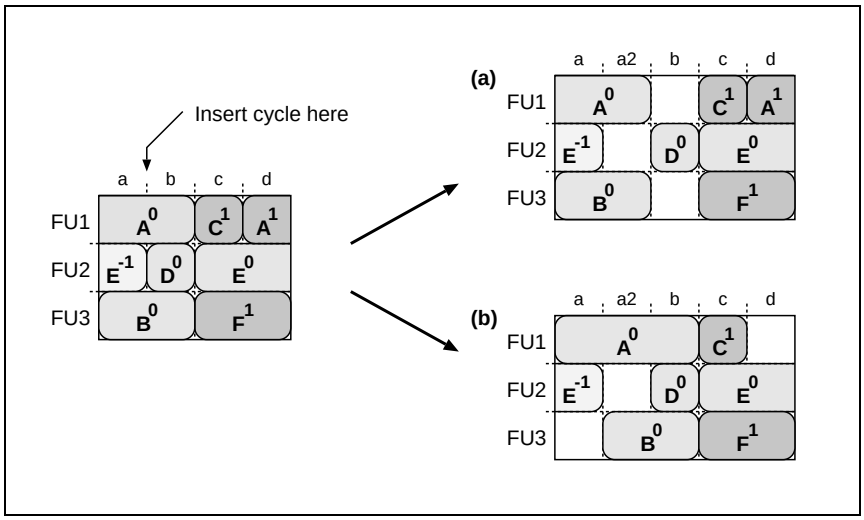

Figure 4.3: *Two different ways to insert cycles in a schedule. A single cycle is inserted before column 2. (a) The proposed way to insert cycles, and (b) another way to insert cycles.*

time an operation *ends* its execution instead of when it *starts*, in order to be implemented efficiently. Refer to Section 4.1.1 to recall how time is represented.

4.2 Algorithm

The algorithm can be summarized as follows:

1. Provide P, an ordered list of all the operations that have to be scheduled, and $T_{0,initial}$, the initial iteration period; set: $T_0 = T_{0,initial}$.

2. Retrieve (and remove) the first operation from the list P. Schedule this operation on the first FU that can execute this operation and set its start time to $T_r(0,0)$.

3. Choose c, the next operation to schedule, as follows: find the first operation from the list P for which at least one direct predecessor or successor operation has already been scheduled. Remove this operation from the list P.

4. Calculate the valid start times for operation c. Use the distance matrix $\mathbf{D}_c^{T_0}$, the start times of the operations that have already been scheduled, and the distance matrix $\mathbf{D}_h$ to calculate R_{nc} and $R_c(fu), fu \in F$.

5. Set $n_{tot,best} = \infty$. Set the preferred start time t_{pref} and the set of base schedule instants B as follows:

   if the range R_{nc} is unbounded then
   if the upper limit of R_{nc} is unbounded then
   set $t_{pref} = R_{nc,min}$
   $B = \{(fu, t_s) | c.type \in fu.optypes \wedge t_s \geq R_{c,min}(fu) \wedge t_s < R_{c,min}(fu) + T_0\}$
   else set $t_{pref} = R_{nc,max}$

$$B = \{(fu, t_s) | c.type \in fu.optypes \wedge t_s \leq R_{c,max}(fu) \wedge t_s > R_{c,max}(fu) - T_0\}$$

else set t_{pref} = get_pref_time_bounded(c, R_{nc}) (See Section 4.3.4)

$$B = \{(fu, t_s) | c.type \in fu.optypes \wedge t_s \geq R_{nc,min} \wedge t_s \leq R_{nc,max}\}$$

Where the base schedule elements in B are ordered by an increasing distance between t_s and t_{pref}. For equal distances, the order is determined by the FU numbering.

6. Retrieve and remove the first base schedule instant from B. Use it to construct a valid schedule instant. The start time t_s and the number of cycles that has to be inserted in the schedule, n_{insert}, are set such that:
   - The communication delays with the scheduled predecessor operations are satisfied.
   - The operation fits in the schedule.
   - The communication delays with the scheduled successor operations are satisfied.
7. Set $n_{tot} = n_{insert}$.
8. If $n_{tot} < n_{tot,best}$ then set s_{best} to the current valid schedule instant and set $n_{tot,best} = n_{tot}$.
9. If B is not empty and $n_{tot,best} > 0$ go to Step 6.
10. Schedule the operation as specified by the valid schedule instant s_{best} and set $T_0 = T_0 + n_{tot,best}$.
11. When the list P is not yet empty, go to Step 3.

4.3 Notes

This section examines several aspects of the global heuristic in more detail. Section 4.3.1 discusses the selection procedure that chooses the next operation that is scheduled. Section 4.3.2 explains the way that the ranges of valid start times are used. After that, Section 4.3.3 discusses the search for valid schedule instants. Section 4.3.4 looks at the preferred start time of the operations. Finally, Section 4.3.5 gives an idea how the global heuristic determines the number of cycles that has to be inserted to construct a valid schedule instant.

4.3.1 Choosing an Operation

When the next operation that is to be scheduled is chosen, the list of operations P is used. However, the heuristic does not always pick the first operation from the list of remaining operations. Instead it requires that at least one direct predecessor or successor operation has been scheduled. This requirement is added to improve the quality of the schedules that are found by the heuristic. The approach was already suggested by [Hei96] and it is supported by empirical data in Section 7.2.2.

4.3.2 Ranges of Valid Start Times

The ranges of valid start times are used to find a valid schedule instant. For every operation that has to be scheduled, a schedule instant has to be found, so that:

- The precedence relations and minimum communication delays with respect to the operations that are already scheduled, are satisfied.
- For all operations that are not yet scheduled, the scheduling ranges (without considering communication delays) will remain non-empty.

To fulfill the first requirement, the range $R_c(fu)$, with fu indicating the FU that executes the operation, is used. The second requirement can be satisfied by considering the range R_{nc}. However, since $R_c(fu)$ is more restrictive than R_{nc} the former can be used too.

The latter raises the question why also the range R_{nc} is calculated. That is because it is used to construct the set of base schedule instants B. The next section explains why. However, when the set B is used to construct the valid schedule instants, $R_c(fu)$ is still used to make sure that the communication delays are satisfied.

4.3.3 Search for Valid Schedule Instants

Two remarks can be made about the search for a valid schedule instant. The first remark is restricted to operations for which R_{nc} is bounded. *Base schedule instants can be considered that have a start time that is outside the range* $R_c(fu)$. The reason is simple. For a given operation, all ranges $R_c(fu), fu \in F$ can be empty. So, when only base schedule instants would be considered that lie within $R_c(fu)$, it can happen that no valid schedule instant is found at all. This explains why the range R_{nc} is sometimes used to construct the set of base schedule instants B.

The second remark holds for both types of operations, those with a bounded or an unbounded range R_{nc}. *In the search for the best valid schedule instant, it is often not necessary to consider all possible base schedule instants.* The heuristic first considers the base schedule instants with a start time that is near to t_{pref}, and continues with those that are gradually further away. Therefore it knows it has found the best valid schedule instant as soon as no extra cycles need to be inserted ($n_{tot,best} = 0$).

4.3.4 Preferred Start Time

For operations with an unbounded range of start times, the preferred start time t_{pref} is set to the fixed limit of the range. Because these operations are not part of a cycle in the DFG, their start time does not directly affect the iteration period. However, they are of influence on the latency and the idea of choosing t_{pref} in this way, is to keep the latency minimal.

How t_{pref} is set for operations with a bounded range of start times (for which it is therefore known that they are part of one or more cycles in the DFG), is specified by the pseudocode

```
function get_pref_time_bounded(c, R_nc)
  { "Determine the preferred start time t_pref for operation c.
     R_nc is the range of valid start times of the operation and it
     is bounded." }
begin
  { "Global variables that are used:
     N_pred,s : The number of direct predecessor operations that
                have already been scheduled.
     N_succ,s : The number of direct successor operations that
                have already been scheduled.
     N_pred,t : The total number of direct predecessor operations.
     N_succ,t : The total number of direct successor operations." }

  if N_pred,s < N_succ,s
    t_pref = R_nc,max;
  else if N_pred,s > N_succ,s
    t_pref = R_nc,min;
  else if N_pred,t < N_succ,t
    t_pref = R_nc,min;
  else if N_pred,t > N_succ,t
    t_pref = R_nc,max;
  else
    t_pref = 0;

  return t_pref;
end
```

Figure 4.4: *The procedure "get_pref_time_bounded", which determines t_{pref} for operations that are part of a loop.*

in Figure 4.4. Section 7.2.3 presents empirical data that shows that the performance of the scheduling method is good when this algorithm is used.

4.3.5 How Many Cycles to Insert?

In Step 6 of the global heuristic it is necessary to determine n_{insert}, the number of cycles that has to be inserted in the schedule to schedule the operation c given the base schedule instant. The heart of the algorithm that calculates n_{insert} is the procedure *how_to_insert*.

The function *how_to_insert* is given in Figure 4.5. It determines how an operation can be inserted in the schedule given a start time and a FU. In the code *schedule* is an abstract object that can store a (partial) schedule. The function *col(t)* returns the relative column index of time t. The function *how_to_insert* returns how the operation can be scheduled such that:

- The operation fits in the schedule.

- If, for the given start time and FU, the operation did not violate any precedence relations or minimum communication delays, these will still not be violated.

The function always assigns the operation to the given FU. However, it can deviate from the start time or insert cycles in the schedule, when that is necessary. The function returns an offset *shift* that indicates how many time units the operation is shifted to the right with respect to the start time that was provided. It also returns the number of cycles N that has to be inserted in the schedule in front of the column corresponding to the initial start time. To illustrate what the function does, four examples are shown in Figure 4.6.

As can be seen in Figure 4.5, the function does not use the ranges R_{nc} and R_c nor any precedence relations specified by the DFG. A consequence is that sometimes more cycles are inserted than is strictly necessary. This can happen because *N1* does not always have to be set to *shift*. An example can be found in Figure 4.7. The proposed way to schedule the operation is shown in Schedule (a), which would be the right way when operation X is a direct predecessor of operation B. However, if there is no precedence relation between operation X and B, the cycle did not have to be inserted and the operation could be scheduled as shown in Schedule (b).

However, this has no effect on the results produced by the scheduling method. For there is then always a different start time that result in a minimum value for n_{insert}. For example, Schedule (b) in Figure 4.7 is generated by the function *how_to_insert* when the initial start time for operation X is set to $T_r(2,0)$. Note that when operation X is a direct predecessor of B, *how_to_insert* will never be called with start time $T_r(2,0)$, because then $R_{nc,max} = T_r(1,0) < T_r(2,0)$.

As is probably clear, the function *how_to_insert* can not be used on its own, but must be fit in an algorithm that takes the ranges R_{nc} and R_c into account. Two separate algorithm can be distinguished here, one that is used when the ranges are bounded and one when the ranges have an unbounded limit. Both algorithm are rather straightforward. In order not to complicate the discussion of the global heuristic any further, these will not given here.

4.4 Detailed Example

In order to understand better how the global heuristic works, a run of the heuristic is discussed in detail. The algorithm that is scheduled is the second-order digital filter that was already shown in Figure 2.1. The multiprocessor configuration consists of four FUs. The hardware distance matrix is as follows:

$$\mathbf{D}_h = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \\ 1 & 2 & 0 & 3 \\ 2 & 1 & 3 & 0 \end{bmatrix}$$

Note that the distance matrix can correspond to a multiprocessor structure in which the four FUs are connected in a chain structure as shown in Figure 4.8. A multiplication requires 2 TU and an addition 1 TU. All four FUs support multiplication and addition.

```
function get_num_free(schedule, fu, col);
  { "Returns the number of consecutive empty cycles in the schedule
    on FU fu, starting at column col." }

function col_delta(schedule, col1, col2);
  { "Returns the number of cycles that column col2 lies beyond
    column col1." }

function add_time(time, delta);
  { "Returns the time time increased by delta." }


function how_to_insert(schedule, c, fu, starttime)
begin
  { "Can operation c be scheduled at the proposed starttime or is it
    in the middle of another operation and should it be moved?" }
  c2 := operation_at(schedule, fu, col(starttime));
  if ( c2=EMPTY or col(c2.starttime)=col(starttime) )
    shift := 0;
  else
    shift := c2.duration -
             col_delta(schedule, col(starttime), col(c2.starttime));
  end;
  N1:= shift;
    { "Insert as many cycles as operation c is shifted, to avoid
      that any precedence relations might be violated." }

  { "Is there already enough room in the schedule or should extra
    cycles be inserted?" }
  F := get_num_free(schedule, fu, add_time(starttime, shift));
  N2 := max(0, c.duration - N1 - F);
  N := N1 + N2;

  { "The operation can be scheduled with a start time that is shift
    time cycles to the right of the proposed start time. N cycles
    have to be inserted before column col(starttime)." }
  return shift, N;
end
```

Figure 4.5: *The function "how_to_insert", which can be used to insert a new operation in a schedule.*

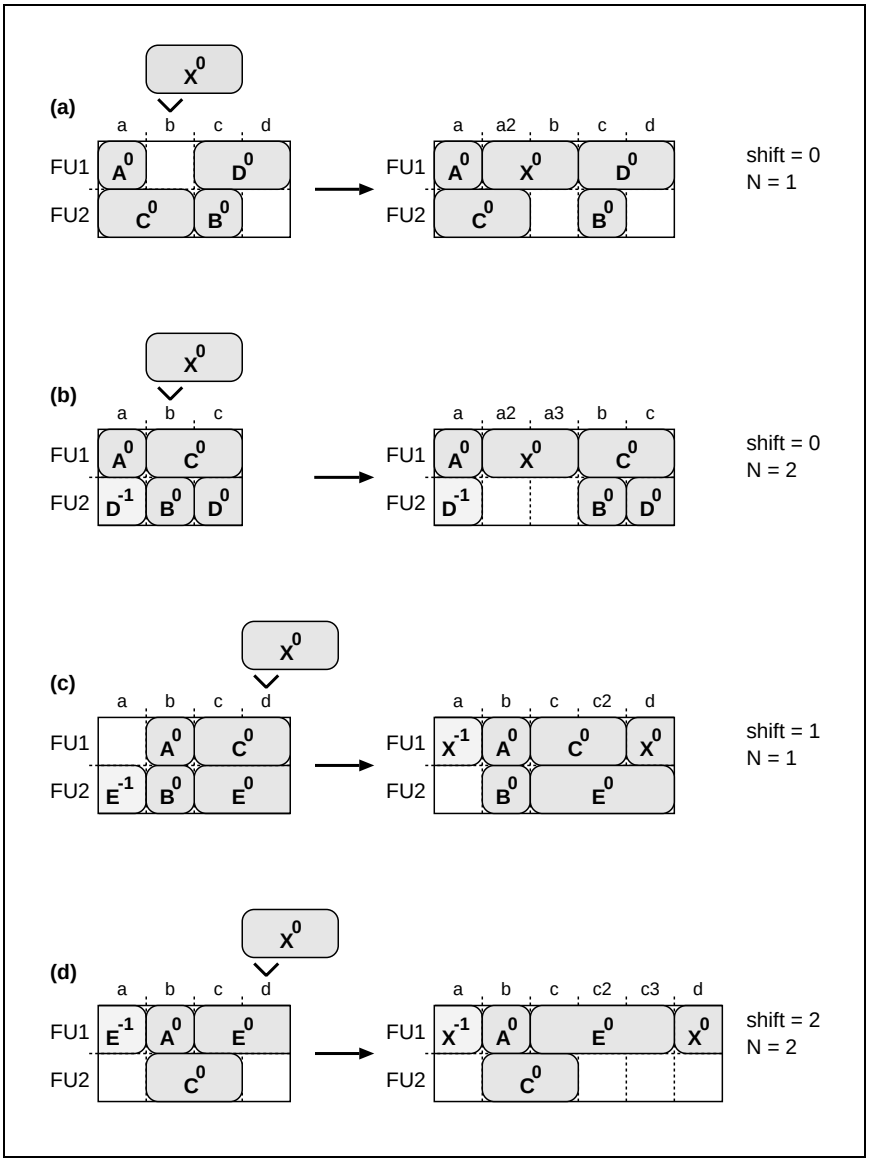

Figure 4.6: *Four examples of how an operation can be inserted in the schedule by the function "how_to_insert". In all four examples (a)-(d) operation X is assigned to FU1.*

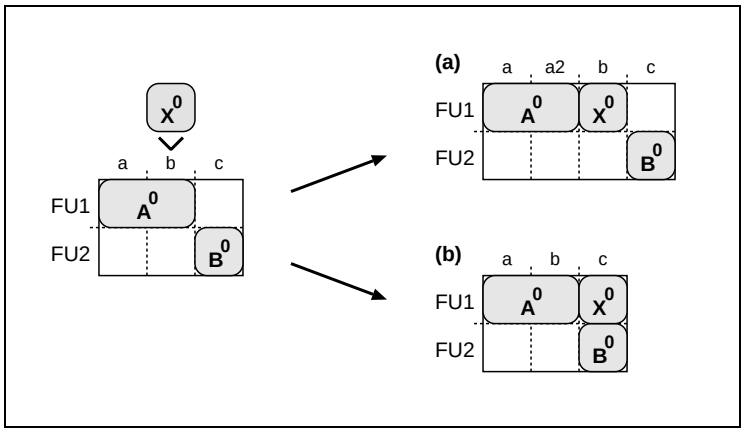

Figure 4.7: *Two ways to schedule operation X on FU1 when the proposed start time is $T_r(1,0)$: (a) the schedule suggested by the function "how_to_insert", (b) a valid schedule when operation B is not a direct successor of X.*

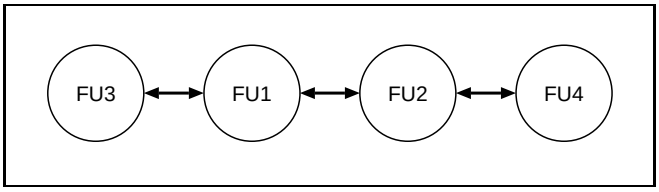

Figure 4.8: *A multiprocessor structure that complies with $\mathbf{D}_h$ given in Section 4.4.*

The permutation P that is provided to the global heuristic is $\{c_3, c_1, c_7, c_2, c_5, c_6, c_8, c_4\}$. Therefore, the first operation that is scheduled is c_3. It is assigned to FU1 and starts at time $T_r(0,0)$, as is shown in Figure 4.9 (a).

The next operation that is picked from the list P is c_1, a direct successor of operation c_3. The time ranges for the operation and the preferred start time can be found in Table 4.2. The table shows for every operation the values of some intermediate variables, when the operation is about to be scheduled.

The set of base schedule instants is:

$$\begin{aligned} B = \ \{ \ & (\text{FU1}, T_r(2,0)), (\text{FU2}, T_r(2,0)), (\text{FU3}, T_r(2,0)), (\text{FU4}, T_r(2,0)), \\ & (\text{FU1}, T_r(0,1)), (\text{FU2}, T_r(0,1)), (\text{FU3}, T_r(0,1)), (\text{FU4}, T_r(0,1)), \\ & (\text{FU1}, T_r(1,1)), (\text{FU2}, T_r(1,1)), (\text{FU3}, T_r(1,1)), (\text{FU4}, T_r(1,1)), \ \} \end{aligned}$$

The first base schedule instant that is used to construct a valid schedule instant does not require the insertion of a cycle in the schedule:

$$(\text{FU1}, T_r(2,0)) \rightarrow \text{Schedule on FU1 at } t = T_r(2,0)$$

It is immediately concluded that no better way to schedule the operation can be found. Therefore the operation c_1 is scheduled as given by the valid base schedule instant, see Figure 4.9 (b).

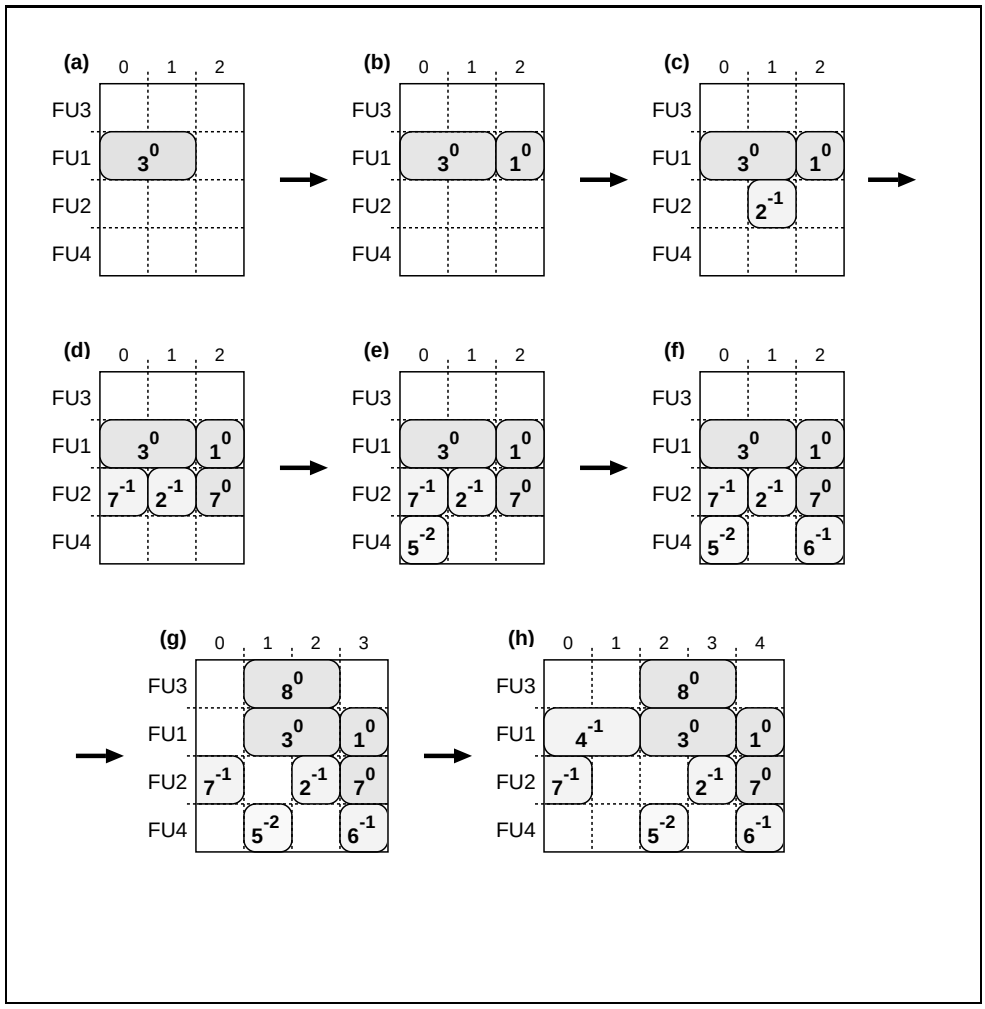

Figure 4.9: *Scheduling steps for the example run of the global scheduling heuristic.*

	R_{nc}	fu	$R_c(fu)$	t_{pref}
c_3	-		-	-
c_1	$[T_r(2,0), T_r(1,1)]$	FU1	$[T_r(2,0), T_r(1,1)]$	$T_r(2,0)$
		FU2	$[T_r(0,1), T_r(0,1)]$	
		FU3	$[T_r(0,1), T_r(0,1)]$	
		FU4	$[T_r(1,1), T_r(2,0)]$	
c_2	$[T_r(0,1), T_r(2,1)]$	FU1	$[T_r(0,1), T_r(2,1)]$	$T_r(0,1)$
		FU2	$[T_r(1,1), T_r(1,1)]$	
		FU3	$[T_r(1,1), T_r(1,1)]$	
		FU4	$[T_r(2,1), T_r(0,1)]$	
c_7	$[T_r(2,0), +\infty]$	FU1	$[T_r(0,1), +\infty]$	$T_r(2,0)$
		FU2	$[T_r(2,0), +\infty]$	
		FU3	$[T_r(1,1), +\infty]$	
		FU4	$[T_r(0,1), +\infty]$	
c_5	$[T_r(2,1), +\infty]$	FU1	$[T_r(0,2), +\infty]$	$T_r(2,1)$
		FU2	$[T_r(2,1), +\infty]$	
		FU3	$[T_r(1,2), +\infty]$	
		FU4	$[T_r(0,2), +\infty]$	
c_6	$[T_r(1,1), T_r(2,1)]$	FU1	$[T_r(2,1), T_r(0,1)]$	$T_r(2,1)$
		FU2	$[T_r(1,1), T_r(1,1)]$	
		FU3	$[T_r(0,2), T_r(2,0)]$	
		FU4	$[T_r(2,1), T_r(2,1)]$	
c_8	$[T_r(2,-1), T_r(0,1)]$	FU1	$[T_r(0,0), T_r(1,0)]$	$T_r(0,0)$
		FU2	$[T_r(2,-1), T_r(2,0)]$	
		FU3	$[T_r(1,0), T_r(0,0)]$	
		FU4	$[T_r(0,0), T_r(0,1)]$	
c_4	$[T_r(3,0), T_r(0,1)]$	FU1	$[T_r(0,1), T_r(3,0)]$	$T_r(3,0)$
		FU2	$[T_r(3,0), T_r(0,1)]$	
		FU3	$[T_r(1,1), T_r(2,0)]$	
		FU4	$[T_r(0,1), T_r(3,0)]$	

Table 4.2: *The time ranges and the preferred start time for the operations during the example run of the global heuristic.*

The list P, which now has the value $\{c_7, c_2, c_5, c_6, c_8, c_4\}$, is again used to select the next operation that is scheduled. The operation that is selected is not the first from the list but the second, c_2, because it is required that at least one immediate predecessor or successor has already been scheduled. Six valid schedule instants are constructed before one is found that does not require one or more cycles to be inserted:

$(\text{FU1}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on FU1 at $t = T_r(0,1)$
$(\text{FU2}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on FU2 at $t = T_r(1,1)$
$(\text{FU3}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on FU3 at $t = T_r(1,1)$
$(\text{FU4}, T_r(0,1)) \rightarrow$ Insert 2 cycles before column 0, schedule on FU4 at $t = T_r(2,1)$
$(\text{FU1}, T_r(1,1)) \rightarrow$ Insert 1 cycle before column 1, schedule on FU1 at $t = T_r(2,1)$
$(\text{FU2}, T_r(1,1)) \rightarrow$ Schedule on FU2 at $t = T_r(1,1)$

Operation c_2 is scheduled as specified by the last valid schedule instant, see Figure 4.9 (c).

The next operation that is scheduled is operation c_7. As can be seen in Table 4.2, the ranges of valid start times have an unbounded upper limit. The first base schedule instant corresponds to a valid schedule instant where no cycles have to be inserted:

$(\text{FU2}, T_r(2,0)) \rightarrow$ Schedule on FU2 at $t = T_r(2,0)$

The operation is scheduled as is specified by the valid schedule instant, see Figure 4.9 (d). Subsequently operation c_5 and c_6 are scheduled. The result can be found in respectively Figure 4.9 (e) and (f). Then operation c_8 is scheduled. Although there are 20 base schedule instants, none of them leads to a valid schedule instant that does not need to insert cycles in the schedule. This is understandable because, given the schedule in Figure 4.9 (f), there is no way to schedule the operation without inserting at least one cycle. The valid schedule instant that is chosen requires that one cycle is inserted in the schedule:

$(\text{FU3}, T_r(0,0)) \rightarrow$ Insert 1 cycle before column 0, schedule on FU3 at $t = T_r(1,0))$

The final operation that needs to be scheduled is c_4. It also can not be scheduled without inserting a cycle:

$(\text{FU1}, T_r(3,0)) \rightarrow$ Insert 3 cycles before column 3, schedule on FU1 at $t = T_r(4,0)$
$(\text{FU2}, T_r(3,0)) \rightarrow$ Insert 2 cycles before column 3, schedule on FU2 at $t = T_r(3,0)$
$(\text{FU3}, T_r(3,0)) \rightarrow$ Insert 3 cycles before column 3, schedule on FU3 at $t = T_r(5,0)$
$(\text{FU4}, T_r(3,0)) \rightarrow$ Insert 3 cycles before column 3, schedule on FU4 at $t = T_r(4,0)$
$(\text{FU1}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on FU1 at $t = T_r(0,1)$
$(\text{FU2}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on FU2 at $t = T_r(1,1)$
$(\text{FU3}, T_r(0,1)) \rightarrow$ Insert 3 cycles before column 0, schedule on FU3 at $t = T_r(1,1)$
$(\text{FU4}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on FU4 at $t = T_r(0,1)$

The first of the valid schedule instants that requires only one cycle to be inserted is chosen. The resulting schedule is shown in Figure 4.9 (h). The iteration period is 5 TU and the latency

is 9 TU. This is certainly not a very good schedule. It should be noted that the global heuristic can find the optimal schedule with an iteration period of 3 TU and a latency of 5 TU. However the permutation P was now chosen such that it illustrates clearly how the global heuristic functions.

5

The Black-Box Scheduling Heuristic

The black-box scheduling heuristic is presented in this chapter. Firstly Section 5.1 shows the change that is necessary in the global heuristic to include the black-box heuristic in the scheduling method. Section 5.2 then specifies the extended hardware model that is used by the black-box heuristic. After that, Section 5.3 gives a specification of the algorithm of the black-box heuristic. In Section 5.4 some characteristics of the algorithm are considered more closely. The black-box heuristic is illustrated in Section 5.5, where a run of the heuristic is discussed in detail. Finally, Section 5.6 gives an example of a run of the global heuristic in combination with the black-box heuristic.

5.1 Changes to the Global Heuristic

When a black-box algorithm is added to the scheduling method, the algorithm of the global heuristic given in Section 4.2 must be adjusted. Step 7 must be replaced by the following step.

7. When $n_{insert} \geq n_{tot,best}$ go to Step 9. Otherwise, use the valid heuristic schedule instant as a base black-box schedule instant to construct the corresponding valid black-box schedule instant:

   - Schedule the operation as specified by the valid schedule instant in the global heuristic. When cycles have to be inserted, the global heuristic notifies the black-box heuristic so that the detailed schedule maintained by the latter is updated correspondingly.
   - Call the black-box heuristic, provide as a parameter operation c.
   - The black-box heuristic completes the scheduling of operation c and returns the extra number of cycles, $n_{insert,bb}$, that it had to insert.
   - Unschedule the operation. First the black-box heuristic is called to make the scheduling undone. Then the global heuristic itself undoes the changes that it made.

   Set $n_{tot} = n_{insert} + n_{insert,bb}$.

5.2 Hardware Model

The black-box heuristic that has been implemented, supports the routing of data transfers. When data has to be transferred between two operations that do not reside on the same FU, the data is routed. FUs are connected by *communication links*. A communication link can be unidirectional or bidirectional. Every link can only be used for a single data transfer every time unit. It may even take longer than one time unit for a communication link to transfer the data. The time that is required by every link to transfer data is constant and is equal to δ. The transfer of data over a single communication link is called a *communication task*.

There does not have to be a communication link between every pair of FUs. Therefore data transfers may need to be routed over more than one communication link. Or in other words, scheduling a data transfer may require scheduling more than one communication task. Furthermore, two FUs can be connected by several paths of communication links. The one that is chosen is not fixed beforehand, it depends on the availability of the communication links at the moment the data transfer is scheduled.

5.3 Algorithm

The algorithm of the black-box heuristic that has been implemented to complete the scheduling of an operation is given below:

1. Provide c, the operation that has just been scheduled by the global heuristic.
2. Set $n_{insert,bb} = 0$.
3. Find all data transfers that have to be scheduled and place them in a list, T. The data transfers can be found as follows:

   Every direct predecessor or successor operation of c that has already been scheduled, and that does not execute on the same FU as c does, requires that a data transfer is scheduled.
4. Sort the list T so that the data transfers are ordered by increasing slack time. The slack time is given by:

$$\begin{aligned} t_{slack} &= c_{dest}.starttime - c_{source}.starttime - c_{source}.duration \\ &\quad + T_0 \times \sum_{d \in n(p)} d.multiplicity - \delta \times \mathbf{D}_h[c_{source}.fu, c_{dest}.fu] \end{aligned} \tag{5.1}$$

   Where c_{dest} is the operation that produces the data that is subsequently consumed by operation c_{source}. *n(p)* is the set of nodes in the path that connects c_{source} to $c_{destination}$ (the path can only contain delay nodes).
5. If T is empty, go to Step 15. Otherwise retrieve and remove the first data transfer t from T.

6. If $c = c_{dest}$ the data transfer will be scheduled from c_{source} to $c_{destination}$, otherwise from $c_{destination}$ to c_{source}[12].

7. Set t_{slack} equal to the slack time of the data transfer. Set $t_{min} = c_{source}.starttime + c_{source}.duration$. Set the set R so that it contains all paths with minimal length that connect $c_{source}.fu$ with $c_{dest}.fu$.

8. Set $n_{tot,best} = \infty$ and set $t_{pref} = t_{min}$. Set L so that it contains every communication link that is the first link of one or more of the paths in R. Use L to set $B = \{(l, t_s) | l \in L \wedge t_s \geq t_{min} \wedge t_s \leq t_{min} + t_{slack}\}$.

   The base schedule elements in B are ordered by an increasing distance between t_s and t_{pref}. For equal distances, the order is determined by the numbering of the communication links.

9. Retrieve and remove the first base schedule instant from B. Use it to construct a valid schedule instant for the communication task. Set the start time t_s and the number of cycles that has to be inserted in the schedule, n_{insert}, so that:

   - The communication task fits the in the link schedule.

10. If $n_{tot} < n_{tot,best}$ then set s_{best} to the current valid schedule instant and set $n_{tot,best} = n_{tot}$ and $t_{s,best} = t_s$.

11. If B is not empty and $n_{tot,best} > 0$ go to Step 6.

12. Schedule the communication task as specified by the schedule instant s_{best}. Set $n_{insert,bb} = n_{insert,bb} + n_{tot,best}$.

13. If the routing of the data transfer has been completed go to Step 5.

14. Update the slack time $t_{slack} = t_{slack} + t_{min} - t_{s,best}$. Set $t_{min} = t_{s,best} + \delta$. Update the list R by removing all paths that did not start with the communication link that was chosen. For the remaining paths, truncate them by removing the first link. Go to Step 8.

15. Return $n_{insert,bb}$, the total number of cycles that has been inserted.

5.4 Notes

It is useful to examine some aspects of the black-box heuristic in more detail. Section 5.4.1 discusses the direction in which the data transfers are scheduled. Section 5.4.2 explains that the way the FUs are numbered can effect the results of the scheduling method. Section 5.4.3 focuses on some similarities that exist between the global heuristic and the black-box heuristic.

[1]Please note that the direction in which the data transfer is scheduled is not necessarily the direction that the data will travel. The data always goes from c_{source} to $c_{destination}$.

[2]In the next steps, the data transfer is routed from c_{source} to $c_{destination}$ in order not to complicate the discussion of the algorithm.

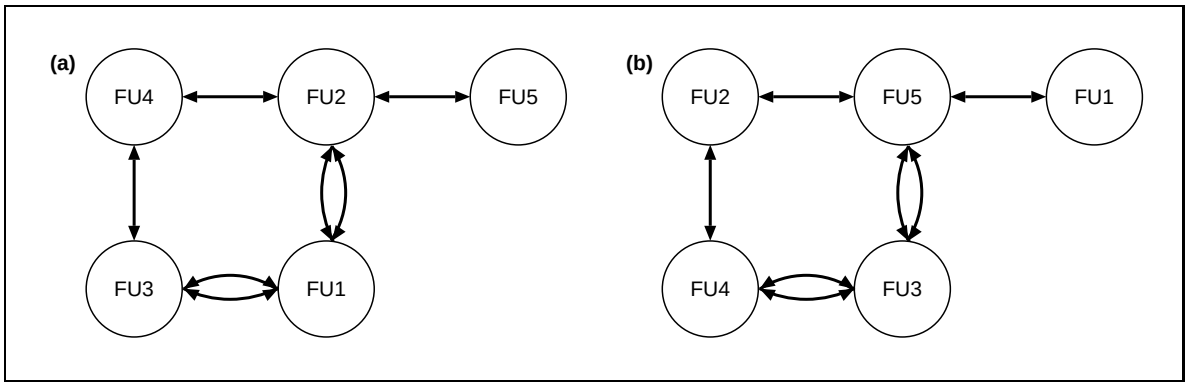

Figure 5.1: *(a) A good way to number the FUs, and (b) a bad way to number FUs.*

5.4.1 Scheduling Direction

When two operations that are directly connected in the DFG, are not assigned to the same FU a data transfer must be scheduled. As can be seen in Section 5.3, the scheduling of the data transfer does not always start at c_{source}. For if $c_{source} = c_{cur}$ the scheduling starts at $c_{destination}$.

The reason why this is done is because then, when no free communication link can be found a cycle will be inserted near to c_{cur}. Since all the data transfers that the black-box heuristic has to schedule in that run, start or end at c_{cur} this increases the chance that the newly inserted time slots can be used efficiently. Whether this way of choosing the scheduling direction is indeed better than to simply always schedule from c_{source} to $c_{destination}$ is examined in Section 7.3.1.

5.4.2 Numbering of FUs

In the specification of the hardware all FUs are assigned a unique number ranging from 1 to $|F|$. These numbers result in an ranking or ordering of the FUs. It is wise to choose this order carefully as it has a noticeable effect on the performance of the black-box heuristic. This will now be explained briefly.

The order of FUs determines how the global heuristic assigns the operations to the different FUs. The first operation that is scheduled is assigned to the first FU that can execute it (which is often FU1). After that, operations that are subsequently scheduled tend to be assigned to the same FU or surrounding FUs, with a preference for FUs with a low number. This becomes clear when you carefully study the global heuristic. The effect can be observed in the example run of the global heuristic that is shown in Figure 4.9.

A result is that FUs at the heart of the multiprocessor structure that have the largest communication link capacity should be ranked first. At the end of the ranking should be the FUs with the lowest link capacities. This encourages an efficient use of the communication network and tends to minimize the contention in the network. Figure 5.1 gives an example of a good and a bad way to rank the FUs.

It should be noted that the way the FUs are ordered, already has an effect in the global

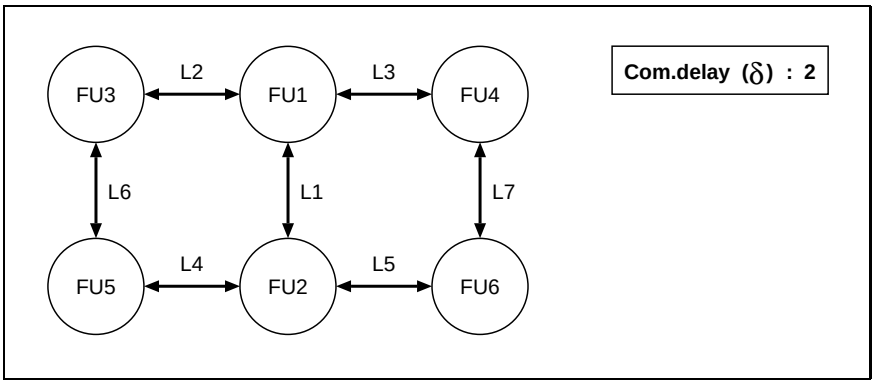

Figure 5.2: *The multiprocessor configuration used in the detailed example.*

scheduling heuristic when the black-box heuristic is not attached. For the distance matrix $\mathbf{D}_h$ can create differences between the "accessibility" of the FUs. The global heuristic does take this partly into account, because the distance matrix is used to calculate the time ranges R_c and these ranges are used when an operation is scheduled. However, there still is a tendency that operations are assigned first to the FUs with a low number.

The above is illustrated in Section 7.3.2. An experiment is presented where the order of the FUs is varied for a certain multiprocessor configuration.

5.4.3 Resemblance to the Global Heuristic

If you look at the global heuristic and the black-box heuristic, you can notice several similarities. One is that both use base schedule instants to construct valid schedule instants. The valid schedule instant that is chosen, is the one that required the least number of cycles to be inserted (and that had a start time that was closest to t_{pref}).

However, there is also a similarity in the schedules that they both produce. The global heuristic assigns operations to FUs and the black-box heuristic assigns communication tasks to communication links. This is however in essence the same: both assign tasks to resources. Furthermore, both insert cycles in the schedule, determine how many cycles must be inserted to schedule a task in the schedule, etc.

These similarities cause that it is reasonably simple to implement a new black-box heuristic (based on different hardware model). The global heuristic already provides a lot of functionality that can be reused straight away by the black-box heuristic.

5.5 Detailed Example

To understand clearly how the black-box heuristic functions, it is useful to look at a detailed example. The multiprocessor structure that is used is shown in Figure 5.2. The part of the DFG that is scheduled that is relevant for the example is shown in Figure 5.3.

The example starts when several operations and some data transfers have already been sched-

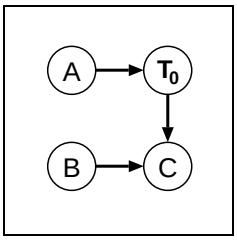

Figure 5.3: *The relevant part of the DFG for the detailed example.*

uled. The intermediate schedules are shown in Figure 5.4 (a)[3]. The top schedule shows how the relevant operations are scheduled (this schedule is maintained by the global heuristic). The bottom schedule shows the communication tasks that have already been scheduled (this schedule is only used by the black-box heuristic). Operation C has just been scheduled by the global heuristic. The black-box heuristic is then executed to complete the scheduling of the operation.

First, the black-box algorithm puts all data transfers in a list T, calculates the corresponding slack times and sorts the list by increasing slack time. This results in:

$$T = \{(\text{A(FU1)} \rightarrow \text{C(FU5)}; \text{slack time} = 1), (\text{B(FU4)} \rightarrow \text{C(FU5)}; \text{slack time} = 2)\}$$

Where the slack times are calculated as follows (see Equation 5.1):

$$\text{A}\rightarrow\text{C: slack time} = T_r(1,1) - T_r(3,0) - 1 + T_0 \times 1 - \delta \times 2 = 5 - 3 - 1 + 4 - 4 = 1$$
$$\text{B}\rightarrow\text{C: slack time} = T_r(1,1) - T_r(0,-1) - 1 + T_0 \times 0 - \delta \times 3 = 5 - (-4) - 1 + 0 - 6 = 2$$

The data transfer that is scheduled first is the one from A to C. The data will be scheduled from source to destination: from FU1 to FU5. The set of paths with minimal length is $R = \{(L1, L4), (L2, L6)\}$ and $t_{min} = T_r(0,1)$. This can also be found in Table 5.1 which shows the values of some variables throughout the run of the black-box heuristic. Every step corresponds to the scheduling of a single communication task.

To schedule the first communication task, four base schedule instants are considered before a valid schedule instant is found that does not require the insertion of a cycle:

$(\text{L1}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on L1 at $t = T_r(0,1)$
$(\text{L2}, T_r(0,1)) \rightarrow$ Insert 1 cycle before column 0, schedule on L2 at $t = T_r(1,1)$
$(\text{L1}, T_r(1,1)) \rightarrow$ Insert 2 cycles before column 1, schedule on L1 at $t = T_r(1,1)$
$(\text{L2}, T_r(1,1)) \rightarrow$ Schedule on L2 at $t = T_r(1,1)$

So, communication link L2 is chosen. The entire data transfer has not yet been scheduled, one communication task still needs to be scheduled. Therefore the set R, t_{slack} and t_{min} are updated (see Table 5.1, Step 2). Note that the slack time has been reduced by one.

[3]The intermediate state is chosen so that it best illustrates how the black-box heuristic functions. It is not an intermediate state that occurred during an actual run of the global scheduling heuristic.

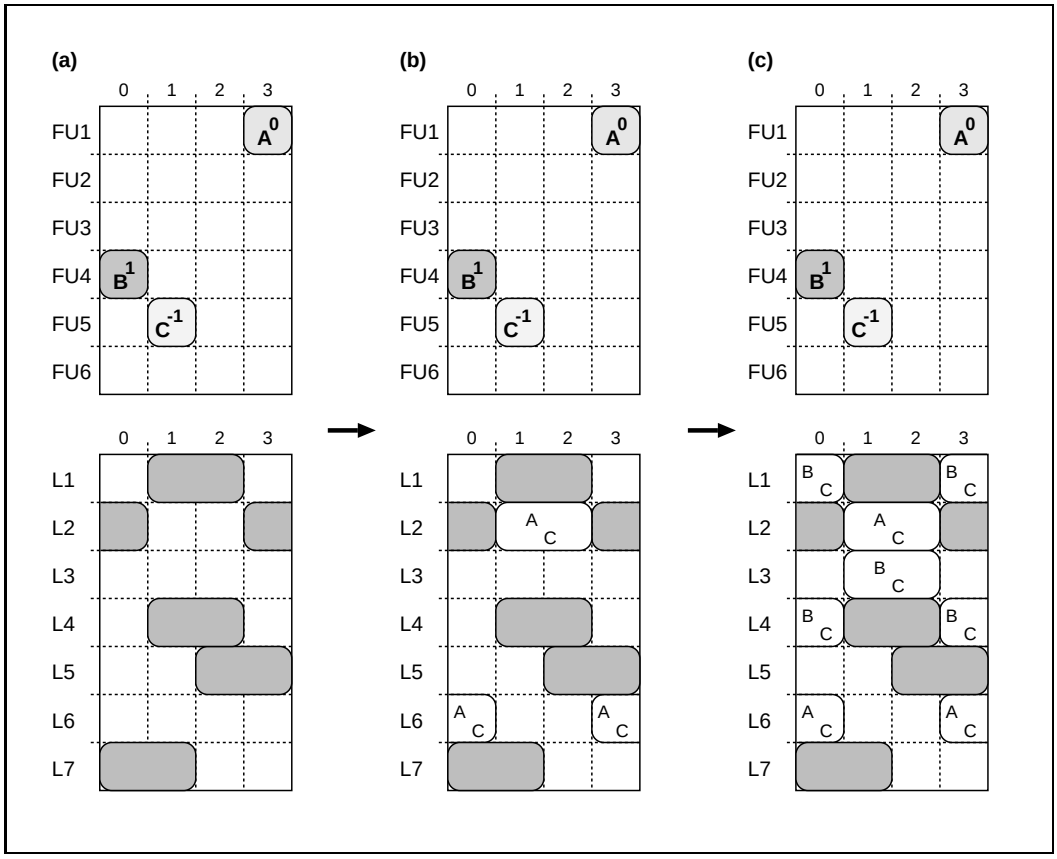

Figure 5.4: *Scheduling steps for the example run of the black-box heuristic.*

Step	t_{slack}	t_{min}	R	B
1	1 TU	$T_r(0,1)$	{(L1, L4), (L2, L6)}	{(L1, $T_r(0,1)$), (L2, $T_r(0,1)$), (L1, $T_r(1,1)$), (L2, $T_r(1,1)$)}
2	0 TU	$T_r(3,1)$	{(L6)}	{(L6, $T_r(3,1)$)}
3	2 TU	$T_r(1,-1)$	{(L3,L1,L4), (L3,L2,L6), (L7,L5,L4)}	{(L3, $T_r(1,-1)$), (L7, $T_r(1,-1)$), (L3, $T_r(2,-1)$), (L7, $T_r(2,-1)$), (L3, $T_r(3,-1)$), (L7, $T_r(3,-1)$)}
4	2 TU	$T_r(3,-1)$	{(L1,L4), (L2,L6)}	{(L1, $T_r(3,-1)$), (L2, $T_r(3,-1)$), (L1, $T_r(0,0)$), (L2, $T_r(0,0)$), (L1, $T_r(1,0)$), (L2, $T_r(1,0)$)}
5	2 TU	$T_r(1,0)$	{(L4)}	{(L4, $T_r(1,0)$), (L4, $T_r(2,0)$), (L4, $T_r(3,0)$)}

Table 5.1: *Intermediate values of some variables that are used by the black-box heuristic during the example run.*

A new set of base schedule instants is generated. It contains a single base schedule instant. No cycles need to be inserted in the schedule:

$$(\text{L6}, T_r(3,1)) \rightarrow \text{Schedule on L6 at } t = T_r(3,1)$$

The communication task is assigned to link L6 at the specified time. This completes the scheduling of the first data transfer. The resulting schedule can be found in Figure 5.4(b).

Next, the data transfer from B to C is scheduled. It requires that three communication tasks are scheduled. The results will be explained briefly. Refer to Table 5.1 for more details.

There are three possible communications paths with a minimal length, that connect FU4 to FU5, $R = \{(\text{L3,L1,L4}), (\text{L3,L2,L6}), (\text{L7,L5,L4})\}$. The first communication task is assigned to L3 for it was available at the prefered start time. So, two possible paths still remain. The link that is then chosen is L1. The scheduling of the data transfer ends by assigning the final communication task to L4. Because another communication task is already assigned to link L4, three base schedule instants are considered before a start time is found that does not require the insertion of extra cycles. The resulting schedule is shown in Figure 5.4(c).

This ends the run of the black-box heuristic. The heuristic returns the value $n_{insert,bb} = 0$ to indicate that it did not have to insert extra cycles in the schedule.

5.6 Example

This section will give a scheduling result that is produced by the global scheduling heuristic in combination with the black-box scheduling heuristic. The scheduling problem that is presented in this section, is chosen because it is expected to be a very good benchmark. One of the reasons is that a scheduling method will not be able to generate optimal schedules for this problem when it does not assign the operations to the different processors carefully.

The algorithm that must be scheduled is a well-known benchmark filter, namely the fifth-order wave digital filter that is shown in Figure 5.5. Figure 5.6 shows the multiprocessor structure on which the filter must be scheduled. The structure shows how many FUs there are, how they are connected and the set of operations supported by each FU. It also specifies the communication delay δ and the duration of the operations. It should be noted that, for sake of convenience, substraction is considered equal to addition in the multiprocessor specification.

Now that the data-flow graph and the duration of each of the operations are known, the theoretical lower bounds that were introduced in Chapter 2 can be calculated. There are two critical loops in the data-flow graph with a length of 16 TU (they are shown in Table 5.2). These two loops lead to an iteration period bound, $IPB = 16\,\text{TU}$. The corresponding minimal latency is $PDB = 14\,\text{TU}$ (it is determined by the I/O-path that is also given in Table 5.2). The minimum number of processors that is necessary to meet the iteration period bound is given by the processor bound, $PB = \lceil (26 \times 1\,\text{TU} + 8 \times 2\,\text{TU})/16\,\text{TU} \rceil = 3$.

A schedule for the given problem has been generated by using a genetic algorithm on top of the two heuristic algorithms. The main goal of the genetic algorithm was to minimize the iteration

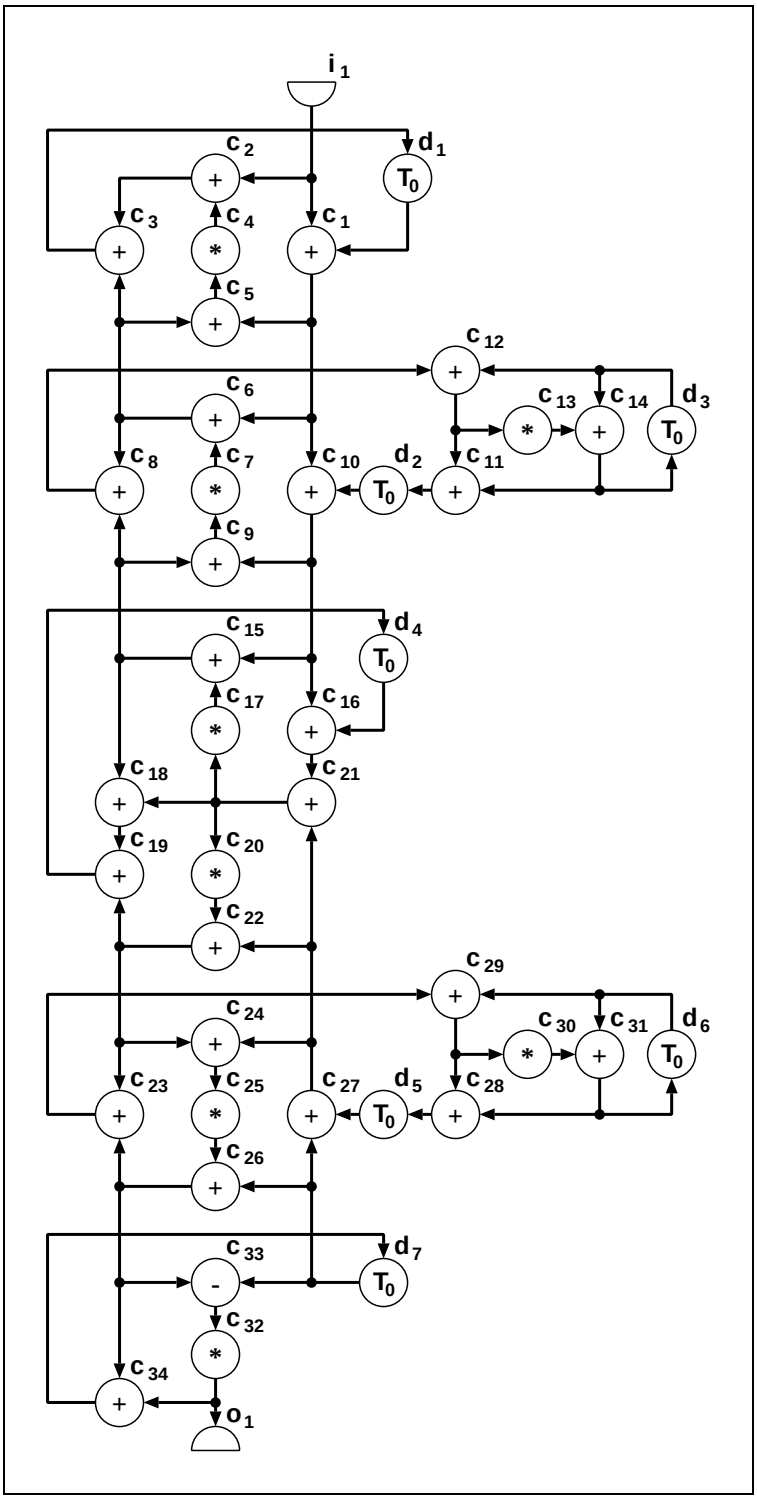

Figure 5.5: *The IDFG for the fifth-order wave digital elliptic filter.*

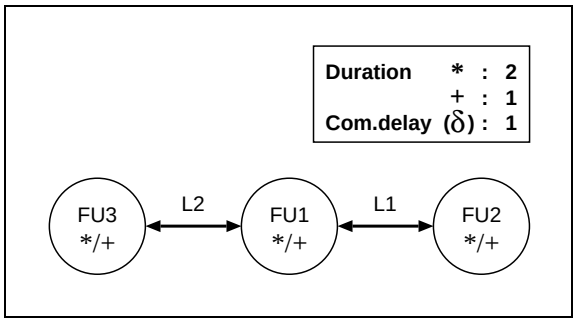

Figure 5.6: *The multiprocessor configuration used in the example.*

	Length [TU]
Loops:	
21-17-15-9-7-6-5-4-2-3-D-1-10-16	16
21-17-15-9-7-6-8-12-13-14-11-D-10-16	16
21-20-22-24-25-26-23-29-30-31-28-D-27	15
21-20-22-24-25-26-33-32-34-D-27	13
I/O-path:	
1-10-16-21-20-22-24-25-26-33-32	14

Table 5.2: *The most important loops and paths in the elliptic filter. Multiplication operations are underlined.*

period. Next to that, a subgoal was to minimize the latency. One of the schedules that was generated is shown in Figure 5.7. It can be seen that the iteration period of the schedule is 18 TU and the latency is 20 TU. A better schedule was not found[4].

Clearly, both the iteration period and the latency do not equal the minimum theoretical bounds IPB and PDB, whereas the number of processors in the multiprocessor configuration is equal to PB.

However, it can easily be seen that the minimal iteration period can not be reached when there is no duplication of operations and only fully static schedules are considered. Both critical loops need to be executed on a single FU in order not to introduce communication delays. This can not be the same FU because the iteration period then needs to be at least 22 TU. However, it is not possible to schedule both loops entirely on two separate FUs because they have operation c_{21} in common. Therefore at least two communication delays are introduced and the iteration period can not be less than 18 TU.

A latency of 14 TU is possible by scheduling all the operations in the critical I/O path on one FU. This can not be done, however, to obtain a schedule with $T_0 = 18\,\text{TU}$. Whether a latency smaller than 20 TU is possible can not easily be determined. It is expected that the latency can not be much smaller; it may even be optimal already.

It is certainly not easy to find a schedule with an optimal T_0 for the problem given in this

[4]It should be remarked that this schedule is better than the schedule that was claimed to be optimal in [BG97].

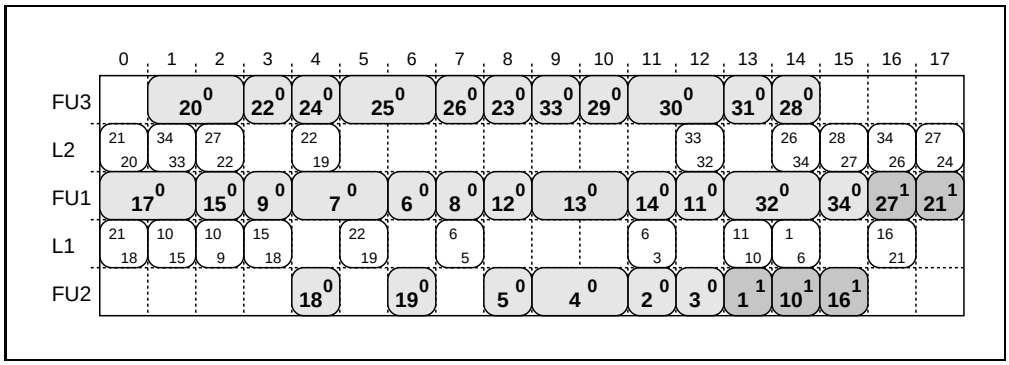

Figure 5.7: *A schedule for the digital elliptic filter that is found by the global heuristic in combination with the black-box heuristic.*

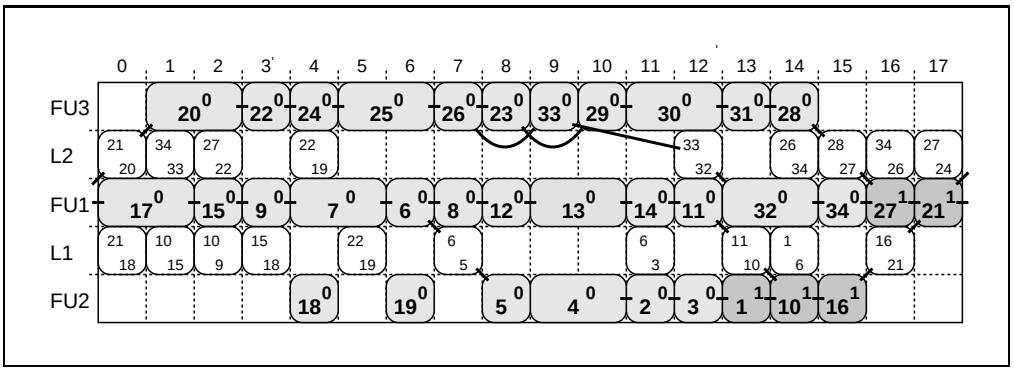

Figure 5.8: *The schedule for the digital elliptic filter that was already given in figure 5.7. The most important loops are now shown.*

section[5]. The main reason is that there are four important loops (see Table 5.2), which all have operation c_{21} in common, and there are only three FUs available. This makes it very hard to schedule these loops when there are communication delays and the interconnection network has a limited capacity and a limited structure.

How these loops are scheduled in the optimal schedule that is given in Figure 5.7, is shown in Figure 5.8. It can be seen that each of the four loops are scheduled on two FUs. That the complicated way in which the four loops are scheduled is necessary to get an optimum iteriation period, will become clear when you try to find a schedule by hand.

[5]It should be noted that the scheduling method rarely finds a schedule with $T_0 = 18\,\text{TU}$ for the problem that is presented here. See also Section 8.3, Problem M.

6

The Genetic Algorithm

The top layer of the scheduling method, the genetic algorithm, is discussed in this chapter. It consists of only two sections. First, Section 6.1 explains the basic principles of genetic algorithms. Subsequently, Section 6.2 presents the implementation that is used.

6.1 Basic Principles of Genetic Algorithms

Genetic algorithms (GAs) are general methods that can be used to solve a wide range of optimization or search problems. GAs are inspired by genetic processes that can be found in nature. They are based on the principle that is known as the "survival of the fittest" that was first described by Charles Darwin in "The Origin of Species".

[Gol89] and [Dav91] give a detailed description of GAs. Both also present numerous examples of applications that use GAs. [BBM93a] and [BBM93b] jointly give a thorough overview of GAs. The former discusses the fundamentals of GAs whereas the latter explores various more advanced aspects. There is a huge diversity of genetic algorithms, but most of them share the basic principles that are explained below.

6.1.1 Organisms

A genetic algorithm maintains a *pool of organisms* or a *population* in short. Every *organism* represents a feasible solution of the optimization problem.

Genotype

An organism has a *genotype*. A genotype consists of one or more *chromosomes*, typically one. A chromosome is a string of values or *genes*. These genes can be seen as parameters that define a feasible solution. In other words, the genotype of an organism encodes a feasible solution of the problem.

Several different types of chromosomes exist. For example, each gene in a chromosome can be a binary symbol with possible values "0" and "1". These chromosomes are called *binary*

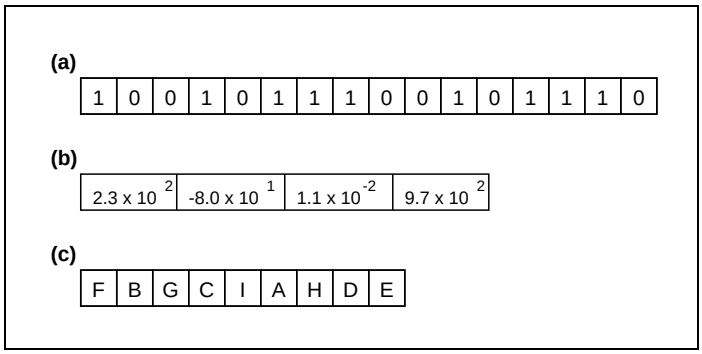

Figure 6.1: *Examples of different chromosome types: (a) a binary chromosome, (b) a floating point chromosome, and (c) a sequence chromsome.*

chromosomes. However, chromosomes exist that have more complex genes, e.g. each gene can be a real number. *Sequence chromosomes* are another type of chromosome. Sequence chromosomes encode an order of elements, where it is often required that every element occurs exactly once. Figure 6.1 illustrates these different types of chromosomes.

Phenotype

A genetic algorithm contains an *evaluation function*. The evaluation function can evaluate a genotype to construct the solution that it encodes. Inspired by biological terminology this solution is called the *phenotype* of the organism. What the evaluation function looks like highly depends on the genetic algorithm and the problem that has to be solved. It can be very simple and fast when the genotype encodes a phenotype in a straightforward way but it can also be complicated and time consuming[1]. The execution of the evaluation function is often simply called an *evaluation*.

Fitness

Every solution that is found by the genetic algorithm is assigned a *fitness* or *score*, a numerical value that is directly related to the quality of the solution. The calculation of the fitness is performed by a *fitness function*. The fitness can be based on a single feature of the solution. However, GAs often use more sophisticated fitness functions that are based on a combination of different performance measures. In order to improve the performance of the GA, *fitness normalization* is often applied to the raw fitness value of each of the organisms in the population. The fitness values are adjusted such that they better reflect the relative differences in fitness between the organisms, in order to improve the convergence behaviour of the population.

[1]The latter is the case in the proposed scheduling method. Although the global heuristic and the black-box heuristic execute fast, the overhead of the GA is negligible to the run time of both heuristics.

6.1.2 Evolution

A genetic algorithm tries to optimize the fitness of every organism in the population by a process called *evolution*. A population evolves by mating organisms to create new *offspring*. Usually, different *generations* are distinguished during evolution. In each generation a new population is created. At the start of the GA, an initial population is created by filling it with organisms that have a randomly generated genotype. Subsequently, for each generation the current population is used to derive the population for the next generation.

The GA continues until the *termination criterion* is met. This can be when a fixed number of generations or a fixed number of evaluations have passed. However, there are termination criteria that are slightly more sophisticated. For instance, criteria exist that cause a GA to stop as soon as its population has converged to a certain extent. This is in general more efficient, because no execution time is wasted when it is unlikely that fitter organisms are created.

Parent selection

The reason that the process of evolution optimizes the fitness of the organisms in the population, is that fitter organisms have a higher probability to be selected for mating. Therefore their genetic information, which turned out to produce a good solution, has a higher probability of being used to produce offspring.

Different *parent selection* mechanisms exist that pick parents for mating. A well-known parent selection mechanisms is *roulette wheel selection*. The probability that an organism is selected is proportional to the value of its fitness divided by the fitness summed over all organisms in the population. This can be pictured by imagining a roulette wheel where every organism is assigned a pie-slice proportional to its fitness.

Another parent selection mechanism is *tournament selection*. A fixed number of organisms is randomly picked from the population and from these organisms the one is selected that has the best fitness. The user must specify the number of organisms that is picked, which is called the *tournament size*. Changing the tournament size affects the *selection pressure*. A relatively large tournament size reduces the chance that less fit organisms are selected for mating.

Genetic operators

Genetic algorithms use *genetic operators* that build new chromosomes from existing ones. Mating is typically performed by applying a *crossover* operator. The crossover operator uses two chromosomes (from different parents) to construct one or two[2] new chromosomes. Without loss of generality, it is assumed here that crossover creates a single new chromosome.

Several crossover operators exist. One of them is *single point crossover*. First, a position is chosen where both parent chromosomes are cut into two parts. Subsequently the "head" of the first chromosome is combined with the "tail" of the second chromosome to construct a new chromosome. An example is shown in Figure 6.2.

[2]The second chromosome can always be created from the genetic material that was not used to create the first chromosome.

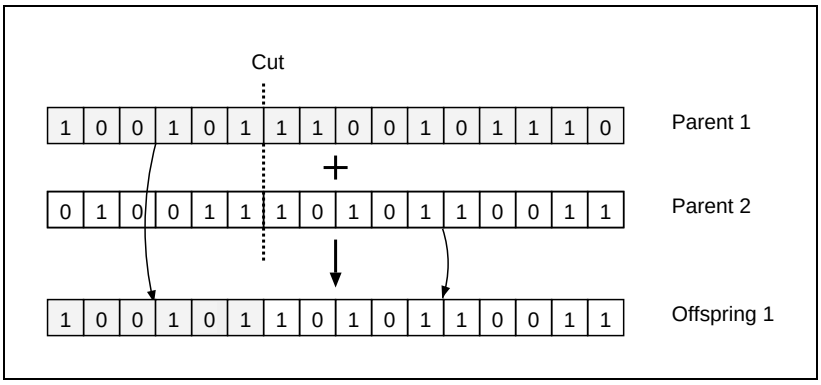

Figure 6.2: *Single point crossover.*

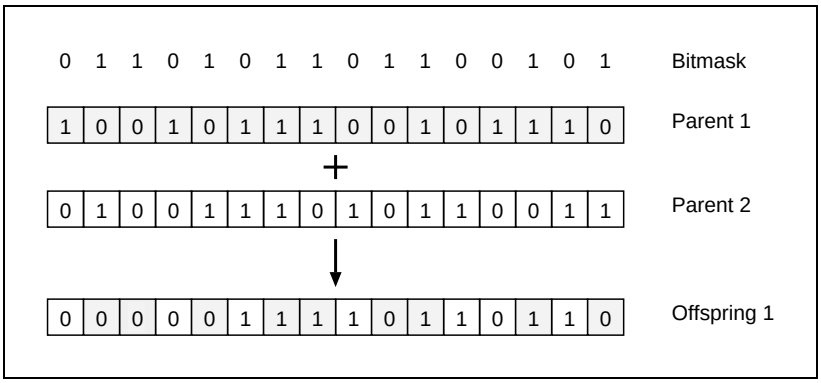

Figure 6.3: *Uniform crossover.*

Another crossover operator that is often used is *uniform crossover*. A random bit mask is created with a length that equals the number of genes in the chromosome. Every bit in the mask corresponds to a gene position. If the bit is "1" the corresponding gene of the first parent is picked, otherwise the gene of the second parent is copied to the child chromosome. This is illustrated in Figure 6.3. When the bit mask is randomly generated, the probability must be specified that a bit will be "1". The probability is called the *copy bias*.

Not every new organism is created by crossover. Mutation is another operator that is often used. Mutation "distorts" or mutates a single chromosome to create a new one. An example of a mutation operator that can be applied to binary chromosomes is *bit mutation*. A bit at a random position in the chromosome is selected and its value is changed. One of the advantages of using mutation is that it can introduce new (or maybe inadvertently lost) genetic information in a population.

To summarize the main principles of GAs, Figure 6.4 illustrates a plain genetic algorithm. The names of the procedures that are called have been chosen such that they indicate what they are supposed to compute.

```
procedure plain_genetic_algorithm()
begin
   generations := 0;
   evaluations := 0;

   { "Initialize population" }
   new_population := {};
   for i=1 to population_size
      organism.genotype := create_random_genotype();
      organism.phenotype := evaluate(organism.genotype);
      organism.fitness := calculate_fitness(organism.phenotype);
      evaluations := evaluations + 1;

      new_population := new_population ∪ {organism};
   end

   { "Evolve population" }
   repeat
      generations := generations + 1;
      old_population := new_population;
      new_population := {};

      for i=1 to population_size
         parent1 := select_parent(old_population);
         parent2 := select_parent(old_population);

         child.genotype := crossover(parent1, parent2);
         child.phenotype := evaluate(child.genotype);
         child.fitness := calculate_fitness(child.phenotype);
         evaluations := evaluations + 1;

         new_population := new_population ∪ {child};
      end
   until (population_converged(new_population) or
            generations > generation_limit);
end
```

Figure 6.4: *A plain genetic algorithm.*

6.2 Implementation Details

The current section briefly discusses the genetic algorithm that is used in the proposed scheduling method. Only the most important characteristics of the GA are discussed. Aspects of the GA that can easily be replaced, e.g. the parent selection mechanism or the genetic operators, are not discussed here. Section 7.4 examines these superficial implementation details.

The genotype of each organism consists of a single chromosome. The chromosome is a sequence chromosome that represents a permutation of operations that have to be scheduled. Therefore its length equals the number of operation nodes in the DFG.

The phenotype is a valid schedule for the DFG so that it can be executed on the hardware configuration supplied by the user. The evaluation function that constructs the schedule from the genotype is the global heuristic algorithm in combination with the black-box heuristic algorithm.

The fitness of every organism is determined by two characteristics of the schedule it represents. Namely the iteration period and the latency. The fitness function is such that minimizing the iteration period is always valued more than minimizing the latency. An exact equation is given in Section 7.4.3.

The genetic algorithm is implemented in the *GECO* (Genetic Evolution through Combination of Objects) package version 2.0 by G.P.W. Williams, Jr. GECO is an extensible, object-oriented framework for prototyping genetic algorithms in Common Lisp.

7

Tuning the Scheduling Method

It is recommended to experiment with the scheduling method to see if its performance can be improved. This is referred to as the tuning of the scheduling method and it is discussed in this chapter.

A small problem arises when different versions of the scheduling method have to be compared. This can not be done straightaway. Section 7.1 explains why. It also presents the measure, called the probability of success, that is used to overcome the problem.

The probability of success is used in the next three sections. Section 7.2 discusses the tuning of the global heuristic. Some small changes are made to the global heuristic that was presented in Chapter 4. The scheduling results of these variations are compared in order to see which gives the best performance. Similarly, Section 7.3 discusses the tuning of the black-box heuristic. Finally, Section 7.4 tries to find a good configuration for the genetic algorithm. Amongst others, several genetic operators are compared.

7.1 Probability of Success

Two different versions of a genetic algorithm that solve the same problem can not be compared directly. For there are two characteristics of the genetic algorithm that are of importance. The first one is the probability that a *satisfactory solution* is found after the GA has terminated. A satisfactory solution is a solution that has a fitness value that is as good as or better than a fitness value limit specified by the user. The fitness limit can be set such that the set of satisfactory solutions only contains optimal solutions. However, it is also possible that solutions that have a fitness value that is close to optimal are considered satisfactory too.

The second characteristic of a GA that is of importance, is the average time that it takes the GA to terminate. Different runs of the same GA can have run times that differ a lot. This is the case when a termination criterion is used that stops the GA once the population has converged. When the overhead of the genetic algorithm is negligible compared to the time it requires to evaluate the genotype, which is the case for the proposed scheduling method, the run time can be expressed in the number of evaluations that were required.

To illustrate the difficulty of comparing two genetic algorithms, consider the following example.

What is better? Algorithm I that requires an average of 612 evaluations and that finds a satisfactory solution 46% of the time. Or a Algorithm II that finds a satisfactory solution 57% of the time but that requires on average 760 evaluations.

The answer can not be given without a precise knowledge of the user's wishes. For instance, it is possible that the user can not allow more than 720 evaluations for some peculiar reason. It is then very likely that the user prefers Algorithm I. Nevertheless, without knowledge of the user's wishes a general comparison can still be made. In order to do so, the *probability of success for a fixed number of evaluations*, $P_s(R)$, is defined:

$$P_s(R) = 1 - (1 - p_s)^{R/E} \tag{7.1}$$

Where R is the total number of evaluations that the GA is allowed to use, p_s the probability that a run of the GA finds a satisfactory solution and E the average number of evaluations it takes the GA to terminate.

$P_s(R)$ is called probability of success in short. It can be interpreted as the probability that the GA finds a satisfactory solution when it is allowed to use R evaluations. These R evaluations may be distributed over more than one run of the genetic algorithm. This interpretation of $P_s(R)$, as $P_s(R)$ itself, should be used with care as is explained later on in this section.

First, however, the use of $P_s(R)$ is illustrated by comparing Algorithm I and Algorithm II. It follows that:

Algorithm I: $P_s(1000) = 1 - (1 - 0.46)^{1000/612} = 0.63$
Algorithm II: $P_s(1000) = 1 - (1 - 0.57)^{1000/760} = 0.67$

So, based on the probability of success, Algorithm II is rated slightly better than Algorithm I.

Note that comparing genetic algorithms by way of $P_s(R)$ is only sensible when the average run time that an evaluation requires is the same for all genetic algorithms. This is the case in this thesis; otherwise, it is useful to define a probability of success based on a fixed run time.

Finally, it must be stressed that the probability of success is a useful concept but it should always be used with common sense. What is meant here with common sense will be clarified by an example.

Consider a single run of a GA that generated a satisfactory solution and required 1200 evaluations. It then follows that $P_s(2000) = 1.00$. However, that does not say very much because it is based on only a single run. Therefore to be of use, the probability of success should always be based on a considerable number of runs of the GA.

Furthermore, in the above situation it also follows that $P_s(10) = 1.00$, which gives a completely incorrect impression and is entirely useless. For the probability that the genetic algorithm finds a satisfactory solution after ten evaluations is probably close to zero. So, it is not recommended to use $P_s(R)$ with a value of R that is less than the average number of evaluations that the GA requires.

Finally, suppose that the GA is executed many times and it turns out that $P_s = 0.56$ and

$E = 1220$. It then follows that $P_s(1600) = 0.66$. However, the probability that a satisfactory solution is found in 1600 evaluations is probably not equal to 66%. For it is likely that a GA run must be interrupted prematurely to avoid that more than 1600 evaluations are used. Since the value of $P_s(R)$ is merely based on completed runs of the GA, the actual probability is likely different.

7.2 Tuning the Global Heuristic

This section discusses the tuning of the global heuristic. Section 7.2.1 examines how much effect the initial iteration period that is provided as a parameter to the global heuristic, has on the quality of the schedules that are produced. Section 7.2.2 looks at several algorithms that use the list that is provided to the global heuristic to construct an actual order in which the operations are scheduled. Section 7.2.3 experiments with different heuristics that select a preferred start time for operations. Finally, Section 7.2.4 summarizes the most important results from the previous sections and gives some recommendations.

7.2.1 Initial Iteration Period

The genetic algorithm must provide the global heuristic with an initial iteration period $T_{0,initial}$ (see Section 4.2). In order to make sure that precedence relations can always be satisfied, $T_{0,initial}$ can not be less than IPB. However, the minimal possible T_0 for a given scheduling problem can be larger than IPB. An example can be found in Section 5.6. Therefore it is interesting to see to what extent the value of $T_{0,initial}$ affects the performance of the scheduling method.

A $T_{0,initial}$ that is larger than the optimum T_0, $T_{0,optimum}$, never leads to an optimum schedule. For the global heuristic can increase the iteration period but not decrease it. However, it is expected that when $IPB \leq a < b \leq T_{0,optimum}$, a GA with $T_{0,initial} = b$ will on average produce better results then a GA with $T_{0,initial} = a$. The reasoning is as follows. Inserting a cycle in the schedule disrupts the schedule. It creates one free time slot in the schedule for every resource. It is questionable whether these slots can all be used efficiently, especially when almost all operations have already been scheduled. On the average, less cycles need to be inserted in the schedule for a higher value of $T_{0,initial}$. So, it is expected that a higher value of $T_{0,initial}$ improves the average quality of the schedules that are produced, as long as $T_{0,initial} \leq T_{0,optimum}$. This hypothesis is supported by results that are now presented.

The scheduling problem that is considered, is the same as the one used in Section 5.6. Namely, the fifth-order wave digital elliptic filter (see Figure 5.5) is scheduled onto a chain-structured multiprocessor consisting of three FUs (see Figure 7.1).

The configuration of the genetic algorithm is shown in Table 7.1. It is chosen such that the population converges reasonably fast. Section 7.4 explains the configuration of the GA and presents empirical results that support why this particular configuration is chosen.

Table 7.1 also shows that the scheduling method was executed 50 times for each value of $T_{0,initial}$. Scheduling results are compared by using the probability of success. Therefore the

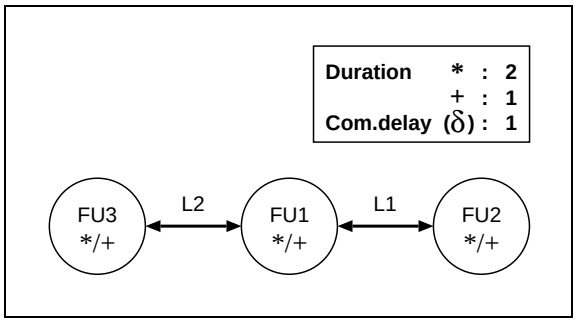

Figure 7.1: *The multiprocessor configuration that is used to tune the global heuristic.*

Scheduling problem	DFG	fifth-order elliptic filter
	Hardware	see Figure 7.1
Genetic algorithm	Population size	60
	Crossover	80%, UNIPERM (copy bias=0.50)
	Mutation	0%
	Selection	tournament (size=6)
	Termination	margin=10%, fraction=90% *or* max. number of evaluations=5000
Results	Number of runs	50
	Satisfactory solution	$T_0 \leq 19\,\mathrm{TU}$

Table 7.1: *Configuration of the scheduling method used to generate the data in Table 7.2, 7.3 and 7.4.*

table also shows what is considered a satisfactory solution. Note that schedules that have an iteration period of 19 TU are also considered satisfactory, even though the optimum iteration period is 18 TU (see Section 5.6). This is done to present useful results because the scheduling method does not find the optimum schedule very often (see Section 8.3, Problem M).

The initial iteration period is varied from 16 TU (IPB) to 19 TU. The results are shown in Table 7.2. The table presents three figures for each value of $T_{0,initial}$. The figure at the top is the average number of evaluations that the GA required. The bottom value gives the fraction of runs that produced a satisfactory solution. These two values are used to calculate a probability of success, namely $P_s(1000)$.

The table shows that the value of $T_{0,initial}$ hardly affects the average run time of the GA. However, the probability that the GA is successful is considerably higher for increasing values of $T_{0,initial}$.

$T_{0,initial}$							
16		17		18		19	
752	0.31	757	0.60	753	0.87	781	0.93
0.24		0.50		0.78		0.88	

Table 7.2: *Probability of success when the initial iteration period is varied (configuration of Table 7.1).*

Algorithm to choose operations			
I		II	
818 0.32	0.38	712 0.10	0.14

Table 7.3: *Probability of success when the algorithm that chooses the next operation that is scheduled is varied (configuration of Table 7.1).*

7.2.2 Choosing an Operation

The global heuristic uses a list P that is provided by the GA to choose the operation that is scheduled next. However, it can deviate from the order of operations specified by P. This is for instance the case for the algorithm of the global heuristic presented in Chapter 4. There it is required that a direct predecessor or successor operation of the operation that is scheduled next, has already been scheduled. See also Section 4.3.1. This algorithm of choosing an operation is called here Algorithm I.

There are other ways to choose the next operation that is scheduled. A different approach is to relax the requirement slightly. Namely, pick the first operation from P for which a direct or indirect predecessor or successor operation has already been scheduled. This algorithm is called Algorithm II. The algorithm has the advantage that there are more different orders in which operations can be scheduled. So, there are less chromosomes that produce the same schedule. However, the average quality of these schedules that are produced, deteriorated. For it is harder for the greedy heuristic to always find a good way to schedule each operation.

Algorithm I and II were compared by solving the scheduling problem that was already used in the previous section. The rest of the configuration was also the same, see Table 7.1. The initial iteration period was set to 16 TU.

The scheduling results can be found in Table 7.3. The probability of success that is shown is again $P_s(1000)$. It can be seen that the algorithm that was originally proposed to select the operations (Algorithm I) is superior. Also note that 50 runs is actually too little. The results in the left most column of Table 7.2 and Table 7.3 are based on exactly the same configuration of the scheduling method. Still, both results differ. This inaccuracy must be kept in mind throughout this chapter.

Due to lack of time, several other possible ways to choose operations from the list P have not been tested. Some of those are mentioned below. It is not very likely that any of these algorithm outperforms Algorithm I as will be explained. The first variant is to strictly follow the order of operations in the list P. [Hei96] reports that ignoring any topological information, results in schedules with an inferior quality. Furthermore, since the algorithm is an "exaggeration" of Algorithm II, Table 7.3 gives no reason to believe that the results will be good either.

A variant that is also not tested is the following: always schedule operations that are part of a loop first, and once these are all scheduled consider the remaining operations. It may look a promising approach at first sight, because operations that are not part of a loop do not affect the iteration period. However, in practice it will not make much, if any, difference. For in commonly used DSP filters, almost every operation is part of a loop. Consider for instance the fifth-order elliptic filter, where every operation is part of a loop, and also the filters that are

```
function get_pref_time_bounded(c, R_nc)
  { "Determine the preferred start time t_pref for operation c.
     R_nc is the range of valid start times of the operation and it
     is bounded." }
begin
  if R_nc,min ≥ 0
    t_pref = R_nc,min;
  else if R_nc,max ≤ 0
     t_pref = R_nc,max;
  else
     t_pref = 0;

  return t_pref;
end
```

Figure 7.2: *The procedure "get_pref_start_time_bounded" that implements Heuristic II.*

used in Chapter 8.

Other variants that are not tested are algorithms that incorporate the slack time of loops in their decision. Loops that have a small slack time are harder to schedule. So, scheduling operations in these loops first probably produces schedules that on average have a better quality. However, it is questionable whether these algorithms outperform Algorithm I. One reason is that these algorithms severely limit the search space and effectively cripple the GA. Another reason follows after closely examining the convergence of the chromosomes during runs of the GA. It then follows that the GA is already able to discover quickly which operations should be scheduled first.

However, since the scheduling method rarely finds the optimal solution for the scheduling problem used in this section, it is still useful to experiment with different algorithms.

7.2.3 Preferred Start Time

For operations that have an unbounded range R_{nc} (operations that are not part of a loop) it is highly recommended to set the preferred start to the bounded limit of R_{nc}, as is done in Step 5 of the global heuristic. For there are no reasons why this should not be done and in this way the latency is minimized.

However, for operations that are part of a loop, and for which consequently the range R_{nc} is bounded, a straightforward answer of what the best value of t_{pref} is, can not be given. In Section 4.3.4 a heuristic is presented that determines a preferred start time. The heuristic is referred to in this section as Heuristic I. Although the heuristic is not very easy to understand, it is entirely based on common sense logic. Nevertheless, it is useful to see if other heuristics can lead to better scheduling results. Therefore, it is compared with two other heuristics. These are shown in Figures 7.2 and 7.3, and are respectively called Heuristic II and Heuristic III.

```
function get_pref_time_bounded(c, R_nc)
  { "Determine the preferred start time t_pref for operation c.
     R_nc is the range of valid start times of the operation and it
     is bounded." }
begin
  t_pref = ⌈(R_nc,min + R_nc,max)/2⌉;

  return t_pref;
end
```

Figure 7.3: *The procedure "get_pref_start_time_bounded" that implements Heuristic III.*

Heuristic for t_{pref}					
I		II		III	
688	0.35	651	0.26	724	0.08
0.26		0.18		0.06	

Table 7.4: *Probability of success for different heuristics that provide the preferred start time for operations with a bounded range of start times (configuration of Table 7.1).*

To make a comparison, the same scheduling problem and configuration of the GA are used as in the previous two sections (see Table 7.1). The initial iteration period is again 16 TU.

Table 7.4 shows the scheduling results. The probability of success that is shown is again $P_s(1000)$. The results indicate that the heuristic that was presented in Chapter 4 (Heuristic I) is a good choice indeed.

7.2.4 Conclusions and Recommendations

Section 7.2.1 showed that the initial period has a considerable effect on the probability that a satisfactory solution is found. It is recommended that this is exploited in order to improve the performance of the scheduling method. A straightforward way is to enable the user to provide a lower bound (which can default to IPB). A better way would be to automatically generate a more accurate lower bound. However, a lower bound depends on the black-box hardware model and it is questionable whether an algorithm can be found that quickly generates a lower bound on T_0.

An approach that may look promising is to restart the global heuristic with a higher value of $T_{0,initial}$ as soon as cycles need to be inserted. There are two reasons, however, why this approach is not recommended. First of all, it results in a considerable deterioration of the run time. A rough (optimistic) estimation is that the average run time will be "only" two times as long. The second reason is that it is not always (for every permutation P) possible to choose $T_{0,initial}$ such that no cycles need to be inserted. Some permutations will always create local problems in the schedule, in particular when the black-box heuristic is based on a restrictive hardware model.

A more realistic alternative is to make the GA responsible for determining a suitable value for $T_{0,initial}$ by encoding it in the genotype. This approach is expected to give the desired result and it should certainly be investigated.

Section 7.2.2 looked at different alternatives that determine the order in which operations are scheduled, based on the list P. It followed that the alternative given in Chapter 4 gives the best results. In Section 7.2.3 several heuristics that choose a value of t_{pref} were examined. It turned out that the heuristic presented in Chapter 4 was a good choice.

Finally, a more general remark can be made. It turned out that the scheduling method rarely finds the optimum solution for the scheduling problem that was used in the previous three sections. This is not too dramatic, for it is a very hard scheduling problem as is explained in Section 5.6. So, it is promising that an optimum iteration period can be found nevertheless. Still, it is preferable that an optimum schedule is found more often. Therefore it may be worthwhile to closely examine the global heuristic to see if it can be improved upon. However, care must be taken not too make the global heuristic too "smart". It should be kept as general as possible, because it must be able to perform well for a wide range of black-box heuristics (which may be based on different hardware models) and a wide range of scheduling problems (which can create other difficulties).

7.3 Tuning the Black-Box Heuristic

The tuning of the black-box heuristic is discussed in this section. Section 7.3.1 presents an experiment that examines whether the direction in which data transfers are scheduled affects the quality of the schedules that are produced. After that, Section 7.3.2 shows the effect that the numbering of the FUs has on the scheduling results. Section 7.3.3 follows by giving some conclusions and recommendations.

7.3.1 Scheduling Direction

A data transfer can be scheduled in two directions, from the source FU to the destination FU or in the opposite direction. Recall that in the algorithm of the black-box heuristic that was given in Chapter 5, the scheduling direction of a data transfer was based on which operation was scheduled first. The scheduling direction was namely such that it started from the operation that was scheduled the earliest. This choice of the scheduling direction is called Alternative I this section. A different approach is to always schedule a data transfer from the source operation to the destination operation. This is called Alternative II.

The algorithm that is solved by scheduling problem is again the fifth-order elliptic filter, which is given Figure 5.5. The target multiprocessor configuration is given in Figure 7.4 (a). It is chosen because it is expected that the way communications are scheduled will affect the quality of the schedules that are produced (which is not or hardly the case for the multiprocessor configuration that was used in Section 7.2). The rest of the configuration of the scheduling method is the same as in Section 7.2. It is summarized in Table 7.5.

The scheduling results are presented in Table 7.6. The table shows that both alternatives do

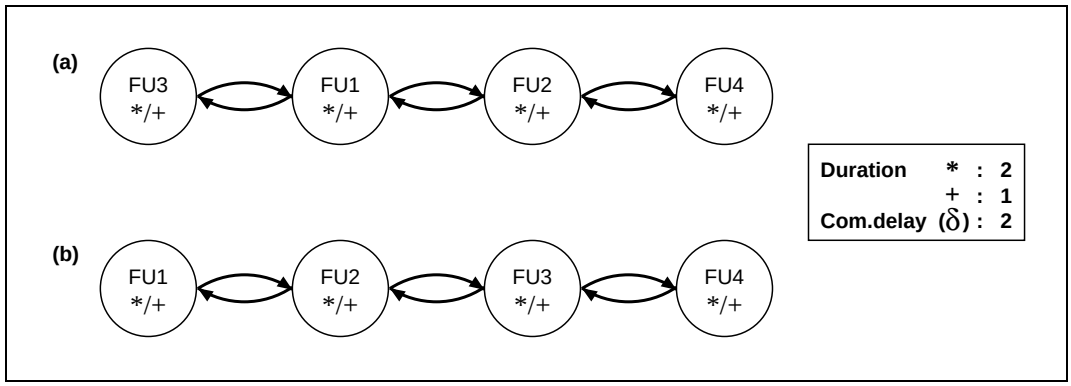

Figure 7.4: *(a)(b) The two multiprocessor configurations that are used to tune the black-box algorithm. Note that only the numbering of the FUs differs.*

Scheduling problem	DFG	fifth-order elliptic filter
	Hardware	see Figure 7.4 (a)/(b)
Genetic algorithm	Population size	60
	Crossover	80%, UNIPERM (copy bias=0.50)
	Mutation	0%
	Selection	tournament (size=6)
	Termination	margin=10%, fraction=90% *or* max. number of evaluations=5000
Results	Number of runs	50
	Satisfactory solution	$T_0 \leq 21\,\mathrm{TU}$

Table 7.5: *Configuration of the scheduling method used to generate the data in Table 7.6 and 7.7.*

Scheduling direction			
I		II	
688 0.56	0.70	788 0.56	0.65

Table 7.6: *Probability of success for different scheduling directions of the data transfers in the black-box heuristic (configuration of Table 7.5).*

FU Numbering			
I		II	
763 0.56	0.66	705 0.32	0.42

Table 7.7: *Probability of success for different FU numberings (configuration of Table 7.5).*

not differ too much in performance. Both alternatives produced a satisfactory solution equally often. Alternative I had an average run time that was slightly better. However, since the difference is small and the results are based on "only" 50 runs of the GA, this difference can be ignored.

7.3.2 Numbering of FUs

As was explained in Section 5.4.2 the way the FUs are numbered in the multiprocessor configuration matters because it influences the assignment of operations to FUs. This section presents an experiment that illustrates the significance of the numbering of FUs.

Figures 7.4 (a) and (b) show the two multiprocessor configurations that are used. These are called Configuration I and II respectively. Both define the same multiprocessor structure, they only differ in the way the FUs are numbered. The rest of the scheduling problem is equivalent to the one used in the previous section (see Table 7.5).

The scheduling results can be found in Table 7.7. It shows that the effect of the FU numbering is indeed significant. Furthermore, the fact that Configuration I gives the best results supports the reasoning in Section 5.4.2.

7.3.3 Conclusions and Recommendations

Section 7.3.1 showed that for the tested scheduling problem, the direction in which the data transfers are scheduled hardly matters. Since the underlying reasoning for the original way of choosing the scheduling direction has not been invalidated, the implementation is not changed.

In Section 7.3.2, it was confirmed that the way FUs are numbered is of influence on the performance of the scheduling method. The effect is significant and should not be ignored. To avoid that the user must specify a good numbering, an automated way of numbering the FUs is preferred. Since the guidelines that lead to a good numbering are not very complicated, it is expected that a simple heuristic algorithm can be designed that numbers the FUs in a multiprocessor configuration.

A different, fancy, approach is to include an encoding of the FU numbering in the GA (in a similar way as the permutation of operations). However, it results in an increased run time of the GA because it will take longer for the population to converge. Still, it is interesting to compare the quality of the schedules that are produced with the quality of the schedules produced when a simple numbering heuristic is applied.

7.4 Tuning the Genetic Algorithm

This section presents some experiments that have been carried out in order to tune the genetic algorithm. First, in Section 7.4.1 different crossover operators are compared. Section 7.4.2 examines how mutation affects the scheduling results. After that, Section 7.4.3 discusses the effect of the fitness function on the performance of the scheduling method. Finally, Section 7.4.4 ends by giving some conclusions and recommendations.

7.4.1 Crossover operator

Since each chromosome encodes a permutation of operations, the GA must use a sequence-based crossover operator. The two crossover operators that are mentioned in Section 6.1.2, can therefore not be used. Both can create illegal sequences in which some operations occur more than once and some not at all.

Several sequence-based crossover operators exist. Three of these are compared in this section. The first is *random respectful recombination* crossover [Rad92]. This crossover operator is provided by the GECO package. It is referred to here as R3 crossover. The second crossover operator is also provided by the GECO package. It is *partially mapped* (PMX) crossover [Gol89]. The third crossover operator resembles ordinary uniform crossover. It is called *uniform crossover for permutations*; it should be noted that different names have been given to this operator [Dav91] [Sys91] [Hei96]. The operator is referred to as UNIPERM crossover in short.

To illustrate what a sequence crossover operator looks like, a description of UNIPERM crossover follows. First, a random bit mask is created in which every bit corresponds to a gene position in the chromosome. At every position where the bit is "1", the gene from the first parent is copied to the corresponding position in the child. Subsequently, the remaining genes are copied, in the order that they occur in the second parent, to the still vacant positions in the child chromosome. An example is shown in Figure 7.5.

The configuration of the scheduling problem that was used to compare the three crossover operators is shown in Table 7.8. The configuration of the GA can be found in Table 7.9. The mutation operator is explained in the next section. The GA terminates after a fixed number of evaluations have passed or after the population has converged. The convergence criterion is that 90% of the organisms must have a fitness that is within 10% of the fitness of the best organism, where 100% refers to the fitness of the worst organism. Since the crossover operators are not equally disruptive they may require a distinct population size to perform well. Therefore, the population size was also varied in order to make a fair comparison. The uniform crossover operator for sequences was tested for two values of the copy bias, namely 0.50 and 0.75.

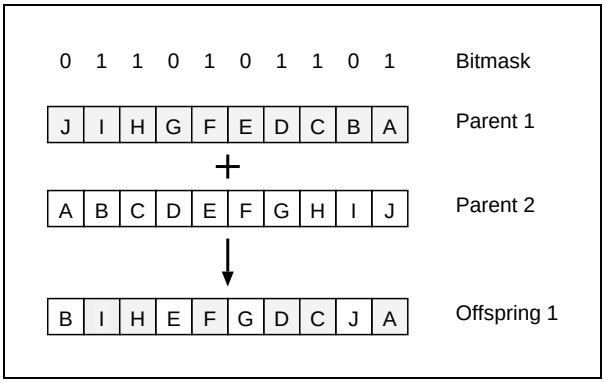

Figure 7.5: *Uniform crossover for sequences.*

Scheduling problem	DFG Hardware	fifth-order elliptic filter see Figure 7.1
Heuristics	$T_{0,initial}$	16 TU
Genetic algorithm	*Various*	
Results	Number of runs Satisfactory solution	50 $T_0 \leq 19\,\mathrm{TU}$

Table 7.8: *Configuration of the scheduling problem used to generate the data in Table 7.10, 7.12 and 7.14.*

Genetic algorithm	Population size	see Table 7.10
	Crossover	80%, see Table 7.10
	Mutation	1%, SWAP
	Selection	tournament (size=6)
	Termination	margin=10%, fraction=90% *or* max. number of evaluations=5000

Table 7.9: *Configuration of the GA used to generate the data in Table 7.10.*

Crossover	Population size					
	40		60		120	
R3	195 0.02	0.27	429 0.00	0.00	2549 0.08	0.09
PMX	190 0.00	0.00	316 0.00	0.00	933 0.12	0.34
UNIPERM (0.50)	340 0.12	0.68	757 0.22	0.63	3078 0.64	0.63
UNIPERM (0.75)	270 0.02	0.20	645 0.20	0.65	2372 0.50	0.58

Table 7.10: *Probability of success for several different crossover operators, the population size is also varied (configuration of Table 7.8).*

The scheduling results can be found in Table 7.10. The probability of success that is shown, is $P_s(3000)$. The table shows that the UNIPERM operator is superior for the tested population sizes. PMX crossover may require a larger population size to perform well. However, the additional robustness of UNIPERM crossover causes that it is used in the GA. It turns out that the value of the copy bias does not have a major influence on the probability of success for the given scheduling problem. [Hei96] recommends a copy bias of 0.50 and therefore that is the value that is used in the GA.

7.4.2 Mutation Operator

In this section two sequence mutation operators are examined. The first operator is very straightforward. It randomly chooses two positions in the chromosome and swaps the genes. The operator is referred to as SWAP. The second mutation operator is hardly more complicated. It also selects two positions in the chromosome at random. Subsequently, it randomly reorders (scrambles) the genes that lie between these two positions. The operator is called the SCRAMBLE operator.

The two mutation operators are compared using the same configuration of the scheduling problem as was used in the previous section (see Table 7.8). Table 7.11 shows the configuration of the genetic algorithm. Since the SCRAMBLE operator is more disruptive than the SWAP operator, the two operators were compared for several values of $P_{mutation}$ (the probability that a chromosome is mutated). The GA was also tested with no mutation at all.

Table 7.12 shows the scheduling results. The probability of success that is shown is $P_s(2000)$. It follows from the table that the SWAP operator performs slightly better than the SCRAMBLE operator. However, it can also be seen that no mutation at all gives the best results. Therefore, it is decided not to use mutation in the genetic algorithm.

Genetic algorithm	Population size	90
	Crossover	80%, UNIPERM (copy bias=0.50)
	Mutation	see Table 7.12
	Selection	tournament (size=6)
	Termination	margin=10%, fraction=90% *or* max. number of evaluations=5000

Table 7.11: *Configuration of the GA used to generate the data in Table 7.12.*

	$P_{mutation}$							
Mutation	0.001		0.005		0.01		0.05	
SWAP	1398 0.44	0.56	1250 0.36	0.51	1281 0.42	0.57	1481 0.34	0.43
SCRAMBLE	1338 0.40	0.53	1314 0.38	0.52	1413 0.36	0.47	1745 0.38	0.42
No mutation	1229 0.52	0.70						

Table 7.12: *Probability of success for two different mutation operators and no mutation at all (configuration of Table 7.8).*

7.4.3 Fitness Function

In Section 6.2, it was mentioned that the fitness function minimizes both the iteration period and the latency, where minimizing the iteration period is always valued more. This can be done by using the following linear fitness function:

$$fitness = (T_0 - IPB) \times \alpha + \min(\alpha - 1, latency - PDB) \tag{7.2}$$

Where T_0 is the iteration period of the schedule and *latency* the latency of the schedule. α is a parameters that can adjust the fitness function specifically to the problem. It affects the number of latency values that produce a different fitness value. IPB and PDB give minimum bounds for respectively T_0 and the latency (see Section 2.1.3). Notice that the GA must minimize the fitness.

Four different values for α were used to create four different fitness functions; namely 1, 5, 10 and 20. In the preceding sections, the fitness function was used with $\alpha = 10$. Note that for $\alpha = 1$, the latency is not considered at all. Since the focus then lies on one optimization goal instead of two possibly conflicting ones, it can be expected that more often schedules are found with a satisfactory iteration limit. It is interesting to see if this hypothesis can be supported by empirical data. The scheduling problem that is considered, is the same as the one used in the previous two sections (see Table 7.8). The configuration of the GA is shown in Table 7.13. It should be noted that the GA does not directly use the fitness values calculated by Equation 7.2 when it selects parents for mating. The fitness value of every organism in the population is first normalized.

The scheduling results that were produced can be found in Table 7.14. The probability of

Genetic algorithm	Population size	60
	Crossover	80%, UNIPERM (copy bias=0.50)
	Mutation	0%
	Selection	tournament (size=6)
	Termination	margin=10%, fraction=90% *or* max. number of evaluations=5000

Table 7.13: *Configuration of the GA used to generate the data in Table 7.14.*

Fitness function							
$\alpha = 1$		$\alpha = 5$		$\alpha = 10$		$\alpha = 20$	
799	0.17	909	0.15	694	0.33	629	0.33
0.14		0.14		0.24		0.22	

Table 7.14: *Probability of success for different versions of the fitness function (configuration of Table 7.8).*

success that is shown is $P_s(1000)$. It can be seen that the results do not correspond to the hypothesis given above. The GA with $\alpha = 1$ performed considerably worse than the GAs with $\alpha = 10$ or $\alpha = 20$.

A possible explanation why the GA performs better when the latency is taken into account, is the following. A lot of permutations generate schedules with a near to optimal iteration period. When the GA does not consider the latency, it has no indication which of the schedules with an equal T_0 are more likely to produce good offspring. Whereas when the latency is included in the fitness, the GA is better able to select organisms for mating. It is possible because, for the scheduling problem that is considered, schedules with a small latency are more likely to produce fitter offspring.

7.4.4 Conclusions and Recommendations

Section 7.4.1 focused on the crossover operator. Several sequence crossover operators were considered, of which uniform crossover for permutations was superior. It can be noted that [Dav91] also reports that this operator gives the best results.

In Section 7.4.2 the mutation operator was considered. Two different sequence based mutation operators were compared. Somewhat surprisingly, the scheduling method performed best when no mutation at all was used. So, it can be concluded that the probability that mutation creates an inferior schedule, outweighs the positive effects of applying mutation.

Section 7.4.3 presented an experiment where different fitness functions were compared. An important conclusion was that it is beneficial to base the fitness function on more than T_0 alone, even when minimizing T_0 is the only optimization goal. So, it is highly recommended to design a fitness function that reflects more accurately which organisms are likely to produce offspring with an improved iteration period. Since minimizing T_0 and minimizing the latency can be conflicting goals for certain scheduling problems, it is recommended to include other measures in the fitness function. There are various measures that look promising, e.g. the total number of communication delays in the (most important) loops, the latest step that cycles were

inserted in the schedule, the total number of communication tasks etc. Notice that a fitness function that uses the latter measure, depends on the black-box hardware model. It is expected that an elaborate fitness function can lead to a drastic improvement of the performance of the GA.

Other aspects of the GA have been examined as well, amongst others the parent selection mechanism, the termination criterion and the population size. These have not discussed here, because the results were not very interesting. However, these results have been taken into account in order to tune the GA satisfactorily.

Finally, it can be noted that the GA that is used, is a rather plain and straightforward GA. More complicated and sophisticated GAs have been designed. There exist for instance GAs that evolve multiple populations in parallel, GAs that adaptively set operator parameters [Dav91], etc. It may be worthwhile to experiment with these. However, whether this will lead to a considerable improvement in performance of the GA, remains to be seen.

8

Results

In this chapter the performance of the proposed scheduling method is illustrated by presenting a large set of representative scheduling results. The environment that is used to generate the results is described in Section 8.1. Then, Section 8.2 presents results for scheduling problems where communication delays are negligible. Subsequently, scheduling problems are considered that are based on the detailed black-box hardware model. This is done in Section 8.3.

8.1 Benchmark Environment

Five DFGs are used to test the performance of the proposed scheduling method. Table 8.1 lists these five filters. The table gives for each filter the shorthand name that is used in this chapter, a short description and a reference to the figure that defines the filter. Note that the second-order filter was already given in Figure 2.1. It is repeated here for sake of convenience. The fifth-order wave-digital elliptic filter was first used in Chapter 5. The figure is not repeated here, because of its size. These five filters have all been used before in scheduling literature, see for instance [HdGGH92] [KG95] [Hei96].

In this chapter, addition and a subtraction always require 1 TU. Because both operations have the same duration, they will not be considered separately: the "+" symbol denotes both.

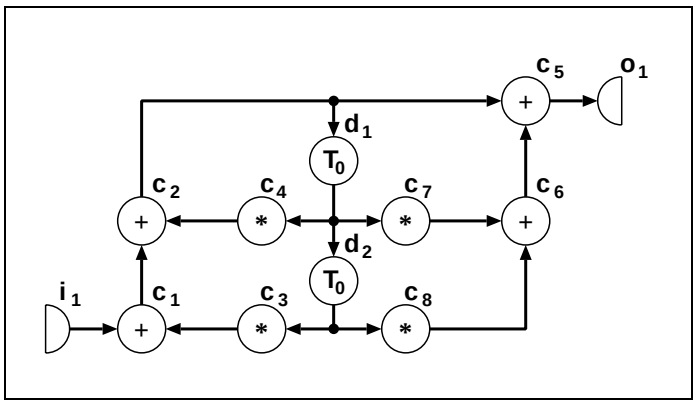

Figure 8.1: *The IDFG for the second-order filter.*

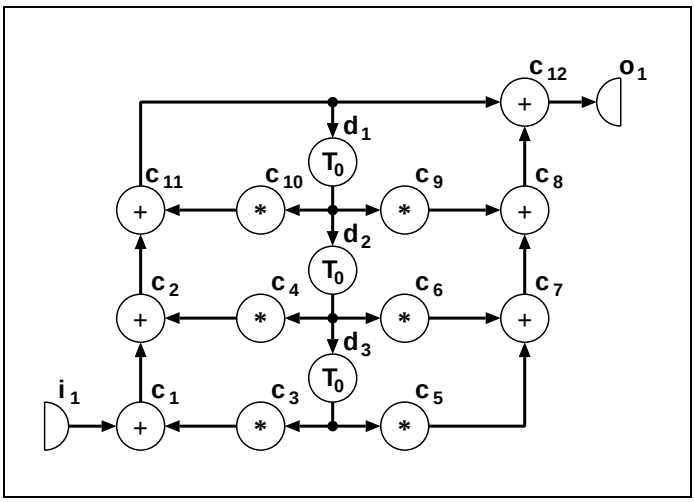

Figure 8.2: *The IDFG for the third-order filter.*

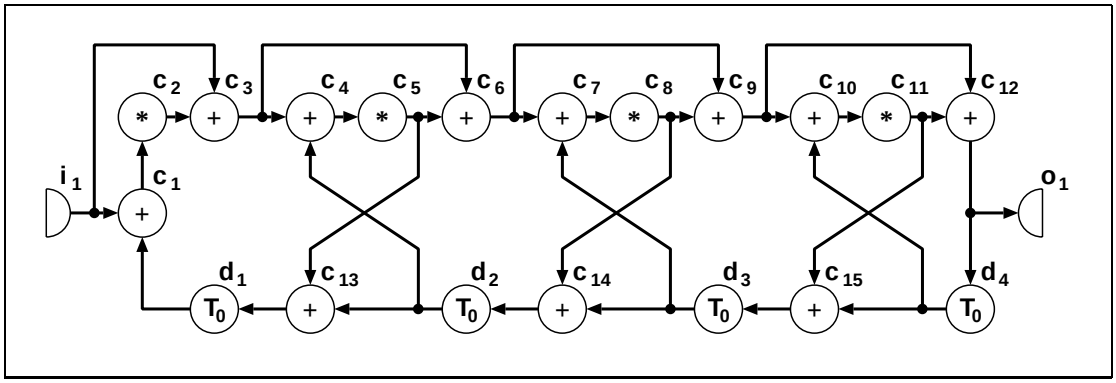

Figure 8.3: *The IDFG for the fourth-order all-pole lattice filter.*

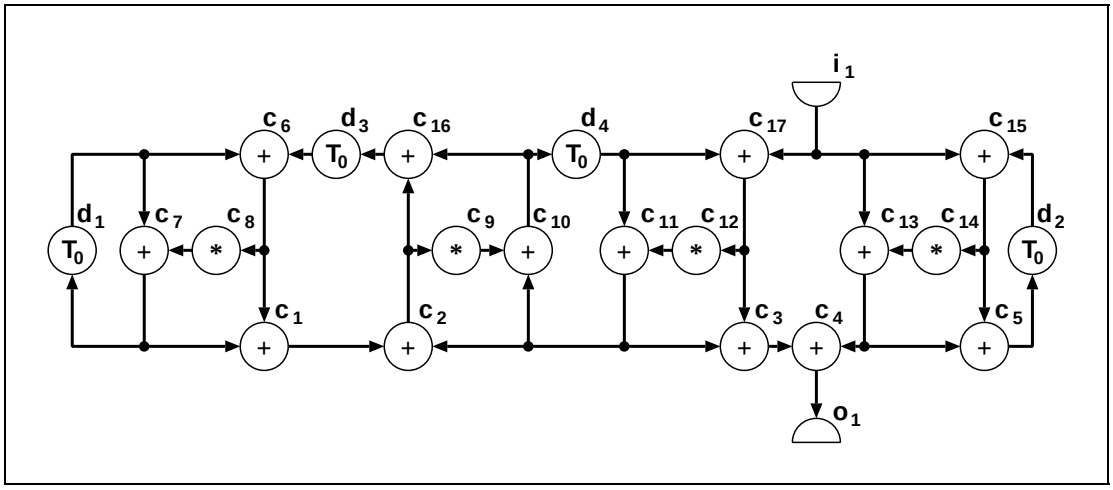

Figure 8.4: *The IDFG for the fourth-order Jaumann wave digital filter.*

Name	Description	Specification
Second	Second-order digital filter	Figure 8.1
Third	Third-order digital filter	Figure 8.2
Lattice	Fourth-order all-pole lattice filter	Figure 8.3
Jaumann	Fourth-order Jaumann wave digital filter	Figure 8.4
Elliptic	Fifth-order wave digital elliptic filter	Figure 5.5

Table 8.1: *The benchmark filters.*

DFG	Dur.		Critical path	Critical loop
	+	*		
Second	1	2	1-2-5	4-2-D
Third	1	2	1-2-11-12	10-11-D
Lattice	1	5	1-2-3-4-5-6-7-8-9-10-11-12	1-2-3-4-5-13-D
				4-5-6-7-8-14-D
				7-8-9-10-11-15-D
Jaumann	1	5	17-12-11-3-4	6-8-7-1-2-9-10-16-D
Elliptic	1	2	1-10-16-21-20-22-24-25-26-33-32	10-16-21-17-15-9-7-6-5-4-2-3-1-D
				10-16-21-17-15-9-7-6-8-12-13-14-D-11

Table 8.2: *The critical paths and loops for the five benchmark filters.*

The duration of a multiplication is not always the same. However, the scheduling problems are chosen such that for a each filter the multiplication has a fixed duration. This has the advantage that there is only one set of performance bounds for each filter. For each filter the duration of a multiplication can be found in Table 8.2. The table also shows the critical path and critical loops that determine respectively the performance bounds PDB and IPB. Note that two filters have more than one critical loop. The values for IPB, PDB and PB are shown in Table 8.3. To calculate PB, the number of operations is required. These values are therefore also given. The performance bounds are helpful to determine the quality of the schedules that are found.

The configuration of the scheduling method is the largely the same for all the problems considered in this chapter. It is shown in Table 8.4. The scheduling method has been implemented in Common Lisp [Ste90] [Kee89]. The results in this chapter have been obtained by executing the program in CMU Common Lisp on a HP 9000/735 server. It should be noted that the implementation of the scheduling method is a prototype. An optimized implementation is likely to have a faster execution speed.

DFG	Dur.		Num.		IPB	PDB	PB
	+	*	+	*			
Second	1	2	4	4	3	3	4
Third	1	2	6	6	3	4	6
Lattice	1	5	11	4	14	28	3
Jaumann	1	5	13	4	16	9	3
Elliptic	1	2	26	8	16	14	3

Table 8.3: *The performance bounds for the five benchmark filters.*

Genetic algorithm	Population size	60
	Crossover	80%, UNIPERM (copy bias=0.50)
	Mutation	0%
	Selection	tournament (size=6)
	Termination	margin=10%, fraction=90% *or* max. number of evaluations=5000
Heuristics	$T_{0,initial}$	IPB, unless stated otherwise.
Results	Number of runs	50

Table 8.4: *Configuration of the scheduling method for the problems in this chapter.*

	Num.	Dur.		Avg.	Avg.	Best Schedule				
	FUs	+	*	evals	runtime	IP		Latency		Specimen
Second	4	1	2	324	5s	3	1.00	3	0.98	See Figure 8.5 (a)
Third	6	1	2	406	11s	3	1.00	4	0.36	See Figure 8.5 (b)
Lattice	3	1	5	237	6s	14	1.00	29	1.00	See Figure 8.5 (c)
Jaumann	3	1	5	217	6s	16	1.00	9	1.00	See Figure 8.5 (d)
Elliptic	3	1	2	834	76s	17	1.00	14	1.00	See Figure 8.5 (e)

Table 8.5: *Scheduling results for the five benchmark filters when communication delays are zero.*

8.2 Negligible Communication Delays

In this section, communication delays are considered negligible. This means that the black-box heuristic is disabled. It also means that the distance matrix $\mathbf{D}_h$ in the global heuristic is effectively disabled because all distances are zero. The filters that are considered are the five benchmark filters listed in Table 8.1. For each filter, the number of FUs that it is allowed to use is equal to PB. Each FU supports addition and multiplication[1].

The scheduling results can be found in Table 8.5. The average number of evaluations and the average time per run, give an indication of the execution speed. Subsequently, the best schedule that was found is described. First, the minimal iteration period that was found is given, together with the fraction of runs that generated a schedule with a minimal T_0. Secondly, for the schedules that have a minimal T_0, the minimum latency that was found is shown. The fraction of runs that resulted in a schedule with a minimal T_0 and minimal latency is also given. Finally, the table refers to figure with an example of a best schedule that was found.

It can be seen that all except two performance bounds are met. The latency for Filter Lattice is one time unit longer than PDB. However, the schedule that was found is optimal. The three critical loops (see Table 8.2) cause that it is impossible to have $T_0 = IPB$ without delaying operation 12 for one time unit (see Figure 8.5 (c)). This results in the increase in the latency. A similar, but more complex, reasoning can be given for the problem with Filter Elliptic. It can be reasoned that a smaller T_0 is not possible without using an additional FU.

The table shows that for each of the scheduling problems, the minimal iteration period was found for every run of the GA. The only problem for which the GA had difficulty finding the

[1]This is the case for each example presented in this chapter. The reason for this choice is that restricting the functionality of the FUs makes a scheduling problem easier and less interesting.

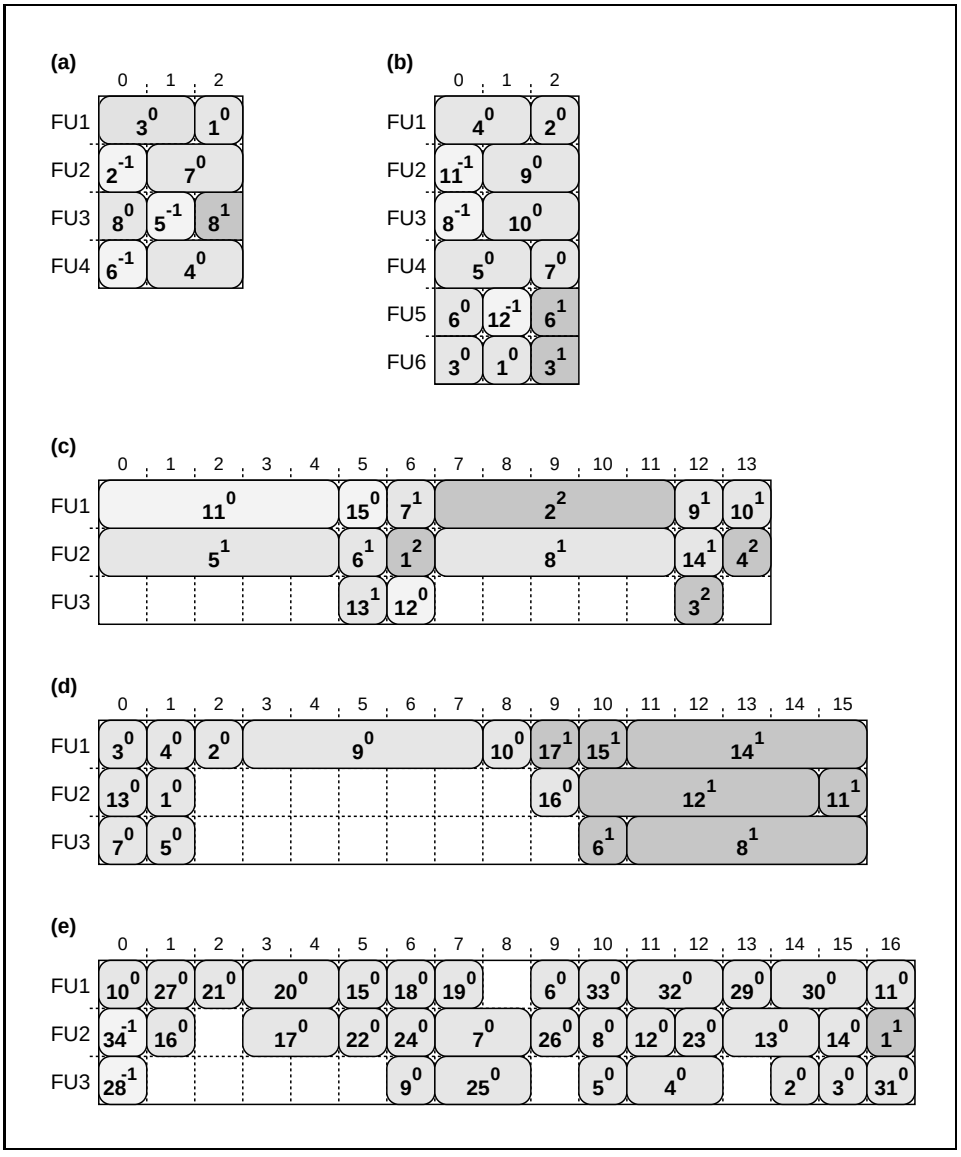

Figure 8.5: *Optimal schedules when there are no communication delays for: (a) Filter Second, (b) Filter Third, (c) Filter Lattice, (d) Filter Jaumann, and (e) Filter Elliptic.*

Name	Specification
Weak-chain-3	Figure 8.6(a)
Weak-chain-4	Figure 8.6(b)
Strong-chain-4	Figure 8.6(c)
Medium-ring-4	Figure 8.6(d)
Strong-ring-4	Figure 8.6(e)
Diamond-6	Figure 8.6(f)

Table 8.6: *The names of the benchmark multiprocessor structures.*

minimal latency, is for Filter Third. The best schedule was only found 36% of the time. A possible explanation is that minimizing T_0 and minimizing the latency are conflicting goals: there are not many schedules with a minimal T_0 that also have an optimal latency. Because the GA has been configured so that is converges quickly, it can get stuck in a local minima and not find the minimal latency.

It can be concluded that the quality of the solutions found by the scheduling method is good. The optimal solution is found for all five problems. The run time of the algorithm is a magnitude larger than that of problems dedicated to solve the scheduling problem for nonnegligible communication delays. For instance, in [HdGGH92] a run time of 0.33s is reported for the problem with Filter Lattice. However, it should be noted that the proposed scheduling method was not designed for this purpose, it is designed to tackle more detailed hardware models. Furthermore, when you realize that during a single run of the GA both heuristics are executed many times, the run time is even surprisingly small.

8.3 Black-box Communication Model

In this section, the black-box communication model is used. The six multiprocessor structures that are used are shown in Figure 8.6. For the sake of convenience, each structure is referred to by a short name. The names of each of the six multiprocessor structures can be found in Table 8.6. Fourteen different scheduling problems are considered. These problems are listed in Table 8.7. The problems have been chosen such that the difficulty varies from reasonably simple to hard.

The results are shown in Table 8.8. It can be seen that the minimum performance bounds are often not met. However, for most of the problems it is fairly easy to see that because of the restricted communication architecture, the performance bounds can not be reached. Whether the optimal schedule has been found for all problems remains to be seen, but so far no counter example has been encountered. Furthermore, when the schedules are examined closely, it follows that it is certainly not easy, or maybe even impossible, to improve upon the results reported here. The quality of the schedules that are found is very good, as is illustrated in Figure 8.7. The figure shows the schedules that were found for five of the more difficult scheduling problems. To illustrate the restrictions that precedence relations impose on the schedule, the most important loops have been indicated in one of the schedules (see Table 5.2 for a list of these loops).

The results show that probability that the scheduling method finds the best iteration period

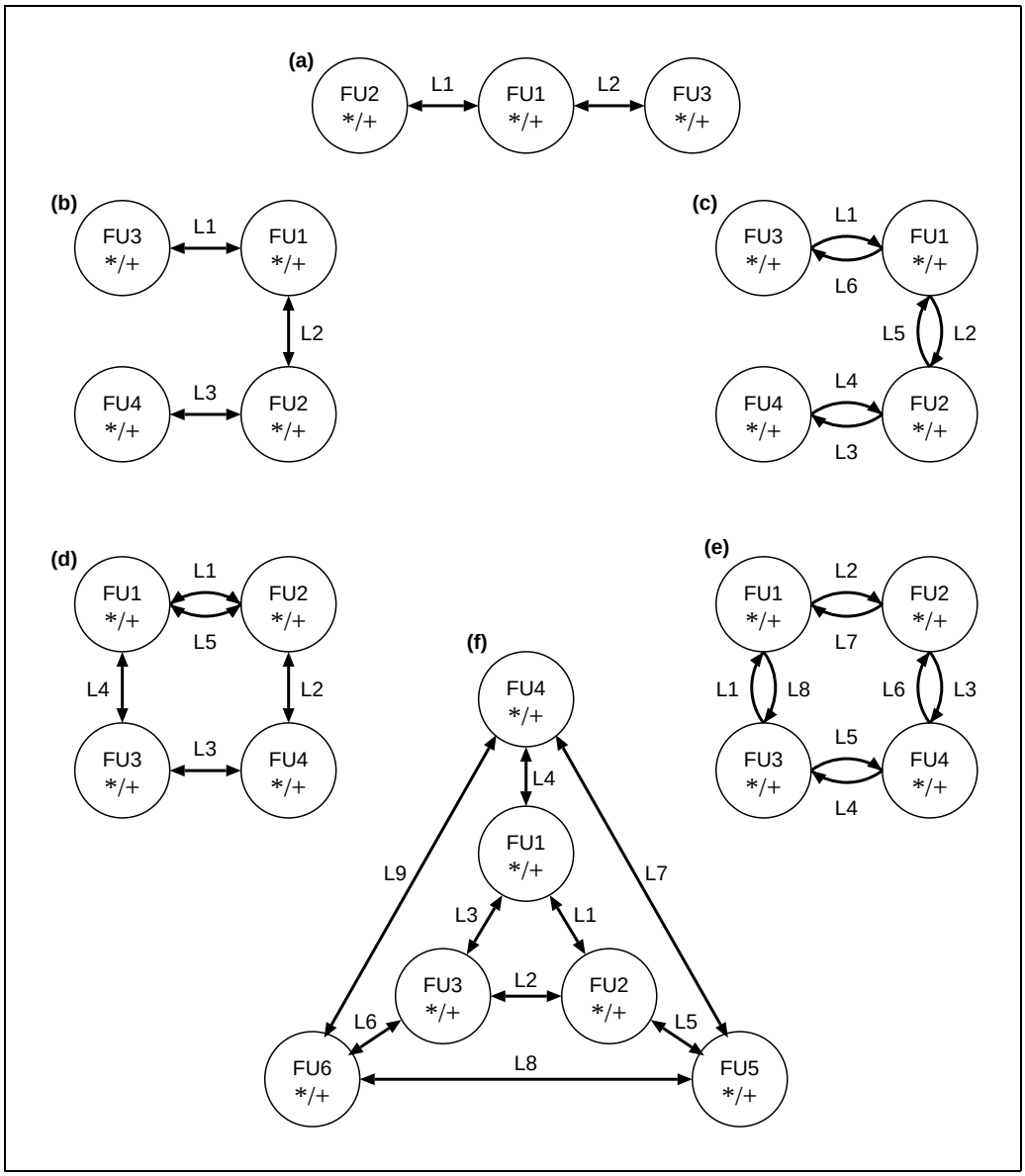

Figure 8.6: *The benchmark multiprocessor structures.*

Problem	Algorithm	Hardware			
		Structure	+	*	δ
A	Second	Strong-ring-4	1	2	2
B	Second	Medium-ring-4	1	2	2
C	Second	Weak-chain-4	1	2	1
D	Third	Weak-chain-4	1	2	1
E	Third	Weak-chain-3	1	2	1
F	Third	Diamond-6	1	2	1
G	Jaumann	Weak-chain-3	1	5	1
H	Jaumann	Weak-chain-3	1	5	2
I	Lattice	Weak-chain-3	1	5	1
J	Lattice	Weak-chain-3	1	5	2
K	Elliptic	Weak-chain-4	1	2	1
L	Elliptic	Weak-chain-4	1	2	2
M	Elliptic	Weak-chain-3	1	2	1
N	Elliptic	Weak-chain-3	1	2	2

Table 8.7: *The fourteen different scheduling problems.*

is not close to 1 for all of the scheduling problems. The probability that additionally the best latency is found is even smaller. It is expected, however, that a carefully designed fitness function can lead to a considerable improvement (see Section 7.4.4).

For three scheduling problems, $T_{0,initial}$ was chosen larger than IPB. This was done because it resulted in a considerable improvement in performance of the scheduling method (see also Section 7.2.1). It is expected that specifying $T_{0,initial}$ can be avoided by letting the scheduling method adaptively choose its value, as was recommended in the previous chapter.

It is useful to compare the run time for the scheduling problems in this section to the run time for the problems in the previous section, where the black-box hardware model was disabled. It follows that the increase in the run time is relatively small compared to the extra detail that is added to the hardware model. This is promising because it indicates the feasibility of supporting even more detailed hardware models by the proposed scheduling method.

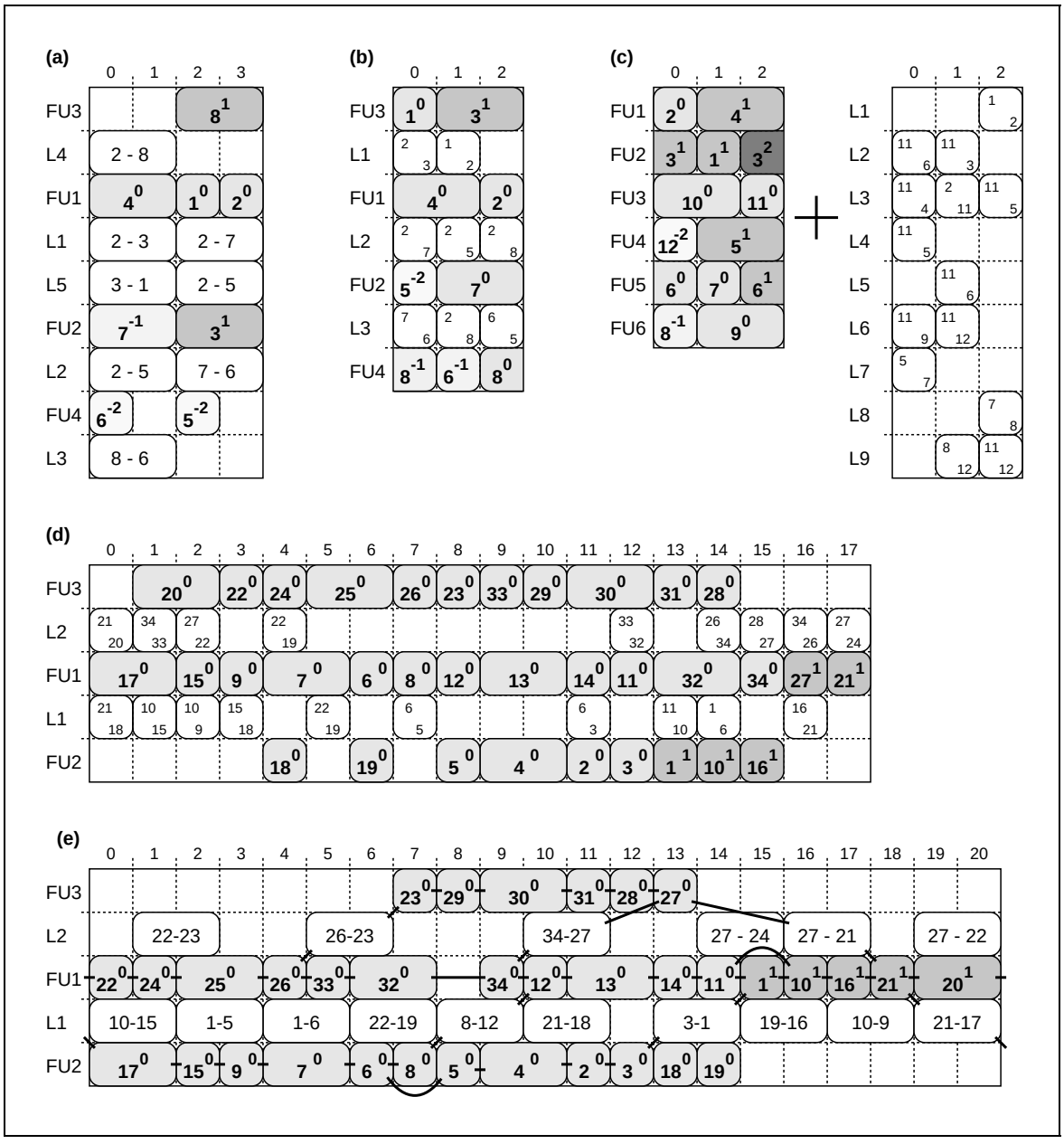

Figure 8.7: *Schedules generated for: (a) Problem B, (b) Problem C, (c) Problem F, (d) Problem M, and (e) Problem N (the most important loops are shown).*

	Avg. evals	Avg. runtime	Best Schedule IP		Latency		Notes
A	249	11s	4	0.98	6	0.44	
B	265	12s	4	0.20	9	0.20	
C	276	12s	3	0.56	7	0.56	
D	376	24s	5	1.00	5	0.12	
E	392	25s	6	1.00	4	0.08	
F	385	27s	3	0.54	9	0.14	
G	485	33s	16	1.00	9	0.94	
H	655	43s	18	1.00	13	0.98	
I	442	27s	16	0.98	30	0.62	
J	508	33s	18	0.98	31	0.16	
K	761	134s	18	0.84	16	0.60	(1)
L	734	129s	21	0.56	18	0.52	
M	709	111s	18	0.08	20	0.02	(1)
N	701	111s	21	0.50	14	0.04	(2)
Notes:		(1) $T_{0,initial} = 18 \neq IPB$					
		(2) $T_{0,initial} = 20 \neq IPB$					

Table 8.8: *The scheduling results for the fourteen problems based on the black-box hardware model.*

9

Conclusions and Recommendations

A scheduling method has been designed that can generate overlapped schedules for execution of fine-grain iterative algorithms onto multiprocessor architectures. The method has been designed such that it is able to support a wide range of hardware models. Furthermore, despite its generality, detailed hardware models can be supported.

This has been achieved by dividing the scheduling method into three layers. The top layer consists of a genetic algorithm which takes care of the optimization. It does not impose any restrictions on the hardware model: it only searches for permutations of operations used by the next layer. The middle layer contains the global scheduling heuristic. The global scheduling heuristic is based on a simple and general hardware model. Nevertheless, it is able to make good scheduling decisions and it can fully exploit intra- and inter-iteration parallelism. It uses an operation distance matrix to guarantee that precedence constraints will always be obeyed. The black-box heuristic can be found in the bottom layer. Only the black-box heuristic uses a detailed hardware model. When a different hardware model must be supported, only the black-box heuristic needs to be replaced. Implementing a new black-box heuristic is facilitated because it is possible to reuse a lot of the functionality provided by the global heuristic.

Both heuristics can insert cycles in the intermediate schedule. This ensures that the scheduling method never gets "stuck": when no resources are available or minimum communication delays are not met, cycles can be inserted to overcome the problem. The insertion of cycles in the schedule by the scheduling method has several important advantages:

- It results in a fast execution of the global heuristic and the black-box heuristic because it avoids that backtracking is necessary.
- It ensures that the genetic algorithm executes efficiently because every permutation of operations will lead to a valid schedule.
- It allows that the scheduling method can be used for a wide range of hardware models. Irrespective of the hardware model that is used, a black-box heuristic can be designed that always produces valid schedules.

Therefore, insertion of cycles is a very powerful utility. It should be noted that empirical data has shown that insertion of cycles has a disruptive effect on the quality of the schedules that

are produced. This, however, is not very serious because the disruptive effects are reduced considerably, when the global heuristic uses an initial schedule with a larger iteration period. It looks promising to include the initial iteration period in the genetic algorithm. It is certainly recommended to investigate this approach.

The scheduling method has been tested with a hardware model that includes communication delays and a communication network with a limited capacity. Empirical data shows that the quality of the schedules that are produced is good. For all the considered scheduling problems there are no signs which indicate that the best schedules that are found are not the optimal ones. However, not every run of the scheduling method produces the best schedule. Nevertheless, the scheduling method is able to produce good quality schedules in acceptable runtime.

A direction for future research is to experiment with other hardware models. Some possible extensions are to include the allocation of registers, pipelining and different communication architectures in the hardware model.

It is also highly recommended to examine whether a more sophisticated fitness function can improve the performance of the scheduling method. Based on the results presented in Section 7.4.3, it can be expected that a carefully designed fitness function can increase the probability that the GA finds a satisfactory schedule within a limited number of evaluations. Section 7.4.4 gave several suggestions of measures that can be included in the fitness function.

A

Notations

C	Set of computational nodes (operations) in an IDFG. See Page 6.
c.duration	Duration of operation c; $c \in C$. See Page 6.
c.fu	The FU that operation c is executed on; $c \in C$. It is chosen by the scheduling method. See Page 20.
c.starttime	Start time of operation c for iteration 0; $c \in C$. It is chosen by the scheduling method. See Page 20.
c.type	Type of operation c; $c \in C$. See Page 19.
D	Set of delay nodes in an IDFG. See Page 7.
d.multiplicity	Multiplicity of delay node d; $d \in D$. See Page 7.
$\mathbf{D}_c^{T_0}[c_1, c_2]$	Operation distance matrix; $c_1, c_2 \in C$. See Page 27.
$\mathbf{D}_h[fu_1, fu_2]$	Hardware distance matrix; $fu_1, fu_2 \in F$. See Page 19.
δ	Link communication delay (in TU). See Page 45.
E	Set of edges in an IDFG. See Page 6.
F	Set of functional units in a multiprocessor configuration. See Page 19.
fu.optypes	Set of operation types that the FU fu can handle; $fu \in F$. See Page 19.
I	Set of input nodes in an IDFG. See Page 6.
IPB	Iteration period bound. See Page 11.
IPBFP	Iteration period bound for a fixed number of processors. See Page 12.
O	Set of output nodes in an IDFG. See Page 6.
PB	Processor bound. See Page 12.
PDB	Periodic delay bound. See Page 12.
$P_s(R)$	Probability of success for R evaluations. See Page 62.

$R_c(fu)$	The range of valid start times when communication delays are not ignored and the operation is scheduled on FU fu; $fu \in F$. See Page 29.
R_{nc}	The range of valid start times when communication delays are ignored. See Page 28.
T_0	Iteration period. See Page 7.
V	Set of vertices in an IDFG. $V = C \cup D \cup I \cup O$. See Page 6.

Bibliography

BBM93a D. Beasly, D.R. Bull, and R.R. Martin. An overview of genetic algorithms : Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

BBM93b D. Beasly, D.R. Bull, and R.R. Martin. An overview of genetic algorithms : Part 2, research topics. *University Computing*, 15(4):170–181, 1993.

BG97 E.R. Bonsma and S.H. Gerez. Overlapped scheduling of fine-grain iterative dataflow graphs for target architectures with communication delays. In *Proc. of the 5th HCM BELSIGN Workshop*, Dresden, Germany, 1997.

BM93 T.P. Barnwell, III and V.K. Madisetti. The Georgia Tech digital signal multiprocessor. *IEEE Trans. on Signal Processing*, 41(7):2471–2487, July 1993.

Böc95 G. Böckle. *Exploitation of Fine-Grain Parallelism*, volume 942 of *Lecture Notes In Computer Science*. Springer-Verlag, Berlin, 1995.

CHK94 Y.Y. Chen, Y.C. Hsu, and C.T. King. MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures. *IEEE Trans. on VLSI Systems*, 2(1):21–32, March 1994.

CM94 B.A. Curtis and V.K. Madisetti. Rapid prototyping on the Georgia Tech digital signal multiprocessor. *IEEE Trans. on Signal Processing*, 42(3):649–662, March 1994.

CR92 D.C. Chen and J.M. Rabaey. A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, December 1992.

DAA94 M.K. Dhodhi, Imtiaz Ahmad, and Ishfaq Ahmad. A multiprocessor scheduling scheme using problem-space genetic algorithms. In *IEEE Conf. on Evolutionary Computation*, pages 214–219, 1994.

Dav91 L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

GB93 P.R. Gelabert and T.P. Barnwell, III. Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs. *IEEE Trans. on Signal Processing*, 41(2):858–888, February 1993.

Gol89 D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

HdG90 S.M. Heemstra de Groot. *Scheduling Techniques for Iterative Data-Flow Graphs.* PhD thesis, University of Twente, Enschede, 1990.

HdGGH92 S.M. Heemstra de Groot, S.H. Gerez, and O.E. Herrmann. Range-chart-guided iterative data-flow graph scheduling. *IEEE Trans. on Circuits and Systems*, 39(5):351–364, May 1992.

Hei96 M.J.M. Heijligers. *The Application of Genetic Algorithms to High-Level Synthesis.* PhD thesis, Tech. Univ. Eindhoven, Eindhoven, 1996.

Kaw92 T. Kawaguchi. Static allocation of parallel program modules onto a message passing multiprocessor system. In *International Symposium on Circuits and Systems*, pages 633–636, 1992.

Kee89 S.E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, Reading, Massachusetts, 1989.

KG95 M.S. Koster and S.H. Gerez. List scheduling for iterative data-flow graphs. In *GRONICS '95, Groningen Information Technology Conference for Students*, pages 123–130, The Netherlands, 1995. University of Groningen.

KKT90 K. Konstantinides, R.T. Kaneshiro, and J.R. Tani. Task allocation and scheduling models for multiprocessor digital signal processing. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 38(12):2151–2161, December 1990.

MC94 V.K. Madisetti and B.A. Curtis. A quantitative methology for rapid prototyping and high-level synthesis of signal processing algorithms. *IEEE Trans. on Signal Processing*, 42(11):3188–3208, November 1994.

PM91 K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. on Computers*, 40(2):178–195, February 1991.

Rad92 N.J. Radcliffe. Non-lineair genetic representations. In *Parallel Problem Solving from Nature 2*, pages 261–270. Elsevier Science Publishers, 1992.

Sch85 D.A. Schwartz. *Synchronous Multiprocessor Realizations of Shift Invariant Flow Graphs.* PhD thesis, Georgia Institute of Technology, 1985.

SSRM94 S. Selvakumar and C. Siva Ram Murthy. Scheduling precedence constrained task graphs with non-negligible intertask communication onto multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 5(3):328–336, March 1994.

Ste90 G.L. Steele Jr. *Common LISP, The Language.* Digital Press, Massachusetts, second edition, 1990.

Sys91 G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 21, pages 332–349. Van Nostrand Reinhold, 1991.