# Scheduling the Sequential Hardware-in-the-Loop Simulator

Master's Thesis
by

Sebastiaan B. Roodenburg

University of Twente, Enschede, The Netherlands
December 10, 2009

# Abstract

Rutgers [17] developed a hardware simulation environment, which incorporates time multiplexing to allow large hardware designs to be run completely in a FPGA. One of the subjects that required future work was the scheduler: the implemented round robin arbiter does not take any dependencies between cells into account. It was expected that a proper scheduler could drastically improve performance. In this thesis we developed a new scheduler for the seq_hils.

The DUV is partitioned by the developer into different cells. Similar cells are mapped onto a small set of hypercells. Since the cells are considered to be Mealy machines, each cells output depends on its inputs and its current state. Dependencies between cells consist of interconnections between cells and combinational paths through cells. An algorithm for deducting a complete dependency graph from a DUV was proposed.

The dependency graph will contain all possible data paths through the DUV. If all deducted data dependencies are taken into consideration for scheduling, we can construct a schedule offline which can guarantee stabilization of the system.

For the scheduling of the seq_hils we have implemented an hybrid online/offline scheduling approach. We chose for a computational intensive initial scheduling offline. This schedule will be optimized during simulation by a simple (low overhead) online scheduler.

We introduced two heuristical approaches for offline scheduling based on the dependency paths, which both have a low-order polynomial time and space complexity. This allows us to apply both heuristical approaches on large designs, and choose the schedule with the best makespan, within tenths of seconds. We showed that a worst-case schedule (i.e. a schedule constrained by all data dependencies) would require 3 to 10% less delta cycles to stabilize a system cycle than the round robin arbiter would require.

As an online optimization approach, we suggest to start with the worst-case offline schedule, and use runtime information about which cells are unstable, to skip over parts of the schedule if cells are already stable. Tests show that this approach can do another 5 to 10% percent performance increase on average. Based on these results we expected the final performance increase to be about 12 to 15%.

After implementing the new algorithms, two realistic designs were simulated with both the round robin and the new scheduler. The actual performance increase matches the expected performance increase based on the results. From this, we expect the 12 to 15% speed increase to hold for a large range of designs.

# Contents

CHAPTER 1

# Introduction

In hardware design, verification (often done by simulation) is an essential step before production. Since system designs tend to get larger [10] and more complex, simulation times grow excessively. Simulating large System-on-Chip designs using a software simulator is no longer feasible (running a simulation of a mid-sized design takes several hours). To speed up simulation Wolkotte and Rutgers [22, 17] have been working on a Sequential Hardware-in-the-Loop Simulator (seq_hils), which moves the simulation from software to a FPGA.

Using the repetitive nature of Network-on-Chip (NoC) and System-on-Chip (SoC) designs, simulation of considerably larger designs than would normally fit in the FPGA, get possible. The seq_hils maps similar pieces of the design onto the same area on the FPGA. By sequentially simulating the different parts of the design on that same part of the FPGA, very large designs can be simulated on a single FPGA. Wolkotte [22] achieves a speedup of a Network-on-Chip simulation with a factor 80 to 300 as compared to a software simulation, using a preliminary version of the seq_hils.

Sequentially simulating parts of the design, which are originally connected to each other, introduces some problems. Due to combinational paths in the design, several parts have to be evaluated multiple times within one system cycle in order to stabilize the entire design. Currently a simple round robin arbiter keeps scheduling unstable parts, until the entire design is stable, before evolving to the next system cycle.

## 1.1 Some background

When designing an application-specific integrated circuit (ASIC), verification of the design is essential before production. Due to very high production costs, we want to be sure that a design functions as specified, before it is produced as an ASIC.

For small ASIC designs, verification can be done by implementing the design in a Field-Programmable Gate Array (FPGA). A FPGA is an integrated circuit, which can be configured (programmed) to perform any function an ASIC could perform. Internally, the FPGA consists of a huge amount of logic blocks and flip-flops, which can be hierarchically interconnected via reconfigurable interconnects. Each logic block can be programmed with a look-up table to perform any logic operation possible.

FPGAs are very fast (although not as fast as an ASIC), and will behave exactly as an ASIC will (when using the same hardware description to program it). It is therefore a very powerful tool in the verification process before production. However, since the FPGA contains a limited number of logic blocks, flip-flops and interconnects (resources), some ASIC designs might not fit a FPGA. Falling back to software simulation for verification of too large ASIC designs is very undesirable.

In System-on-Chip designs, multiple interconnected cores are placed into a single chip design. Such SoC designs are often very large, and don't fit on a FPGA. However, it is quite commonly, that multiple identical (or very similar) cores occur within a SoC design. During the verification process, the speed of the final system is not a real issue. If we program cores which occur multiple times in the design, in a FPGA just once, we can greatly reduce the required resources on the FPGA. Depending on the design, we might even fit all unique pieces of hardware of a SoC design onto a FPGA. When we sequentially evaluate each core in the original design on that same piece of hardware in the FPGA, we can simulate very large designs completely on the FPGA.

This is what the seq_hils does. The seq_hils is a Java based tool chain, which transforms a large SoC design, into a functional identical design which requires less hardware resources. Currently the designer specifies which cores (*cells*) exist in the *design under verification* (DUV), and which cells are similar enough to be mapped onto the same piece of hardware. The tool constructs a *hypercell* for each set of similar cells. Each hypercell contains the logic function of each cell it embeds. The state (register values) and memory of all cells, still need to be duplicated for each cell.

The simulation environment for execution of the design on a FPGA consists of a set of hypercells for the design (packed into the *hypercell array*), a *state section* (the state of each cell) and a *data secion* (containing the memory from all cells). Together with some glue-logic (a *Central Controller* to monitor the state of each cell, a *Scheduler* entity to select cells for evaluation and a *Command Dispatcher* which is an external interface for the simulator), this simulator package can perform a cycle accurate simulation of the entire

DUV.

## 1.2 Problem description

The seq_hils simulates the entire design, by sequentially evaluating all so-called *cells* of the original design on a specific set of so-called *hypercells*.

The current implementation of the seq_hils uses an event driven round robin scheduling algorithm: whenever an input port of a cell is triggered, that cell is marked for (re-)evaluation. The simulation continues until all cells are stable. Combinational paths within and between cells form implicit dependencies between signals. Due to these dependencies, the order in which the cells are evaluated can have a huge impact on the overall performance of the simulator. Since no knowledge of dependencies is used in the current implementation, the simulation usually takes longer than necessary. Therefore, we want to find an improved scheduling algorithm for the seq_hils, with an high average performance.

The problem can be divided in to several sub-problems.

First of all, the data dependencies between cells are not all explicitly available. Data dependencies caused by combinational logic within cells form an essential part. Those combinational paths need to somehow be deducted from the *Design under verification* (DUV).

Offline scheduling is quite complex, but nonetheless will not result in a schedule with an high average performance: the actually dataflow on runtime has an huge impact on performance. Offline scheduling can not take this dataflow into account, with performance hits as a result.

On the other hand, online scheduling should be as simple as possible, to not cause unnecessary overhead on the runtime of the simulation process. Also, the area available on the FPGA is limited. The online scheduling overhead on the required hardware should also be as low as possible.

What combinations of online and offline scheduling approaches will lead to an overall performance increase with an as low as possible average runtime?

And finally, how are we going to test and verify the developed algorithms? No real-life designs are available which makes testing quite hard. Also: if the algorithms work with some real-life designs, will they work with (a large part of the set of) all designs?

## 1.3 Goals

During this masters thesis, we will work on building an improved scheduler for the seq_hils. This new scheduler should meet the following goals:

- The new schedule should use the knowledge of dependencies between cells, so that unnecessary re-evaluations of cells can be omitted, and the simulation process can be sped up.
- The focus should be on the performance increase of the total run time of the simulation process over many system cycles. An as low as possible average number of delta cycles per system cycle should be achieved.
- The scheduling process should be automated: no user interaction should be required. It should be part of the available tool chain constructed by Rutgers [17].

## 1.4 Approach

The first step in this project, is the development of an algorithm to deduct the data dependencies between cells in the DUV. These cells determine the structure of the design and are essential for scheduling.

In order to be able to test the algorithms which are to be developed, we start with setting up a test environment. The test environment consists of a simulator simulation environment (which is an evolved version of the simulation model introduced in the *Research in preparation of scheduling the seq_hils* [16]) which is able to simulate a simulation based on a dependency graph. In order to test to a wide variety of designs, a toolkit is developed to generate random structures.

For scheduling of the seq_hils we have developed an hybrid online / offline scheduling approach: an offline scheduling algorithm can order the cells based on the dependency graph. These schedules will not be optimal for average system cycles, but form a nice base for an online schedule. The online scheduler use the offline schedule, to do online scheduling decisions. Several algorithms are proposed. Different combinations of online and offline algorithms are tested using the simulation model. From this we got a good impression of how well the different algorithms will perform in the seq_hils. With this knowledge we selected which algorithms were to be incorporated into the seq_hils.

With the new scheduling algorithms implemented in the seq_hils, we have simulated two synthetic (but representative for realistic cases) test cases, to see if the expected performance increase is actually obtained.

## 1.5 Layout of this document

In chapter 2 we sketch the existing layout of the sequential hardware-in-the-loop simulator. We discuss the structure of the tool flow, and derive a formal model for the mapping from DUV to seq_hils. This formal model defines the properties which we base the scheduling of the simulator on.

A simulation environment for simulating the scheduling behaviour of the seq_hils (*metasim*) is presented in chapter 3. This chapter also discusses the deduction of a dependency graph from an existing design, and gives the algorithm for generating random structures which are used for benchmarking the scheduling algorithms.

In chapters 4 and 5 we discuss several approaches for offline and online scheduling. The discussed algorithms are compared to the round robin arbiter using the metasim simulation environment. The algortighms which are implemented are choosen based on these simulation results.

The implementation of the different algorithms into the seq_hils is discussed in chapter 6. The chapter finishes with two test cases which are simulated with the seq_hils using both the original round robin arbiter and the newly implemented scheduling.

The document will finish with some conclusions and recommendations for furture work in chapter 7.

CHAPTER 2

# Structure of the seq_hils

In order to gain some insight in the internals of the seq_hils, we will discuss its current design and give a formal description.

The seq_hils is a cycle precise simulator, which can simulate synchronous designs. The simulation itself is performed on a FPGA. In order to simulate large designs on the FPGA, time-multiplexing of hardware is implemented by the simulator. The design is split into cells, where similar cells are mapped onto a single hypercell in the FPGA. The FPGA sequentially evaluates cells on the hypercells and propagates values of changed output throughout the design until the system stabilizes. Currently, a round robin arbiter schedules unstable cells for evaluation. When the system is stable, a single cycle is simulated, and the simulator advances to the next cycle.

In the next sections the design will be described more thoroughly. In the simulator design, three levels of detail can be distinguished; we will describe them in a top-down order: from DUV-level to signal-level. But first we start with defining the timing involved in the simulation process.

## 2.1 Timing in the formalization

Since the design is considered to be synchronous, and the simulator simulates a complete clockcycle at a time (without considering inner-cycle propagation delays), a discrete simulation time $t$ is used to refer to a specific simulation cycle. We define $t$ as: $t \in T$ where $T = \{1, \ldots, N_T\}$. $N_T$ is number of cycles being simulated. Time $t$ is initialized to 1 and incremented once every simulation cycle.

In order to simulate the large design on the FPGA, the simulator time multiplexes the used hardware. Each simulation cycle $t \in T$ is split up into a variable number of delta cycles of unit length. This is specified formally as:

$$\forall t \in T | \mathcal{T}_t = \{1, \ldots\}$$

The time within a simulation cycle $t$ is discrete, given by $\tau_t \in \mathcal{T}_t$. At each increment of $t$, time $\tau_t$ is initialized to 1.

## 2.2 Design on DUV-level (cell level)

In the top level of the hierarchy, the design which should be simulated is located and named the *Design Under Verification* (DUV). The seq_hils is generated for a specific DUV, which will be referred to as design $d$. The DUV should be specified in synthesizable HDL: we assume a technology mapped netlist is available.

The designer specifies a partition of the original design. The (disjunct) parts of design $d$ are encapsulated in *cells*. We will refer to the set of cells as $C_d$, in which each element represents a cell in the DUV. If the partition is chosen such that more cells are alike, we need less hardware to map all the cells onto, and thus larger designs can be simulated.

At the wires where the original design is partitioned, cells have input and output ports. For simplicity we require that only directed signals are used in the design. For cell $c \in C_d$, we reference to its inputs as set $I_c$ and its outputs as $O_c$.

The inputs and outputs of the cells in $C_d$ are interconnected via wires, such that the connected cells form the original DUV. We will refer to those interconnections as $Con_d$, which is a set of tuples of inputs and outputs, or formally defined:

$$Con_d \subseteq \bigcup_{e,f \in C_d} O_e \times I_f$$

Constraints which should hold for $Con_d$ are: an input is connected to exactly one output, and an output can be connected to one or more inputs.

Cells in the design are considered to be Mealy machines, where the values of its outputs are directly dependent on both the cells internal state and its input ports. The internal state of a cell $c \in C_d$ is called its *state vector* and is referred to as $S_c$. The behavior of a cell is given by two functions which take the input and state values (at a certain time $t \in T$) as input: $\varphi_c$ gives the updated output values and $\psi_c$ gives the next state.

In the formalization, the ports and state of cell $c \in C_d$ can be dereferenced to obtain their values at time $t$, using an index: $I_c[t]$, $O_c[t]$ and $S_c[t]$.

As implied by the fact that we allow Mealy machines in the design, there might be a combinational path within a cell which causes an output to be directly dependent on one or more of the cell's inputs. We assume this dependency relation to be known (or deductible), and will refer to this relation as $Dep_c : O_c \rightarrow \mathcal{P}(I_c)$. Combinational cycles which cross the boundary of a cell are not allowed, but combinational cycles within a cell are allowed, as long as they do not oscillate. Sequential cycles within a cell, or over multiple cells are allowed.

## 2.3 Design on simulator-level (hypercell level)

In order to reduce the required hardware for the simulation, at compile-time (nearly) identical cells are mapped onto each other so time-multiplexing can be applied: those (nearly) identical cells are grouped into a hypercell $h \in H_d$. The set $\mathring{C}_h \subseteq C_d$ gives the set of cells which are embedded in hypercell $h$.

Multiple instances of the same hypercell can occur in the simulator. These multiple instances of the same hypercell should be identical, and will be grouped into a hypercell group. The set $G_d$ contains all (distinct) sets of (functional) identical hypercells. For a hypercell group $g \in G_d$, the set $\mathring{C}_g$ is the set of cells embedded in the hypercells in $g$, which should be equal to the $\mathring{C}$ sets of all hypercells in the group:

$$\forall h \in g | \mathring{C}_g = \mathring{C}_h$$

For each cell in $d$, there should be exactly one hypercell group in the simulator on which that cell can be evaluated:

$$\bigcup_{g \in G_d} \mathring{C}_g = C_d$$

$$\forall g, k \in G_d | g \neq k \rightarrow \mathring{C}_g \cap \mathring{C}_k = \varnothing$$

Hypercell groups $g \in G_d$ could be partitioned into $\alpha_g$ pipeline stages. The length of the pipeline $\alpha_g$ is measured in delta cycles. If the hypercell group is not pipelined, then $\alpha_g = 0$. It is not required that all hypercells groups have the same pipeline length.

## 2.4 Design on simulation-level (signal level)

The simulation process consists of sequentially evaluating the cells from the DUV on the available hypercells. Changes on outputs of cells are detected and propagated throughout the system in following delta cycles.

Each hypercell group $g \in G_d$ can start evaluating at most $|g|$ cells $c \in \mathring{C}_g$ each delta cycle. There is no additional constraint on which specific instance of a hypercell $h \in g$ a cell $c$ should be evaluated.

With the evaluation of a cell, we imply calculating the output values and the next state, given the current state and input signals:

$$O_c[t] := \varphi_c(I_c[t], S_c[t])$$

$$S_c[t+1] := \psi_c(I_c[t], S_c[t])$$

Note that the outputs $O_c t$ are connected via port memories to a set of inputs $I[t]$: any output can directly change one or more input ports of (different) cells.

If a hypercell group $g \in G_d$ starts evaluating cell $c \in C_d$ at time $\tau_t$, $c$'s updated output is – due to pipelining – available at time $\tau_t + \alpha_g$. A cell can be evaluated multiple times during a simulation cycle, but not at the same delta cycle (as stated by Rutgers [17], in order to prevent data hazards). Each evaluation overrides the previously stored results.

In order to determine which cells have to be evaluated, for each cell $c \in C_d$ we keep track of a dirty flag $D_c : \{false, true\}$ which marks the cell $c$ as being unstable[1]. Cells which have their dirty flag set to $true$, should be (re)evaluated.

Initially, at $t = 1$ and $\tau_t = 1$, all dirty flags are set: $\forall c \in C_d | D_c = true$. When evaluation of cell $c \in C_d$ starts, its dirty flag $D_c$ is set to $false$. Whenever cell $c$'s input changes, its dirty flag $D_c$ is set to $true$ again. As long as there are dirty flags set to $true$, the system is not stable.

If all dirty flags are set to $false$ and all hypercells are done evaluating cells (all pipelines are empty), the simulation of cycle $t$ is done, so the simulation time can evolve one cycle: $t := t + 1$. When the simulation time evolves, at least all cells $c \in C_d$ which have $S_c[t] \neq S_c[t-1]$, must have their dirty flag is set[2].

[1] In the current implementation, all inputs of a cell are marked with a changed flag. These changed flags are used to determine the dirty-state of a cell.

[2] But in the current implementation, all dirty flags are set at a simulation clock tick

CHAPTER 3

# Simulation model for the simulator

Since the performance of online scheduling algorithms depends on runtime behavior of the DUV, representative performance analysis cannot be done statically. Even the performance of offline scheduling algorithms cannot be compared to the round robin arbiter for the same reason. Therefore a custom (stand alone) simulation environment was developed from scratch, to assist in the performance analysis of the different scheduling algorithms which are to be developed.

The simulation environment provides a complete test bench, which consists of roughly two parts:

- A utility toolkit for both analyzing existing designs and generating synthetic structures. These structures will be interpreted as designs which are to be simulated.
- A simulation framework which simulates the execution and dataflow of cells inside the seq_hils. Into this simulation framework, several scheduling algorithms can be plugged in, in order to compare results.

Both parts will be discussed more thoroughly in the next sections.

## 3.1 Simulation structures

The seq_hils is generated for a specific DUV. In order to find a scheduling algorithm what will work for a broad amount of DUV's, the algorithms should be tested with a big variety of structures.

As discussed in chapter 2, data dependencies between and within cells exist. Due to those data dependencies, the current round robin arbiter needs to re-evaluate cells until the design stabilizes. Thus, those data dependencies that exist in the DUV, determine an implicit execution order of cells.

For simulation of the scheduling process, we do not need to generate a complete functional design. We do not need to know what happens to the data inside a cell, it is sufficient to know how the data flows. The structures which we generate for our simulation environment should represent the dataflows inside the DUV. We abstract the DUV to a dataflow graph with just the toplevel input and output ports of each cell as nodes. Connections between cells are retained, while all internals of the cells are replaced by simple edges which represent the combinational dependencies. We go deeper into the analysis of existing designs in section 3.1.1.

In order to test the scheduling algorithms as good as possible, they are tested to both real-life designs and synthetic structures. Section 3.1.2 will go deeper into the algorithms used to generate the structures used for testing.

### 3.1.1 Netlist analysis

As seen earlier, the data dependencies that exist in the DUV determine the implicit execution order of cells. However, these dependencies are not explicitly available. In order to schedule the seq_hils, an algorithm is developed for deducting the combinational paths through the DUV from the technology mapped EDIF netlist.

Those combinational paths are not only used in the simulation environment being discussed in this chapter, but they are also required to generate a schedule that can guarantee stabilization of the DUV for each clock cycle which is being simulated. We will go deeper into this in chapter 4.

#### Related work

The idea to use the netlist to deduct an execution order is not new. During the research topic in preparation of scheduling the seq_hils [16], several different simulation environments which are available [2, 3, 4, 12, 13, 14, 19, 21] have been discussed. All examined simulators which do some sort of offline scheduling, use knowledge of the dataflow to perform the scheduling.

Simulators which do fine grained simulation on port level [2, 4, 19, 21], can use the netlist (which is already a directed graph) to do the scheduling. A simple topological sort of the netlist (a linear ordering of the nodes in which each node comes before all nodes to which it has outbound edges) is enough to deduct the order in which all ports have to be evaluated.

Simulators which do simulation on a coarser grained cell level [3, 12, 13, 14], also use the connections between cells to do some ordering. However, all simulators which were looked at that used this approach, assumed the cells were black boxes and thus did not use its internal structures. Some simulation environments [3, 13] required the designer to annotate which combinational paths inside cells exist. One reviewed simulation environment [12] assumed all inputs and all outputs of each cell were connected when no annotations were added. This is a worst-case scenario. However, if this introduces combinational cycles (e.g. where a cell's input (in-)directly depends on its own output), an offline schedule can no longer be constructed. If no combinational cycles are introduced, the resulting schedule will be valid, but might be far from optimal (due to the pessimistic assumptions).

Since we have a complete technology mapped netlist available, which could be simulated at port level, it should be possible to deduct combinational paths at cell level from it. The virtual wires emulation system [19, 21] suggest that they deduct data-flow from the netlist, and use that for scheduling. However, an exact algortihm on how this is done, is not given.

### Constructing a dataflow graph from the DUV

The DUV we want to simulate consists of interconnected cells. Both those interconnections and the combinational paths through the cells determine the dependencies between cells: the data which flows through the connections need to flow in that order to produce correct results. When we can deduct a dataflow graph from the DUV at cell level, we have a clear overview of which dependencies between cells exist. In order to be able to construct an offline schedule from the dataflow graph, the DUV should met several requirements:

- The DUV should not contain combinational cycles which cross the boundary of a cell (i.e. the dataflow graph at cell-level should be cycle-free)
- Tri-state ports are not supported: all edges in the dataflow graph should be unidirectional
- The DUV should not contain a cell which oscillates (e.g. the output of such a cell is non-deteministic for a certain clock cycle)

The dataflow graph from the DUV at cell level is a directed acyclic graph, in which nodes represent the top-level input and output ports of all the cells in the DUV, and the edges represent the interconnections (and thus the possible flow of data) between and through the ports of the cells.

Since we do a cycle-accurate simulation of the DUV, we only need the dataflow graph to represent the dataflow within a clock cycle. Data which is synchronized by a register stops flowing until the system is stable and the next clock triger is received. Thus, synchronized dataflow can be omitted in our analysis (i.e. we need to stabilize the combinational logic within a system clock cycle before we can continue simulation).

As already noted in the description of the structure of the seq_hils, for the design of the simulator we assume the cells are Mealy machines. This implies the DUV could contain asynchronous combinational paths between input and output ports of a cell: data presented at an input port, might cause an output port to be triggered within the same clock cycle.

The dataflow graph is constructed from the EDIF netlist. Since we execute a cell at once, we are not interested in the internal structure of each cell. The dataflow graph consists of at most all input and output ports of each cell as nodes. Combinational paths through a cell are mapped to single edges in the dataflow graph.

Furthermore, the DUV consists of a number of interconnected cells. These cells have those interconnections between their top-level input and output ports, and are explicitly available. All these interconnections form the remaining part of our dataflow graph.

So, the construction of the complete dataflow graph consists of three tasks:

- Insert the top-level input and output ports of each cell as nodes in the dataflow graph
- For each interconnection between cells, insert an edge in the dataflow graph
- For each combinational path through a cell, insert an edge in the dataflow graph

The first two tasks are trivial. Retrieving all combinational paths within cells is a bit more complex and will be discussed further.

### Finding combinational paths through a cell

Finding a combinational path through a cell is implemented using a breadth-first search graph traversing algorithm over the technology mapped EDIF netlist of a cell. We start searching from a input node, and traverse the graph. If the path hits a flip-flop element (which synchronizes the design), we will cut-off the traversal algorithm for that node. If we find a flip-flop free path to an output node, we have found a combinational path through the cell, and can add an edge for that input-output pair to the dataflow graph.

A more formal notation of this algorithm is given in Algorithm 3.1. It gives a standard breadth-first search implementation with an extended cut-off condition: break the search when we encounter a flip-flip.

```
input: Top-level input port $p$ from where we start searching for
       combinational paths

todo ← [ $p$ ]
visited ← [ $p$ ]

while todo $\neq \emptyset$ do
    $current$ ← head(todo)
    todo ← tail(todo)

    if isToplevelOutput($current$) then
        storeDependency($p$, $current$)
    else if ¬isFlipFlop($current$) then
        foreach $n \in$ connectedNodes($current$) do
            if $n \notin$ visited then
                todo ← todo + [ $n$ ]
                visited ← visited + [ $n$ ]
            end
        end
    end
end
```

Algorithm 3.1: Finding combinational paths through a cell

This procedure is repeated for each input port of each cell in the DUV in order to complete the entire dataflow graph.

Figure 3.1 gives a small example of how a netlist of a simple design, can be transformed into a dataflow graph. We can see a design consisting of two cells. All top-level input and output ports of the cells, are represented by nodes in the dependency graph (and are annotated with the cell they originate from).

### Collecting dependency paths

Once we have constructed a complete dataflow graph from the DUV, we can use that to deduct all data dependency paths for the entire design. Using these dependencies, we will construct a schedule which can guarantee stabilization of the DUV for each clock cycle which is being simulated.

As noted earlier, the dataflow graph is a directed acyclic graph. A data path is a path over a flowgraph from its source node to a sink node. Each node on the path gives a cell which processes the data. The concatenated sequence of cells forms a single dependency path.

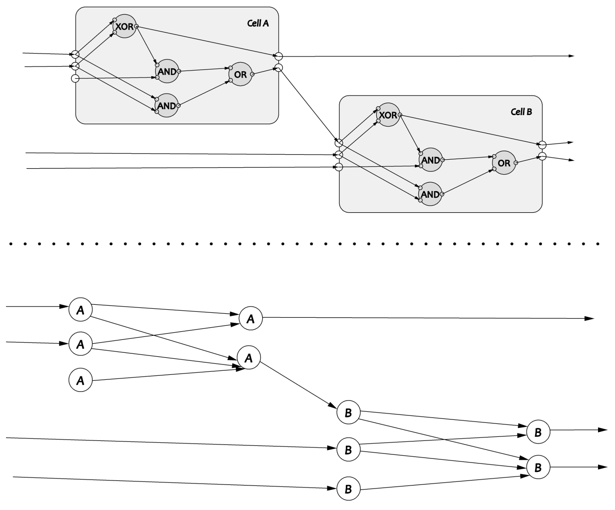


Figure 3.1: Sample showing the relation between the EDIF netlist and the dataflow graph

Collecting all the dependency paths in the DUV is a matter of finding all paths from a root node to all of its leaf nodes, for each tree in the dataflow graph. We use a depth first search for each root node (a node which has no incoming edges) to do so.

During storage of a data dependency path, we apply two additional optimizations, specialized for the scheduling problem.

Firstly, if we walk in the dependency graph over a cell, we might visit both the input and the output port of that cell (e.g. if the edge we follow in the dependency graph represents a combinational path through a cell in the DUV). Since the edge from an input to the output port does not require an additional execution of that cell, we do not want to duplicate that cell in our dependency path. In the current implementation we use a slightly looser condition: two successive cells in a dependency path should be unequal. This is equal to the previous mentioned condition, if no direct connection between a cells output and one of its inputs exist.

As a small example of this optimisation, figure 3.2 gives a design consisting of three cells. A dependencypath from $C$ through $B$ to $A$, has two nodes in cell B. However, we store the path as just $C \rightarrow B \rightarrow A$, since the two nodes in $B$ do not require two seperate executions of $B$ to resolve the dependeny.

Secondly, we store the data dependency paths in a set. This way, we only find unique dependency paths (e.g. a 32bit wide bus will not end up as 32 similar dependency paths in our collection).

```
traverse($n, stack$) :
input: Start node $n$ from where we should traverse all paths
input: Stack $stack$ constains the path so far
begin
    if outgoingEdges($n$) $= \varnothing$ then
        storePath ($stack$)
    else
        foreach $m \in$ outgoingEdges($n$) do
            traverse($m, stack + [m]$)
        end
    end
end

foreach $n \in$ nodes do
    if incomingEdges($n$) $= \varnothing$ then
        traverse($n, [n]$)
    end
end
```

Algorithm 3.2: Collecting dependency paths

### Analyzing a Network-on-Chip

In order to verify that the collecting of dependency paths works correct, an average-size real-life design was used as a small case study. The design evaluated is a Network-on-Chip (NoC), consisting of 12 GuarVC routers in a torus topology. The 12 routers are in a $4 \times 3$ grid, and due to the torus topology, all routers are connected to their 4 neighbors: there are no 'edges' in the network. The design contained only these routers: there are no processing elements connected to the routers.

The design was supplied as a large technology mapped EDIF netlist, and a partitioning was also supplied: the 12 routers form the 12 partitions, which should be mapped onto a single hypercell. Each router consists of roughly 63.000 primitive cells and 95.000 connections; for the entire NoC, that gives approximately 750.000 primitive cells and 1.2 million interconnections.

The netlist was processed using a simple implementation of the algorithm. From this evaluation we can see that in each cell of the NoC, there are about 26 combinational paths (which might have a width of several bits), connecting 5 inputs to 26 outputs. A simplified representation of a cell is given by figure 3.3.

We can see that there are many short combinational paths in the entire design. The longest path consists of three edges. Most of the combinational paths are between a router and its neighbors. In this specific design, we also see that one specific router controls the reset pins of the other routers.

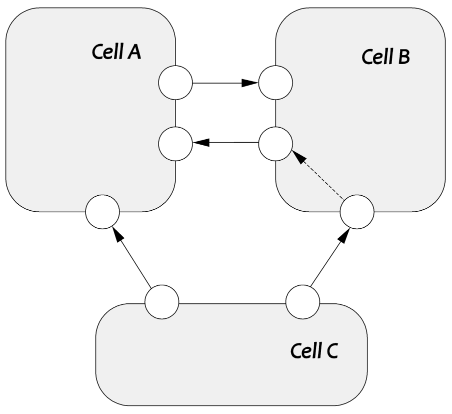


Figure 3.2: A sample design with several data dependencies

### 3.1.2 Structure generation

Since real-life examples which are usable for testing the scheduling algorithms are not excessively available, several procedures were written to generate synthetic test cases to assist in the evaluation of the scheduling algorithms.

Since the simulation only uses the resolved combinational interconnections between cells and within cells, we can quite easily generate non-behavioural test cases which satisfy a specified set of constraints. The synthetic test cases can be either regular or random structures. The regular structures which are generated, are based on the previously inspected network on chip designs (either mesh or torus topologies of GuarVC routers without processing elements). The random structures are actually constructed by randomly adding connections while enforcing the specified constraints.

#### Regular structures

The regular structures which are generated are based on the analyzed Network-on-Chip design. Each node in the network represents a router without a connected processing element. Each node is connected with its 4 neighbors: each connection is a combinational path to that neighbor and back again. Figure 3.3 shows the generated structure for four cells.

Depending on the topology, not all nodes have four neighbors. The toolkit can generate networks with either a mesh or a torus topology. In a torus topology, all nodes have exactly 4 neighbors. In a mesh topology, some nodes lay on an edge.

We will refer to these structures as either *Mesh*$M \times N$ or *Torus*$M \times N$ models,

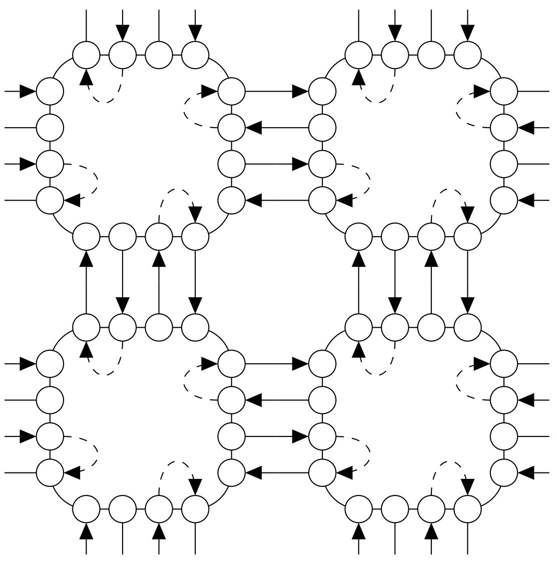


Figure 3.3: Simplified representation of data paths within and between cells in a NoC

where $M \times N$ represents the size of the network (ie. the number of horizontal and vertical nodes in the grid).

In the analysed GuarVC network, we found that one node has some sort of 'master' role: it controls the reset port of all other nodes. This adds some additional dependency paths from this node to each other node. We additionaly generate a torus topology for this structure, and will refer to it as a *CrispM*$\times$*N* model.

### Random structures

The testing process can not be based solely on regular structures, since it will bias the results: if a heuristic works well for a certain structure, it will probably also perform quite well on a whole family of similar structures but not necessarily on other designs. Therefore it is desirable to generate random structures, so testing can be performed on an as wide as possible spectrum of possible input parameters.

The random structure will be generated based on several given parameters:

- the number of cells in the generated structure
- the number of ports per cell (each cell will have the same number of ports; half of the ports will be input ports, the other half will be output ports)

- the average number (and a tolerance) of outgoing connections per cell
- the average number (and a tolerance) of combinational paths within a cell

Note that these parameters are global: the given averages per cell are the average over the entire structure.

To allow for a bit more diverse random structures, both the number of connections and the number of combinational paths are flexible. The two parameters (average and tolerance) determine the range of allowable connections and combinational paths per cell: the exact number will be in the range $[(average - tolerance), (average + tolerance)]$.

The number of ports, together with the number of connections, determines the density of the interconnections.

For the generation of connections between ports, we use an iterative process: we keep adding connections until we reach a situation were all parameters are met. For each step, we also check if the added connection does not invalidate one of the given additional constraints:

- An added combinational dependency should not introduce a cyclic dependency
- An added combinational dependency should not introduce combinational paths which are longer than $x$ steps (where $x$ is also a parameter for the generation process)

A random structure is generated using a pseudo-random number generator, for which its seed is initially set to a specific identifier. This way, the generated structures can be reproduced based on this identifier and its parameters.

## 3.2 Design of metasim

The design of the simulator is based on the concept of Monte Carlo simulation [9]. Monte Carlo simulation is usually used in cases where simulating all possible input combinations would be impossible (e.g. due to enormous state space or nondeterministic behavior). By simulating enough random cases, the average outcome will converge to a realistic result. Since the behavior of the cells is unpredictable, and precise results are not required, the Monte Carlo approach seems suitable.

Cells are simulated using a non-behavioural stochastic model. Two probabilities $Prob_1$ and $Prob_2$ define the behavior of the cell; note that those probabilities are global and thus equal for all cells. $Prob_1$ gives the chance of a combinational path changing an output when an input changes. $Prob_2$ describes the probability of a changing output when the internal state changes. Assumed is that the internal state always changes after each system clock tick, and that the propagation of the internal state is combined with one evaluation (as in the current seq_hils implementation). Changed outputs mark inputs of the connected cells as dirty, and thus mark such a cell for re-evaluation. A random number generator is used to enforce the probabilities during simulation.

The meta-simulator has two additional parameters: *Alpha* which gives the pipeline depth of all hypercells, and *Parallel* gives the number of hypercells in the simulator. For simplicity, it is assumed all cells can be evaluated on all hypercells, and the pipeline depth for all hypercells is equal. This is as if the simulator contains only identical cells, which are grouped into a single hypercell group.

For the simulation of the individual cells, we use a stochastic model. Two steps can be identified in the simulation of a cell: *starting* evaluation of a cell, and *finishing* evaluation of a cell. By splitting the simulation in two parts, we can delay the time between start and finish as long as we want, and thus simulate pipelining. For the pipelining, we define a queue for each hypercell in our simulation model.

When we start simulating a cell, we copy all dirty flags for the input ports and the state into a local copy, after which we reset all the dirty flags. The local copy of the dirty flags is placed into the front of a pipeline queue for later use.

To finish simulation of a cell, we pop the dirty flags from the pipeline queue, and use that to trigger connected cells:

- For each combinational path from input port $i$ to output port $o$, if $dirty_i$ is set then, draw a random number $0 \leq r \leq 1$ and if $r < Prob_1$ trigger output $o$.
- If the state was dirty, then for each output port $o$, draw a random number $0 \leq r \leq 1$ and if $r < Prob_2$ then tigger outpur $o$.
- For all triggered outputs, trigger the input ports which they are connected to

The simulation of a single data cycle, consists of:

- Selecting a cell to schedule for each hypercell
- Start evaluation of all selected cells

- Finish evaluation of all cells which are done (depending on the pipeline length)

The meta-simulator schedules evaluation of the cells, using the selected algorithm, until all cells are stable. The number of delta cycles until stabilization is counted. To get somewhat more representative results, each simulation consists of at least 100 system cycles and is repeated at least 50 times. This is to even out the influence of the random number generator.

## 3.3 Testing framework for algorithms

As noted before, the simulation environment consists of a scheduler simulator in which several scheduling algorithms can be plugged in, and a structure generation toolbox.

A single structure can be simulated multiple times with identical impulses and configuration, but with different scheduler instances. This way, schedulers can be compared as good as possible.

The plugging in of different schedulers into the simulation environment goes via an abstract class which can be overridden by different implementations. The instance of a scheduler gets initialized once before the start of a simulation. It receives a reset event at the start of a system cycle, and a scheduling request for each hypercell at each delta cycle. This closely matches the required behavior in the seq_hils.

An array of instantiated schedulers is used to schedule each generated structure: for each structure, each scheduler is requested to perform a complete simulation (consisting of several system cycles, which is repeated a number of times). The framework gathers statistics (required number of delta cycles until stabilization) for each scheduler.

The framework generates a number of regular or random structures which are to be scheduled in a loop. Parameters for the generated structures are predefined. This way, large tables with scheduling results can be generated quite easily.

For a broader comparison of the scheduling algorithms, the simulator can load or generate different hardware structures using the algorithms described in section 3.1. Doing the construction of a structure and the simulation of multiple algorithms inside a loop, provides a convenient automated testing framework which allows for easy gathering of large amounts of simulation results.

## 3.4 Metasim results

The metasim simulation envronment was used to benchmark several online and offline scheduling algorithms. Results obtained for specific algortihms will be discussed in sections 4.6 and 5.2.

If we compare the results obtained from simulation with metasim and the final implementation as described in section 6.3, we see that our metasim simulation results are comparible with results from the seq_hils. We can thus conclude that our proposed stochastic model of the system matches the real world close enough to provide usable figures.

CHAPTER 4

# Offline scheduling approaches

In this chapter the offline scheduling algorithm for the seq_hils will be discussed.

## 4.1 Introduction to offline scheduling

In general, scheduling algorithms describe how a set of tasks or jobs should be assigned to a set of machines (resources). Jobs have a certain length (i.e. the processing time it takes to complete a job), and only one job can be run on a machine at a time. Sometimes, jobs have additional constraints, like a release date (the earliest time at which can be scheduled), a due date (the latest time at which a job should be finished) or precedence constraints (in case a job depends on other jobs to be finished, before it can start). The time it takes from the start of the first job until the last job is finished, is called the *makespan* of a schedule.

In the simulator, the evaluation of the cells can be considered as the jobs to be scheduled, and the available hypercells are the machines to which the jobs can be assigned. The jobs in the simulator have a predefined run time, and can not be preempted. As discussed earlier, there is also a notion of dependencies between jobs: due to wiring, some jobs depend on others. Combinational paths through cells introduce more dependencies.

The evaluation time of a job does not completely match the runtime of a job. When evaluation of a cell starts, it takes a number of clock-cycles (because of pipelining) until the results are available. This evaluation time, is required for correctly enforcing precedence constraints. However, the hypercells can start evaluating one cell each clock cycle, as if the job occupies the machine for just one time unit.

The scheduling algorithm which will be chosen for this project will (initially) be used to schedule a single system cycle. Since the seq_hils will simulate many system cycles, and we want to speed up the entire simulation process, our scheduling goal is to minimize the average makespan of the schedule.

## 4.2 Why an offline schedule?

Generating an optimal schedule is often very hard. It is therefore undesirable (or even impossible) to do so online. Offline we have often more computational power available to find a (near) optimal schedule. Furthermore, finding an optimal schedule only has to be done once, and can be used for every system cycle.

The ability to spend some offline time on finding a (near) optimal schedule has some nice advantages.

If we construct our scheduling based on all possible data dependencies in the DUV, we can guarantee that our system stabilizes using that schedule: if there is a data flow in the DUV which is not captured by our schedule, that data dependency was not in our dependency graph, and thus our data dependency graph did not contain all possible data dependencies.

The (minimal) makespan of the schedule which embeds all data dependencies, gives an upper-bound for the run-time required for evaluating a system cycle in the DUV.

However, many of those combinational dependencies are not hard constraints on runtime: it is not certain that a combinational path will cause an output port to change.

Constructing a schedule based on a dependency graph which excludes all combinational paths though cells, gives a lower-bound for the makespan: a best-case schedule in case none of the combinational paths trigger re-evaluation of connected cells. However, this is a weak lower-bound, since it might be that outputs do not change at all at an evaluation, and thus direct connections between cells do not cause a re-evaluation either: it might be that some system cycles stabilize much faster than the best-case schedule suggests.

## 4.3 Finding optimal schedules

As an initial approach, in order to set some boundaries for further heuristic approaches, we have tried to find optimal schedules for some smaller regular designs. Initially, to slightly simplify the finding of schedules, we will schedule systems with a single hypercell without pipelining.

The process of scheduling consists of picking a cell to evaluate each delta cycle until the system is stable. Unfortunately, the scheduling problem is quite hard: we can not constructively build an optimal schedule. Only when we have found a schedule, we can determine if it is better than previous found schedules, but we do not know whether we have constructed an optimal schedule. In order to find an optimal schedule, an exhaustive search over the state space has to be performed, before we know we have actually found a minimum.

Since the state space grows exponentially when the size of the design increases, some smart tricks have to be implemented to keep the number of examined states as low as possible while ensuring an optimum will be found.

### 4.3.1 Integer Linear Programming formulation

The first approach for finding an optimal schedule is specifying the problem using an Integer Linear Programming formulation. Although solving ILPs, in general, is NP-Hard, several effective heuristically approaches for solving them are implemented in (highly optimized) tools which are available.

In order to be able to use several tools, a more constraint variant of an ILP, namely a Binary Integer Programming Formulation, is being used to formalize the problem. Therefore we have to introduce binary variables to describe the problem. We want to schedule a set of cells, based on a dependency graph consisting of top-level ports of those cells.

The set of cells we want to schedule is $C$, each individual cell $c_j \in C$ has a number of binary variables to indicate when it should be scheduled: $c_{j,t}$ indicates that cell $c_j$ should be scheduled at each $t$ where $c_{j,t}$ has value 1. A cell can be scheduled multiple times, but should be scheduled at least once.

The set of ports in the dependency graph is $P$. Each port in the graph can be referenced as $p_i$. A port should be scheduled at a certain time, therefore a variable $p_{i,t}$ is introduced: at the time $t$ when $p_i$ is scheduled, variable $p_{i,t}$ is 1. At all other times $t$ $p_{i,t}$ should be 0. Each port $p_i$ should be scheduled exactly once.

In order to enforce the constraints, some relations are needed. The relation $Cel : P \rightarrow C$ maps each port to the cell by which it is evaluated. The set $Constraint : P \times P$ contains tuples, defining the precedent constraints between ports in the dependency graph.

The timing in the formilisation is defined in an integer range starting from 0, and is limited by an upperbound H. The upperbound $H$ could be a weak upperbound, but it is required to limit the statespace (which would be unlimited if the timing was unbound).

We would like to minimize the makespan $z$. The entire ILP formulation looks like this:

`minimize:`

$$z$$

`subject to:`
(makespan should include all evaluations cells:)

$$z \geq t * c_t \quad \forall c \in C, t \in 1 \ldots H$$

(schedule each port exactly once:)

$$\sum_{t=0}^{H} p_t \quad = \quad 1 \qquad \forall p \in P$$

(only one cell at a time:)

$$\sum_{c \in C} c_t \quad \leq \quad 1 \qquad \forall t$$

(a port only if a cell is scheduled:)

$$p_t \quad \leq \quad Cel(p)_t \qquad \forall p \in P, \forall t$$

(enforce predecent contraints; port i should be scheduled before port j:)

$$\sum_{t=0}^{H} tp_t \quad \leq \quad \sum_{t=0..H} tp'_t \qquad \forall (p, p') \in Constraint$$

(enforce binary variables:)

$$p, c \in 0, 1 \quad \forall p \in P, c \in C$$

This ILP is initially implemented using the `lp_solve` 5.5 library. Unfortunately did not work very well: only a Mesh2x2 was schedulable in a reasonable amount of time. Scheduling larger designs take several hours or run out of memory. Using the AIMMS program showed similar results. Due to disappointing results, the ILP approach was discarded.

Since the ILP approach did not provide any usable results, a specialized algorithm for finding an optimal schedule was developed in several iterations.

### 4.3.2 Brute force

Taking a brute force approach for finding schedules is not a good idea due to the huge state space of the scheduling problem. However, it provides an initial framework on which smarter algorithms are based.

The brute force approach consists of iterating over each possible schedule. Since there is an infinite number of schedules, we should pick an upper bound for the makespan: we should not try longer schedules. The upperbound does not have to be a hard upper bound; it can be adjusted depending on the schedules found: if no schedules can be found for a certain upperbound, we should increase it. If a schedule is found, the makespan for that schedule can be used as upperbound; it is useless to look for longer schedules.

A recursive algorithm, as outlined in algorithm 4.1, can be used to search for a schedule.

```
doSchedule($t$) :
input: Time $t$ is the deltacycle we are about to schedule
begin
    if Schedule fulfills all dependencies then
        if $t < Makespan$ then
            $Optimal \leftarrow Schedule$
            $Makespan \leftarrow t$
        end
        return
    end
    if $t > Upperbound$ then
        return
    end
    foreach $c \in$ cells do
        $schedule[t] \leftarrow c$
        doSchedule($t + 1$)
    end
end
```

Algorithm 4.1: Brute-force approach for finding an optimal schedule

This algorithm iterates over all $cells^{upperbound}$ possible schedules. The *is-Done* functions checks whether the constructed schedule fulfills all dependency constraints; if that is the case, we could cut-off this branch in the search.

### 4.3.3 Backtracking

The average performance of the brute-force implementation can quite easily be increased, if we check for a valid schedule every time we schedule a cell (instead of when we are done). This way we can usually cut off large branches in the search tree. Although backtracking greatly increases performance on average, in a worst-case scenario the performance is still as bad as the brute-force approach.

Scheduling a cell is only useful, if scheduling that cell would resolve at least one dependency constraint. This way we can constructively search for good schedules. However, we still need to iterate over all good schedules to find the best.

In order to determine whether scheduling a certain cell would actually resolve any dependencies, we should keep track of the already resolved dependencies. Each time we schedule a cell, for each resolved dependency we remove the corresponding edges from the graph. Keeping track of the previous state of the dependency graph in our recursive algorithm, allows us to effectively search the entire state space.

To simplify the implementation of the algorithm, the dependency graph is extended with a virtual *done* node. All nodes which do not have a successor, will receive an edge to this extra node. This way, all nodes with an outgoing edge should be scheduled; as long as there are edges left, scheduling is not finished. Thus the state of the system is fully described by the edges in the dependency graph.

The brute force search can be adapted to the backtracking approach as outlined in algorithm 4.2. Although the worst-case complexity of the backtracking is equivalent to the complexity of the brute-force approach, different tests show a significantly improvement on the required runtime.

### 4.3.4 Dynamic programming

Although the backtracking approach is already a lot more efficient than the brute force approach, still a lot of duplicate work is being done. For the scheduling without pipelining, the state of the system is solely determined by the remaining dependency graph. We can quite easily end up in identical states through different paths during backtracking (e.g. if there is no dependency between cell A and B, the order in which we schedule them does not matter: scheduling A before B will lead to exactly the same state as scheduling B before A).

With backtracking, each time we visit the same state we do exactly the same search over and over again. Rewriting the backtracking approach to

```
doSchedule($t, graph$) :
input: Time $t$ is the deltacycle we are about to schedule
input: Graph $graph$ contains the current state of the dependency
       graph
begin
    if $graph = \varnothing$ then
        if $t < Makespan$ then
            $Optimal \leftarrow Schedule$
            $Makespan \leftarrow t$
        end
        return
    end
    if $t > Makespan$ then
        return
    end
    foreach $c \in$ cells do
        $newgraph \leftarrow graph$
        Strip edges from $newgraph$ which are resolved by scheduling $c$
        if $newgraph \neq graph$ then
            $schedule[t] \leftarrow c$
            doSchedule($t + 1, newgraph$)
        end
    end
end
```

Algorithm 4.2: Backtracking approach for finding an optimal schedule

apply dynamic programming, allows us to omit doing this duplicate work. Implementing the dynamic programming approach is done by introducing a memo. A memo is a large memory in which we store states which are already visited, what we scheduled for that state and what the optimal makespan is for the rest of the schedule.

We can use this memo in the recursion: if we enter a function, we look up whether we have already visited that state. If we have already visited that state, we return what we have done previously. If we have not visited a state before, we schedule as we did with normal backtracking, but store our results in the memo before returning.

The adapted algorithm is listed in this algorithm 4.3.

Since the dependency graphs tend to require a relatively large amount of memory, it is not very practical as an index for a memo: usually the memo size is a bottleneck for dynamic programming, and looking up a state will be quite slow when we have to compare large blocks of memory.

```
doSchedule($t$, $graph$) :
input: Time $t$ is the deltacycle we are about to schedule
input: Graph $graph$ contains the current state of the dependency
       graph
begin
    if $graph = \varnothing$ then
        return $0$
    end
    if $memo[graph] \neq \varnothing$ then
        return $memo[graph]$
    end
    foreach $c \in$ cells do
        $newgraph \leftarrow graph$
        Strip edges from $newgraph$ which are resolved by scheduling $c$
        if $graph \neq newgraph$ then
            $result =$ minimum($result$, doSchedule($t + 1, newgraph$))
        end
    end
    return $(memo[graph] \leftarrow (result + 1))$
end
```

Algorithm 4.3: Dynamic programming approach for finding the minimal makespan

To reduce the memory required for the memo, a specialized graph representation is used. Some insights on which the representation is founded are that the dependency graph is very sparse (e.g. in the regular structures, each node has a degree of $\leq 2$: there is no branching in the dependency graph), and that since all states during scheduling are constructed by removing edges, the initial dependency graph supersets all states. To store each state, a bitmap representation of the graph is used: each bit maps to one edge in the initial dependency graph. A 1 in the bitmap indicates the edge is still there, a 0 marks that the edge has been removed. Using this bitmap representation, the state size for scheduling a Mesh6x6 network (with more then 550 nodes in the dependency graph) could be reduced to just 16 bytes.

The memo itself is a hash-map which maps a graph state onto a tuple of the makespan for optimally scheduling the remaining graph and the cell which should be scheduled for this sub-graph. The stored makespan is used to find an optimal makespan for the entire schedule, and since we store which cell should be scheduled, we can reconstruct an optimal schedule when the memo is filled.

### 4.3.5 Optimal schedules

The discussion of the algorithms above, assumed scheduling without pipelining. Pipelining complicates things a bit: when the execution of a cell starts, it takes several delta cycles before the data is available (and thus connected cells are actually triggered). The hypercell can start executing the next cell at the next delta cycles, before execution of previously started cells is done. This has huge impact on the state of the system.

Scheduling with pipelining has a similar approach as scheduling without pipelining, but as noted there are some problems: the state of the scheduler is determined by each sub-graph in the pipeline, so the size of each state increases linearly with the number of pipeline stages and thus pressuring the memory usage for the memo. But even worse, the statespace itself (and thus computation times) grow exponentially with the number of pipeline stages!

Using this optimized implementation, optimal schedules were generated for several regular synthetic designs. The designs which were generated are networks with a mesh topology, as described. In the tables 4.1 and 4.2, the best- and worst-case makespans are given for all computable cases. The exact schedules are not listed, since there are many schedules which have the same minimal makespan. The tables also give the computation time that was required to generate the schedule.

| | Best-case | | Worst-case | |
|---|---|---|---|---|
| Topology | Makespan | Time | Makespan | Time |
| Mesh2×2 | 6 | 0.000s | 8 | 0.000s |
| Mesh3×2 | 9 | 0.110s | 12 | 3.375s |
| Mesh4×2 | 12 | 7.363s | 16 | 25.343s |
| Mesh3×3 | 14 | 85.505s | 18 | 565.731s |

Table 4.1: Makespan for 1 hypercell and no pipelining

| | Best-case | | Worst-case | |
|---|---|---|---|---|
| Topology | Makespan | Time | Makespan | Time |
| Mesh2×2 | 8 | 0.000s | 11 | 0.015s |
| Mesh3×2 | 10 | 0.171s | 13 | 7.929s |
| Mesh4×2 | 13 | 21.884s | - | - |

Table 4.2: Makespan for 1 hypercell and a pipeline depth of 2

From the tables we can see that the required computation time grows real fast. However, not only the required time was a bottleneck. The required memory for storing the memo also increased dramatically for larger designs.

Due to a limited available memory of 2GB, it was impossible to calculate schedules for larger or more complex cases using this approach.

## 4.4 Shortest Common Supersequence problem

From the previous section we can conclude that finding an optimal schedule for a structure is very impractical: even for very small designs, both computation time and memory usage form a huge bottleneck.

In specific cases (just one hypercell and no pipelining), our scheduling problem resembles the well studied mathematical *shortest common super-sequence* (SCS) problem.

The problem of finding the SCS can be formulated as follows: given two sequences $T = t_1, t_2, .., t_n$ and $S = s_1, s_2, .., s_m$ over an alphabet set $E$, we say that $T$ is a super-sequence of $S$, if for every $j \in 1, .., m$ there is a $t_{i_j}$ corresponding to $s_j$, for which $1 \leq i_1 < i_2 < .. < i_m \leq n$, meaning that the sequence $S$ can be found in the sequence $T$, while traversing $T$ from left to right. For a finite set of sequences $S = S_1, S_2, .., S_k$, sequence $T$ is a common super-sequence of $S$ iff $T$ is a super-sequence of each $S_i \in S$. The shortest common super-sequence of $S$, is the super-sequence with minimal length.

Finding a SCS of two strings has a polynomial complexity, but finding the SCS for n sequences is known to be NP-complete. Using dynamic programming for finding the SCS for a set of $k$ sequences of length $n$, takes both time and space complexity of $O(n^k)$.

Fortunately, several effective heuristics exist for finding short common super-sequences (although we cannot tell if the found super-sequence is actually the shortest possible). Those heuristics usually take a greedy approach and are very fast. Different heuristics will be discussed in section 4.4.1.

If we try to schedule a seq_hils with one hypercell and no pipelining, the set of cells is our alphabet and the optimal schedule will be the shortest common super-sequence of all dependency paths in the design.

For example, when scheduling a design like sketched in figure 4.1, we can see there are three dependency paths. A shortest common super sequence of these dependency paths, is $CABA$, which would be an optimal schedule for this design if there was just one hypercell without any pipelining.

### 4.4.1 Heuristic approaches for SCS

How well a heuristic works, depends on several aspects: the number of strings, the length of the strings, the number of letters in the alphabet, etc.

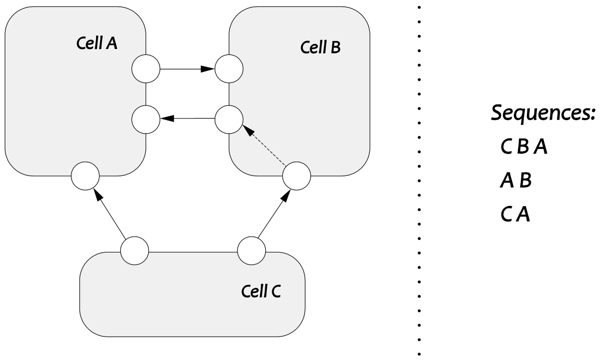


Figure 4.1: A sample design with several data dependencies

For scheduling the seq_hils we expect to have a lot (tens to hundreds) of short (less than 5 cells) sequences. A couple of existing heuristics, which are known to work quite well on average, will be discussed in this section.

The *Min-Height* greedy method [11], picks the first element of the longest sequence left. When there is a tie, a random element is picked. This way, we effectively try to balance and minimize all strings, which is expected to work well when the input strings are not very short and all strings are about the same length.

The *Majority-Merge* heuristic [11, 8] selects the element that occurs in front of the sequences the most frequently. In case of a tie, we again pick a random element. This way we try to reduce the sequences as fast as possible. Although no competitive ratio exists for this heuristic, it has a quite good average performance [8].

Note that when we say to pick a random element, this means the algorithm does not specify which element to pick: it does not have to be explicitly random, but can be deterministic depending on the implementation.

Both Min-height and Majority-merge heuristics can be extended with a look-ahead approach. In the look-ahead approach, we try each possible schedule several steps deep, before using the heuristic to pick which option would have been best. However, complexity grows exponentially with the number of steps. Due to the number of input sequences, it is infeasible to look ahead more then a few (approximately 3) steps.

### 4.4.2 Applying SCS heuristics on the seq_hils

The SCS problem only matches our scheduling problem, if our simulator has just one hypercell without pipelining. This hypothetical case is actually not realistic. In order to be able to use the heuristics for scheduling the seq_hils in all cases, we have to slightly adapt them.

To solve this problem, using a filtering approach (i.e. only select a specific set of nodes to take into account) in the heuristical function is proposed. The heuristical function will only evaluate the subset of strings for which certain conditions holds. From this subset a cell to schedule is picked.

Since there are two aspects where the SCS does not match our scheduling problem, the filter checks for two conditions.

The first condition should check if the first cell in a string could be evaluated on the hypercell we are currently scheduling. This way, we pick the best choice for this specific hypercell.

The second condition we check for, should enforce correct timing due to pipelining. To achieve this, we keep an "earliest starting time" for each string and a global timer. The global timer indicates the current delta cycle we are scheduling. Whenever we schedule a certain cell, for each string which has that cell as the first element, we remove the first character and set the "earliest starting time" to the current time + the pipeline depth. The "earliest starting time" for each string is initially 0. In our heuristic function, we should only consider strings which have their "earliest starting time" $\leq$ the global time.

When scheduling the seq_hils, for each delta cycle we can schedule all hypercells after each other while applying both filters.

## 4.5 The coalescing heuristic

As we will see in section 4.6, the SCS based heuristics work quite well for scheduling the seq_hils. However, if we take some known properties of the seq_hils into account, we can actually do better in some cases. In this section we will look into a specialized heuristic which was developed for use in the seq_hils.

We assume we have the set of dependency paths through the design we wish to schedule available. Like for the SCS based heuristics, to enforce pipelining we keep track of an "earliest starting time" for each dependency path, using an array `earliest`. The heuristic uses this information to make sure a dependency is not marked as resolved when scheduling a cell, before it actually is. Whenever a cell is scheduled, for each dependency path which starts with that cell, we remove the cell from the dependency path and set the earliest starting time for that dependency path at $time + pipeline\_depth$.

The scheduling itself is quite straightforward, and similar to the scheduling with the SCS heuristic. As long as there are non-empty dependency paths, we let a `heuristic()` choose a cell to schedule for the current time and hypercell. Then we remove the dependencies resolved by scheduling this cell. Finally we continue scheduling the next hypercell or evolve time. A more elaborate algorithm is given by algorithm 4.4.

```
$time \leftarrow 0, hypercell \leftarrow 0$
while count(dependencyPaths) $> 0$ do
    $cell \leftarrow$ heuristic( $time$ )
    $\text{schedule}_{time,hypercell} \leftarrow cell$
    foreach $path \in$ dependencyPaths do
        if headOf($path$) $= cell$ and $\text{earliest}_{path} \leq time$ then
            dependencyPaths $\leftarrow$ (dependencyPaths $- path$) $+$ tail($path$)
            $\text{earliest}_{path} \leftarrow time + \text{alpha}$
        end
    end
    $hypercell \leftarrow hypercell + 1$
    if $hypercell =$ parallel then
        $time \leftarrow time + 1$
        $hypercell \leftarrow 0$
    end
end
```

Algorithm 4.4: Global scheduling approach based on a heuristic function

The interesting part of the scheduling is handled in the `heuristic()` function, which should pick a cell to schedule based on the current state of the dependency paths. The heuristic function does this picking of a cell in a couple of steps.

The first phase of the heuristic function, is delaying all cells which can be coalesced: if multiple similar cells are in front of a dependency path, it can be smart to coalesce them together (ie. delay all similar cells until a time at which all cells can be actually executed). We do this as outlined by algorithm 4.5.

The array coalesceTime now contains earliest time at which a node should be evaluated if it was coalesced. The earliest time *mintime* at which a node is schedulable, is then searched for.

Next there is checked whether it is realistic to coalesce cells at this *time*: if coalescing means nothing can be scheduled then we are probably better off with not coalescing at all (it would be a shame to wait when we could have done something useful). In such a case, we undo it using algorithm 4.6.

Finally, we search for the cell which should have been scheduled the longest time ago as outlined by algorithm 4.7. In case of a tie, we pick an arbitrary (but not necessarily random) cell which resolves most dependencies.

```
foreach $c \in$ cells do
    $maxtime \leftarrow -\infty$
    foreach $p \in$ dependencyPaths do
        if headOf($p$) $= c$ then
            $maxtime \leftarrow$ maximum($maxtime$, earliest$_p$)
        end
    end
    foreach $p \in$ dependencyPaths do
        if headOf($p$) $= c$ then
            coalesceTime$_p \leftarrow maxtime$
        end
    end
end
```

Algorithm 4.5: Delay dependencies by coalescing

```
if $mintime > time$ then
    foreach $p \in$ dependencyPaths do
        coalesceTime$_p \leftarrow$ earliest$_p$
        $mintime \leftarrow$ minimum($mintime$, coalesceTime$_p$)
    end
end
```

Algorithm 4.6: If necessary, undo coalescing

The function `heurtistic()` now returns this *cellToSchedule* as a suggested cell for *time*.

We wil refer to this heuristic as the *Coalescing Heuristic*. In section 4.6, we will show some results of how this heuristic performs as compared to the other suggested heuristics.

### 4.5.1 An example

Assume we have a design with 4 cells. Between those cells, the following eight dependencies exist:

$A \rightarrow B$, $B \rightarrow A$, $A \rightarrow C$, $C \rightarrow A$, $B \rightarrow D$, $D \rightarrow B$, $C \rightarrow D$, $D \rightarrow C$

In this example, we want to schedule this design on one hypercell with a pipeline depth of 2. We apply the heuristic approach for scheduling.

For convenience, we use a grid layout for the dependencies to represent the timing: each column represents 1 time unit (in delta cycles), each row contains a dependency path. The bottom row contains the final schedule built up so-far. All dependent cells are located two timeslots away from their

```
$cellToSchedule \leftarrow$ '-'
$bestDependencyCount \leftarrow 0$

foreach $c \in$ cells do
    $depCount \leftarrow 0$
    foreach $p \in$ dependencyPaths do
        if headOf($p$) $= c$ and coalesceTime$_p$ $= mintime$ then
            $depCount \leftarrow depCount + 1$
        end
    end
    if $depCount > bestDependencyCount$ then
        $bestDependencyCount \leftarrow depCount$
        $cellToSchedule \leftarrow c$
    end
end
```

Algorithm 4.7: Select cell to schedule

predecessor to enforce the pipeline depth of 2. Once a cell is scheduled, the successive cells are moved in time, to enforce timing constraints.

| | | |
|---|---|---|
| A | | B |
| B | | A |
| A | | C |
| C | | A |
| B | | D |
| D | | B |
| C | | D |
| D | | C |

| | | |
|---|---|---|
| | | |

$\rightarrow$

| | | |
|---|---|---|
| | | B |
| B | | A |
| | | C |
| C | | A |
| B | | D |
| D | | B |
| C | | D |
| D | | C |

| | | |
|---|---|---|
| A | | |

**Step 1:** Initially all paths can start at time 1. All cells would resolve an equal number of dependencies, so we arbitrarily choose to schedule cell $A$.

| | | | | |
|---|---|---|---|---|
| | | B | | |
| | | B | | A |
| | | C | | |
| | | C | | A |
| | | B | | D |
| D | | B | | |
| | | C | | D |
| D | | C | | |

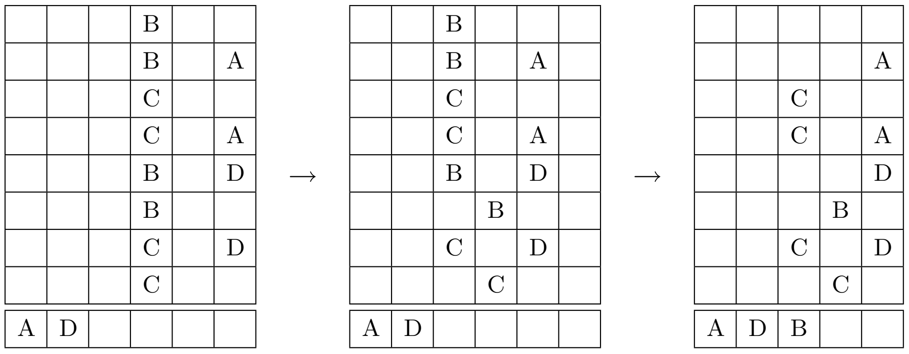
| | | | | |
|---|---|---|---|---|
| A | | | | |

$\rightarrow$

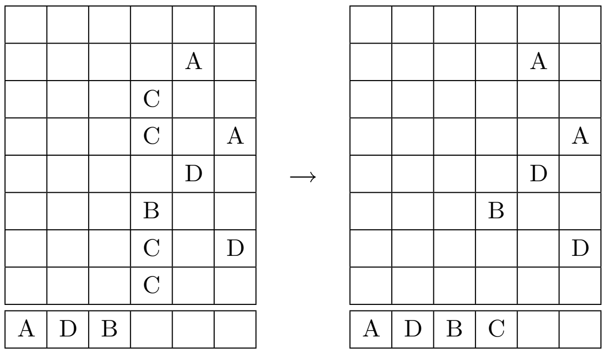
| | | | | |
|---|---|---|---|---|
| | | B | | |
| | | B | | A |
| | | C | | |
| | | C | | A |
| | | B | | D |
| | | | B | |
| | | C | | D |
| | | | C | |

| | | | | |
|---|---|---|---|---|
| A | D | | | |

**Step 2:** With an *A* scheduled, we can coalesce cells *B* and *C*. After coalescing, only *D* can be scheduled at time 2.



**Step 3:** Coalescing *B* and *C* leaves nothing to schedule for time 3, so we undo the coalescing. Both *B* and *C* would resolve three dependencies, so we arbitrarily choose to schedule *B* arbitrarily.



**Step 4:** After coalescing cell *C*, we choose to schedule *C* (which resolves most dependencies) for time 4.

Eventually, the schedule will yield to "ADBCBAD-": a makespan of 8 (after flushing pipelines), which is (coincidentally) optimal.

### 4.5.2 A counter example

Since this is a heuristic approach, it does not always yield to an optimal schedule. In cases where (from the heuristic's point of view) a tie exist at a specific time, it might arbitrarily choose a 'wrong' cell to schedule. And as we will see in section 4.5.3, not only in cases of a tie we can make a wrong decision! The term 'wrong' might be a bit misleading: the choice will always lead to a correct schedule; however it is not guarantied to be optimal.

As a simple example of how this might go wrong, we extend the previous example with a additional $A \rightarrow D$ dependency.

For the first timeslot, an $A$ will be scheduled (since it will resolve three dependencies, where $B$, $C$ or $D$ would only resolve two). Coalescing during the second step does not help: nothing will be left to schedule for time two, so the coalescing will be undone.

The state will look like this:

| | | |
|---|---|---|
| | | B |
| B | | A |
| | | C |
| C | | A |
| B | | D |
| D | | B |
| C | | D |
| D | | C |
| | | D |

| | | |
|---|---|---|
| A | | |

This leaves a three-way tie, in which each option yields to a - from the heuristic's point of view - similar situation:

| | | | |
|---|---|---|---|
| | | B | |
| | | | A |
| | | C | |
| C | | A | |
| | | | D |
| D | | B | |
| C | | D | |
| D | | C | |
| | | D | |

| | | | |
|---|---|---|---|
| A | B | | |

or

| | | | |
|---|---|---|---|
| | | B | |
| B | | A | |
| | | C | |
| | | | A |
| B | | D | |
| D | | B | |
| | | | D |
| D | | C | |
| | | D | |

| | | | |
|---|---|---|---|
| A | C | | |

or

| | | | |
|---|---|---|---|
| | | B | |
| B | | A | |
| | | C | |
| C | | A | |
| B | | D | |
| | | | B |
| C | | D | |
| | | | C |
| | | D | |

| | | | |
|---|---|---|---|
| A | D | | |

But eventually, scheduling a $B$ or a $C$ will lead to a makespan of 9, where scheduling a $D$ could still lead to an optimal makespan of 8.

### 4.5.3 Another counter example

If we add another dependency to the first counter example, namely $B \rightarrow C$, things start to look even worse, since the heuristic will deliberately pick a non-optimal cell:

| | | |
|---|---|---|
| A | | B |
| B | | A |
| A | | C |
| C | | A |
| B | | D |
| D | | B |
| C | | D |
| D | | C |
| A | | D |
| B | | C |

| | | |
|---|---|---|
| | | |

$\longrightarrow$

| | | |
|---|---|---|
| | | B |
| B | | A |
| | | C |
| C | | A |
| B | | D |
| D | | B |
| C | | D |
| D | | C |
| | | D |
| B | | C |

| | | |
|---|---|---|
| A | | |

At this point, no tie exists. The heuristic will decide to schedule a $B$, since it resolves three dependencies where $C$ and $D$ would only resolve two. Eventually this will lead to a schedule “ABCBCDABC-” which has a makespan of 9.

If however we would have chosen $D$ (which resolves less dependencies) as second cell to schedule, we could have got a makespan of 8.

## 4.6 Evaluation of the different heuristics

In order to compare the different heuristic scheduling algorithms, we let them schedule a wide veriety of random structures on several configurations for the simulator. We also simulate the round robin arbiter on the same structures, using $Prob1 = 100\%$ and $Prob2 = 100\%$ (which will lead to a worst-case schedule) in order to compare the round robin scheduler with the offline scheduling approaches under comparable conditions.

The impemented heuristical approaches are the Majority-Merge and our own Coalescing heuristic. The Min-Height heuristic was not implemented, because it is expected to perform about equally (or even less, due to the relative short sequences) as compared to the Majority-Merge heuristic.

The generated structures which were used to benchmark the algorithms, can be grouped into 4 categories:

- 50× a random structure consisting of 12 cells with 8 ports per cell
- 50× a random structure consisting of 25 cells with 20 ports per cell
- 20× a random structure consisting of 48 cells with 30 ports per cell
- 1× a $4 \times 3$ torus topology Network-on-Chip

For each structure, the simulation is repeated with a number of different parameter combinations for the pipeline depth and the number of hypercells. There was choosen for a select set of combinations, such that we can discriminate between pipelining and no pipelining, and parallelism verses no parallelism. Exact values did not matter that much. A short test showed that the exact pipeline depth does not hava a huge impact on the scheduling performance: depths of 7, 10 and 15 resulted in almost similar figures (as there are enough cells in the DUV's to keep the pipelines filled). Also the exact number of parallal hypercells does not matter that much: 2, 3 of 4 also show comparable figures. The final parameters were choosen in order to keep the required simulation times as short as posible.

### 4.6.1 Simulation results

The simulation and scheduling results which are obtained are summarized in table 4.3.

| Parallel hypercells | Pipeline stages | Round robin arbiter | | Coalescing heuristic | | SCS: Majority Merge | |
|---|---|---|---|---|---|---|---|
| # | # | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| 1 | 1 | 25.01 | 0.78 | 19.52 | 2.01 | 18.72 | 1.51 |
| 1 | 7 | 44.18 | 1.19 | 35.14 | 2.68 | 35.26 | 1.90 |
| 2 | 7 | 31.04 | 0.61 | 28.00 | 2.65 | 26.80 | 0.94 |
| 1 | 1 | 50.41 | 1.39 | 47.18 | 2.12 | 45.48 | 1.84 |
| 1 | 7 | 66.54 | 1.87 | 58.68 | 2.75 | 59.60 | 1.88 |
| 2 | 7 | 42.29 | 1.04 | 38.70 | 2.01 | 37.74 | 1.37 |
| 1 | 1 | 91.90 | 1.31 | 86.00 | 3.21 | 83.85 | 3.13 |
| 1 | 7 | 106.30 | 1.76 | 93.90 | 2.95 | 98.00 | 2.63 |
| 2 | 7 | 61.27 | 0.85 | 55.30 | 2.17 | 55.85 | 1.77 |
| 1 | 1 | 24.00 | 0.00 | 24.00 | 0.00 | 24.00 | 0.00 |
| 1 | 7 | 46.50 | 0.00 | 43.00 | 0.00 | 44.00 | 0.00 |
| 2 | 7 | 33.72 | 0.00 | 37.00 | 0.00 | 33.00 | 0.00 |

Table 4.3: Offline scheduling with several algorithms

As we can see, both offline scheduling approaches perform quite equally. However, on individual cases there is some difference in performance: sometimes the new heuristic shows better results, where in other cases the Majority-Merge heuristic performs better.

In all cases, at least one of the offline scheduling algorithms performs better then the round robin arbiter.

### 4.6.2 Approach to be implemented

We have looked at several approaches for finding optimal schedules. However, these options have a very bad time complexity, and are thus not suitable for scheduling the seq_hils.

Also, two heuristical approaches are presented. The first approach is a modified version of the Majority-Merge heuristic which was originally developed for finding a shortest common super-sequence. The second approach is developed specifically for the seq_hils: it uses the concept of coalescing the evaluation of cells to prevent double work.

Both heuristical approaches build a schedule on the same base: all data paths through the system. Also, both algorithms also have a $\mathcal{O}(n * m)$ time complexity (where $n$ is the number of sequences and $m$ is the makespan of the schedule; $m$ is upper bound by the sum of the length of all sequences), versus the exponential complexity for finding optimal schedules.

The simulation of the algorithms shows that they perform about equivalent on average, although there are significant differences in individual cases. Therefore we might as well implement both offline algorithms and pick the schedule with the best makespan as final schedule for the seq_hils.

CHAPTER 5

# Online scheduling approaches

The offline schedules, which can be generated with the algorithms discussed in the previous chapter, form a nice basis for a schedule, but are not yet as good as we would like the final schedule to be. For generating a correct offline schedule, a lot of pessimistic assumptions on data dependencies have been taken: not all combinational paths in the system will actually toggle each delta cycle. In order to further decrease the number of delta cycles per system cycle (and thus increase the performance of the schedule), we would like to incorporate an online scheduler which is based on the offline schedule, and optimize it as information about unstable cells becomes available.

## 5.1 Approaches for online scheduling

The online scheduler has been eventualy implemented in VHDL as part of the simulator. The implemented algorithm should be simple enough (i.e. runtime and space complexity should comparible to the round robin arbiter), so that we effectively increase performance of the seq_hils (e.g. requiering only a few delta cycles less, but with a much slower clock-frequency for the resulting seq_hils, is not desirable).

At runtime, the online scheduler has information about which cells are unstable at each time we should pick a cell for evaluation. The online scheduler could base its decisions on the current time $\tau$, the known list of unstable cells and a pre-computed offline schedule.

The online scheduling descisions are made on a per-cycle base: every delta cycle, we evaluate the unstable cells, and for each hypercell pick a cell to schedule.

### 5.1.1 Fixing best-case schedule

The first (and simplest) online scheduling approach, would be fixing the best-case schedule. The best-case schedule might need fixing because not all posible combinational paths are taken into account during the construction of a best-case offline schedule. If however, such combinational paths cause certain cells to become unstable, our schedule does no longer guarantees that the system stabilizes. After the best-case schedule is finished, the cells which are not yet stable have to be stabilized using an online scheduling approach.

We could fix the best-case schedule, by stabilizing the few cells which are left unstable using (for instance) a round robin arbiter. Since this approach is based on optimistic assumpsions, we also assume that the system is almost stable, and the round robin arbiter can stabilize few cells which are unstable without causing many re-evaluations.

### 5.1.2 Optimizing worst-case schedule

Instead of fixing an incomplete offline schedule, we could also base our online schedule on a worst-case schedule which will guarantee stabilisation of the system. Since we know the schedule is pessimistic for an average simulation of a system cycle, we might be scheduling cells which are not unstable. This is a waste of time, and we would like to pick something useful to do in these cycles.

A trivial thing to do would be skipping each unnecessary step in the schedule. There are two slightly different options to do so:

- Locally within a hypercell
- Globally within a hypercell group

Picking the next unstable cell within the schedule for one hypercell, is quite simple to implement: iterate over the (remaining) schedule for the current hypercell in order to find the first unstable cell scheduled for time $\geq \tau$. An outline for this algorithm is given in algorithm 5.1.

A similar approach can be taken for an entire hypercell group. A schedule for a hypercell group can be seen as the set of cells, which have to be evaluated at every cycle. At each time $\tau$, we pick a cell to schedule for each hypercell from the hypercell groups schedule. When a cell is already stable, we pick the next cell, similar to the per-hypercell online scheduling, as suggested in algorithm 5.2.

```
onlineSchedule($hypercell$, $t$) :
input: $hypercell$ is the hypercell within the hypercell group we are
       about to schedule
input: Time $t$ is the deltacycle we are about to schedule
begin
    while $t <$ makespan do
        if isInstable(schedule[$hypercell$][$t$]) then
            return schedule[$hypercell$][$t$]
        end
        $t \leftarrow t + 1$
    end
end
```

Algorithm 5.1: Locally optimize best-case online scheduling approach

Those two approaches only differ if there are multiple hypercells in a hypercell group and when the schedule for one of those hypercells can skip significantly more cells than any other hypercell. Otherwise, scheduling results will be similar.

A small example of this can be given using figure 5.1. Assume we have two hypercells within a hypercell group. The figure gives the two schedules for the two hypercells. Cells A and B are both marked as stable, and thus can be skipped. Using a local optimization approach, will lead to scheduling cells C and E, where applying a global optimization approach will lead to scheduling cells E and F. If B would be unstable, both approaches will lead to the same cells to be selected for scheduling: B and E.

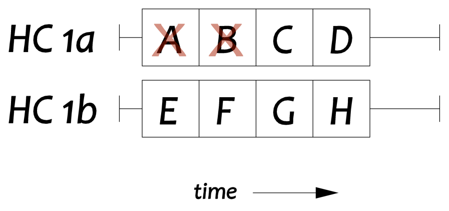


Figure 5.1: Simple online scheduling example

Note that the term “skipping” is used here to jump over cells which do not require scheduling *right now*. If multiple cells were skipped, we should reconsider at the next cycle, all cells which are scheduled for time $\geq \tau_{new}$: it might be that cells which have already been scheduled (or even skipped) in a previous delta cycle are unstable again due to pipelining! However, if a cell, scheduled for time $\tau$, is stable, we can surely skip it without needing to reconsider that cell (e.g. in the example we schedule cell $C$ for now, but we should reconcider cells $B$ and $C$ at the next delta cycle; $A$ can savely be

```
onlineSchedule(hypercell, t) :
input: hypercell is the hypercell within the hypercell group we are
       about to schedule
input: Time $t$ is the deltacycle we are about to schedule
begin
    while $t <$ makespan do
        for $h := 0$ to hypercells do
            $hc \leftarrow (hypercell + h)\%$hypercells
            if isInstable(schedule[$hc$][$t$]) then
                return schedule[$hc$][$t$]
            end
        end
        $t \leftarrow t + 1$
    end
end
```

Algorithm 5.2: Globally optimize best-case online scheduling approach

skipped without need to reconsider it).

### 5.1.3 More complex approaches for online scheduling

During the research topic in preparation of this thesis, several complex algorithms for the online scheduler have been thought off.

However, as we will see in the next section, the proposed simple online approaches already perform quite well. It is not expected that implementing a complex online scheduler would yield in a much better performance (although the makespan of the schedules might get slightly shorter, it would probably be overshadowed by the the increased computational complexity). Therefore designing and implementing a complex online scheduler is discarded for now. Exploring possibilities to effectively implement smarter scheduling approaches, will remain future work.

### 5.1.4 No online scheduling at all

The last option for online scheduling, is ommiting the online scheduler at all. As we will see from the benchmarking results in section 5.2, the offline schedulers can find a schedule which has an expected makespan which is comparable with the makespan the round robin arbiter would provide. Using only this offline schedule, without any online scheduling, in the seq_hils might also be an interesting option. The worst-case offline scheduler can guarantee that the system is stable when the schedule is done. Thus, such an approach would reduce the overhead introduced by the port memories which have to trigger the scheduler. Stripping this overhead from the port memories and the logic for do the bookkeeping of unstable cells from the simulator might make the system more scalable and might allow higher clock frequencies, and thus also cause a performance increase. However, examining this possibility will also remain future work.

## 5.2 Evaluation of the different algorithms

In order to evaluate the suggested online scheduling approaches, they were implemented into the metasim simulation environment. All three online scheduling approaches (fixing best-case, locally optimizing worst-case and globally optimizing worst-case) were combined with both the SCS based heuristic and our coalescing heuristic, which provide the essential offline schedules. The offline schedules were also simulated without any online optimization, to get an indication of how well the online schedulers can optimize the different offline schedules.

A wide range of input structures were used for simulation in the metasim: many different random structures, with three different sizes (i.e. 12 cells with 8 ports each, 25 cells with 20 ports each and 48 cells with 30 ports each; those sizes are arbitrarily chosen to represent small and medium sized SoC-designs). All structures were simulated multiple times using several different simulation parameters (i.e. no pipelining and just 1 hypercell, a pipeline depth of 7 with just 1 hypercell and a pipeline depth of 7 with 2 hypercells in parallel). These values are equal to the parameters chosen for the metasim simulation of the offline scheduling approaches.

By running simulation with many different input structures and many different parameter configurations (which is expected to be a good enough subset of possible parameters, to give unbiased results which would also apply to majority of possible seq_hils inputs), a broad variation of combinations was tested, and thus the different algorithms can be properly compared to each other.

As a comparison on the total performance, we also simulate each structure (with all different parameters) using a round robin arbiter. We run the simulation with the round robin arbiter a 100 times using different orderings of cells, to average out the chance of coincidentally having a good or bad ordering for the round robin arbiter. This allows us to compare the new algorithms with a round robinapproach, and thus predict a performance increase for when the algorithms are implemented in the seq_hils.

For each input structure and all possible input parameters, we simulate each algorithm during 10000 “system cycles”. For each system cycle we count how many delta cycles it takes to stabilize the design. This number of delta cycles is averaged over the complete simulation of 10000 system cycles. Thus, for each possible combination of input parameters, we get one number: the average number of delta cycles for that specific test case.

### 5.2.1 Simulation results

For each simulation run, the number of delta cycles required to stabilize the system is stored. In order to compare the algorithms, we normalize these results. The normalization consists of calculation a relative performance of each simulation run for each algorithm as compared to the results of the round robin arbiter. The relative performance for an algorithm is given by $cycles_{algorithm}/cycles_{roundrobin} \times 100\%$. A lower relative performance corresponds to a higher performance: thus lower is better.

Table 5.1 gives a short overview of the average relative performances and the standard deviation per algorithm.

| Algorithm | $\mu$ | $\sigma$ |
|---|---|---|
| Coalescing heuristic, offline only | 96.37% | 11.07 |
| Coalescing heuristic, locally optimize online | 89.85% | 6.85 |
| Coalescing heuristic, globally optimize online | 89.65% | 6.75 |
| Coalescing heuristic, fix best-case online | 94.58% | 6.79 |
| SCS-based heuristic, offline only | 94.79% | 9.56 |
| SCS-based heuristic, locally optimize online | 89.89% | 6.79 |
| SCS-based heuristic, globally optimize online | 89.62% | 6.65 |
| SCS-based heuristic, fix best-case online | 100.91% | 10.35 |

Table 5.1: Relative performance of different scheduling algorithms as compared to the round robin arbiter

The results show that both our coalescing heuristic and the SCS based heuristic give similar performance increases on average; either without online scheduling as with an optimizing best-case online scheduling strategies. However, in individual cases, there is a lot of spread in performance (in 40% of the runs, the relative performances differ more then 5%): in some cases our coalescing heuristic works much better (e.g. in a single case, the coalescing based offline schedule is 19% better then the SCS based offline schedule; after online scheduling, this coalescing based schedule still performs 14% better), and in other cases the SCS based heuristic works much better (e.g. in a single case, the SCS based offline schedule is 44% better then the coalescing based offline schedule; after online scheduling, this SCS based schedule still performs 7% better). There is no correlation observed between input parameters and the spread in relative performance.

The "fixing best-case" online approach is clearly not suitable: although in most cases where the value for $prob1$ is 1%, its performance is slightly better then the other online scheduling approaches, it is even outperformed by the static worst-case schedule in more then 62% of the test cases! This is since a round robin arbiter (within the fixing best-case approach) only starts fixing unstable cells when the entire offline schedule is finished.

Both "optimize worst-case" approaches show similar results, independent of the offline schedule they are based on.

In appendix A, we have plotted a histogram of the relative performances for each individual scheduling algorithm. We can see that the two offline-only scheduling approaches have a quite semetric histogram (figures A.1 and A.5), with the peak a left of 100 (this shows the offline scheduling approaches perform, on average, better then the round robin abriter). When we apply an optimizing online scheduling approach (figures A.2, A.3, A.6 and A.7), we can see that the peak gets a lot steaper, and almost the entire historgram lies left of the 100-mark. The locally and globally optimizing online scheduling approaches show an almost identical histogram, which shows both approaches work about equally well. The fixing best-case online scheduling approaches are clearly less good than the optimimizing worst-case approaches: the histograms (A.4 and A.8) are less steap, and have much more values past the 100-mark.

Appendix A also lists the complete table of simulation results, from which table 5.1 is deducted. The table shows the performance increase grouped per set of parameters.

### 5.2.2 Approach to be implemented

As we have seen from the simulation results, there is a significantly difference in performance between the two (i.e. fix best-case vs. optimize worst-case) online scheduling approaches. We have concluded that the fixing best-case approach is not a suitable choice. The "optimize worst-case" approach can give a guarantied upper-bound on the online schedule, which is often (e.g. in more then 65% of the test cases) better than what the round robin arbiter could do, and it can optimize the offline schedule another 5% on average; it is an appropriate choice for the seq_hils.

Both optimize worst-case scheduling approaches perform about equally. As one would expect, the difference only shows in the test cases with 2 hyper-cells. Even in those cases, the globally optimizing approach is only slightly better then the locally optimizing approach. Since the locally optimizing approach uses less data (per scheduling entity) and has a loop less, it is expected to yield in less hardware when implemented in the seq_hils.

What we can also note is that both offline schedules can be equally optimized: none of the schedulers let their schedule be optimized significantly better then the other. This shows that both offline schedulers are equally suitable as a base for the online scheduling. This is a nice feature: since it strongly depends on the structure of the DUV which of the heuristics yield in a better offline schedule, we can offline take both approaches and pick the schedule with the shortest makespan to lower the upper-bound for run time. (Note that when we take the minimum makespan of the offline schedulers for all metasim simulation results, we find an average relative performance of 93% with a standard deviation of 9.4, which is almost 2 percent better than any of the individual offline scheduling approaches, while it has less spread in the makespan)

Based on these conclusions, we have implemented both offline scheduling approaches in the tool chain, and reworked the online scheduler in the seq_hils to execute and optimize the worst-case schedule. As an online scheduling approach, we have implemented the locally optimizing worst-case schedule approach.

From these simulation results, one could expect that implementing those algorithms into the seq_hils will lead to an average performance increase of about 12% to 15%.

CHAPTER 6

# Implementation

We have proposed and tested several online and offline scheduling approaches which seemed suitable for scheduling the seq_hils. From these tests, we picked several algorithms which we will implement into the simulator.

The implementation of the scheduling into the seq_hils can be split in roughly two parts:

- The offline scheduling should be performed inside the tool chain developed by Rutgers [17]. The Java based tool will process a technology mapped EDIF netlist, and generates the synthesizable VHDL code for a simulator.
- The online scheduling should be implemented in VHDL, which will be generated by the tool chain

The modifications made to the tool chain will be discussed in section 6.1, the implementation of the VHDL entities will be discussed in section 6.2.

In the third section of this chapter, we will look at two simulations performed with the new scheduling algorithms implemented in the seq_hils. The simulation timings from both simulation with the round robin arbiter and the new algorithms in place are being compared, showing the performance increase that is obtained.

## 6.1 Algorithms implemented in the tool chain

The generation of the seq_hils is done by a Java tool chain developed by Rutgers [17]. It processes the DUV and generates a set of VHDL entities which form the entire simulator. Both the offline scheduling and the generation of the online scheduler are implemented in the existing tool chain.

### 6.1.1 Architecture of the tool chain

The generation of a seq_hils for a specific DUV, is performed by a Java tool chain. The tool reads a configuration file and a technology mapped EDIF netlist as inputs, and constructs several DUV-specific VHDL entities (e.g. the hypercells, hypercell arrays, etc.) to complement the static seq_hils entities (e.g. the central controller, the round robin based scheduler, etc.) to form a complete simulation environment for the DUV [17].

The `Transformer` class is the body of the tool. It contains the `main` function and thus forms the entry point for the tool chain. It starts by loading the configuration file, and initializes the EDIF graph data structures.

After this, it sequentially performs several `TransformTask`'s which all process the EDIF, and optionally restructure the graph or generate the VHDL files. Some tasks depend on other tasks to be performed first: there is a strict ordering in which the tasks are perfomed. For a default run, the following tasks are performed:

`ParseTask`: Loads the EDIF netlist from file into memory, and initializes the technology backend which matches the EDIF. The technology backend contains device specific functions for the FPGA on which the DUV is to be mapped.

`ReportStatsTask`: Dumps an elaborate report on all primitive elements used in the DUV.

`BuildHypercellsTask`: Construct the hypercells from the cells in the EDIF. The configuration file contains a specification of which cells should be grouped into which hypercell

`ExtractStateTask`: Extracts the states from each hypercell.

`ExportNetlistTask`: Export the generated hypercells to VHDL source file files

`GenerateHypercellArrayTask`: Wraps the instances of hypercells into hypercell arrays.

The offline scheduling of the seq_hils should be performed after the `BuildHypercellsTask`: at that time, a complete graph of the DUV is available as well as a mapping of cells onto hypercells. From the configuration, we know how many instances of each hypercell there are.

The online scheduler is generated from this offline schedule, and should be written as VHDL entity from within the `GenerateHypercellArrayTask`: only then we know for sure which address maps to which hypercell group.

However, from a design point of view, it is inconvenient to embed the online scheduling task within the `GenerateHypercellArrayTask`: it requires quite some restructuration to get all information from the offline scheduling (e.g. the generated schedule, the mappings from addresses onto hypercells, etc.) into the `GenerateHypercellArrayTask` instance. Therefore it is currently not implemented in the `GenerateHypercellArrayTask`, but as a part of the `SchedulerTask` right after the offline scheduling is done. Since we have knowledge of the internal data structures used for storing the information about cells, hypercells, hypercell groups and the hypercell arrays, we can reproduce the address assignment as done by the `GenerateHypercellArrayTask` and thus produce valid schedules. Although this is not a clean solution from an architectural point of view, it removes the need to restructure the entire tool.

### 6.1.2 Dependency graph traversal

The first step in the scheduling is the graph traversal of the EDIF netlist in order to deduct the dependency paths, using the algorithm as discussed in section 3.1.1. This is done in a separate task in preparation of the scheduling: the `PathfinderTask`.

The task starts with constructing a dependency graph structure. A node in the dependency graph, `DependencyNode`, stores references to the instance of a cell it maps to and a instance to a top-level port in that instance. Two maps `incomingEdges` and `outgoingEdges` map nodes to a set of connected nodes. This actually stores redundant data: each edge is effectively stored twice, but this redundant storage allows for fast lookup of edges.

In order to fill the graph structure, we walk over each top-level input port of all cells in the DUV. For each input port we find the output port of (another) cell to which it is connected, and add an edge for that connection to the dependency graph. For that same input-port we also do a breadth-first search through the netlist of the cell (analog to algorithm 3.1), to find combinational paths to output ports. The technology backend provides information about which ports in the netlist are flipflops, and thus break the combinational path.

If the EDIF netlist is incomplete (i.e. the design contains black-box entities), we cannot deduct the combinational paths between elements. In order to be able to generate a valid schedule, we do a worst-case assumption: for black-boxes we add all combinations between input ports and output ports as a combinational dependency to the dependency graph. Although this is a pessimistic approach, this is the only way we can guarantee to find a correct schedule which we can optimize online later on. The same approach was used by Penry et al. [12] in the Liberty Simulation Environment.

Since we use maps and sets for storage of the edges, we automatically omit double paths in the graph.

When we have completed the dependency graph, we can traverse the graph to unwind all dependency paths. Since we use two maps, to store both the incoming edges and outgoing edges of each node, we can easily check (constant time complexity) if a node has no incoming edges and thus is head of a path. The traversing algorithm is implemented analog to the algorithm described in section 3.1.1. Each dependency path is stored as a `List` of cell instances in a `Set`. The `PathfinderTask` keeps this set of unique dependency paths through the system in memory for use by the offline scheduling algorithms later on.

### 6.1.3 Offline scheduling

The offline scheduling starts with initializing some data structures required for scheduling. We lookup some configuration settings (which hypercells and how many instances are there), and store them in a convenient way: we build a map of which cell is mapped on which hypercell, we store the multiplicity of each hypercell, and store the address of each cell within a hypercell.

We also copy the set of dependency paths. This allows us to modify the data dependencies at will, while keeping the data available for later use.

For the offline scheduling itself, both the SCS based heuristic and our coalescing heuristic are implemented for finding a worst-case schedule. The overlap in the algorithms is implemented in the abstract `SchedulingAlgorithm` class. Both the `SCSBasedSchedulingAlgorithm` and the `Heuristical-SchedulingAlgorithm` inherit from this class, and override only the heuristic function.

Since both heuristic approaches have low-order polynomial time complexity, they can be executed very fast. The algorithms are implemented as described in chapter 4, with as small addition that in case of a tie in the heuristic, we explicitly pick a random cell for scheduling: we build a list of all options which form a tie, and use a random function to pick one of those options (the original algorithms specified we are free to arbitrarily pick one; in the metasim implementation of the algorithms, we deterministically picked the first option we found)

The schedulers schedule the system according to the dependency paths, and return the makspan for the entire schedule. Both the fact that we use a heuristical approach, and the non-deterministic behavior within the scheduling heuristic, introduces a chance of finding a coincidental “bad” schedule. Running the scheduling algorithms several dozens of times on a single test case, showed that multiple runs of the same heuristic on the same data structures can find schedules which have different makespans: a spread of about 5% was observed. In order to increase the chance of finding a good schedule, we execute each scheduling heuristic 10 times (10 should be enough to decline a coincidental bad schedule), and pick the schedule with the best makespan for use in the seq_hils.

### 6.1.4 VHDL generation

Once we have constructed an offline schedule for the seq_hils, we can generate the VHDL entities for the online scheduling. For each hypercell, we construct a scheduler entity. A schedulecontainer will instantiate all scheduling entities, and connects them to the simulator. The VHDL implementation of the entities will be discussed in the next section.

The writing of the VHDL files goes through a `ScheduleWriter` helper class, which constructs the VHDL entities from a set of static blocks of code.

## 6.2 Generated VHDL

The VHDL code which is generated by the tool chain, is developed and tested as separate VHDL code before it was integrated. This section will go into the design of the VHDL entities used for scheduling in the seq_hils.

### 6.2.1 Architecture of the original round robin based scheduler

In the seq_hils, the scheduler (the `sch_a` entity) determines which unstable instance has to be scheduled next at which hypercell. For this task it keeps track of an ‘unstable’ flag for all instances of cells in the DUV. These flags are stored in an `inst_unstable_r` register: one bit for every cell.

The scheduler is responsible for the bookkeeping of the `inst_unstable_r` vector. The scheduler receives updates on invalidated instances at every delta cycle form the Central Controller (CC). This information is combined (using a bitwise OR) with the previous set of unstable cells to keep a complete overview. Each time a cell is scheduled for evaluation, it is directly masked out as unstable.

The scheduler contains an instance of a round robin arbiter for each hypercell. The arbiter receives a selection of the `inst_unstable_r` vector, which it uses to select the next cell to schedule. Each round robin arbiter keeps its own state: the arbiter knows which instance it has previously selected for evaluation; this knowledge of the previous instance is used in the scheduling process. The arbiters output consists of a valid bit which indicates a valid instance to schedule is available, and a natural number containing the address of the selected instance. Note that the arbiter is implemented completely combinational: selecting an instance to schedule is done within a clock cycle.

There are several control input lines which determine the state of the scheduling. The states available are `RESET`, `IDLE`, `INIT`, `RUN_SCH`, `HOLD_RUN_SCH` and `FLUSH`; the state names are self-explanatory.

### 6.2.2 New implementation of the online scheduler

For the new scheduler to be a drop-in replacement of the old scheduler, we need it to have the same entity interface and behave similar to the control lines. To achieve this, the code for the new scheduler is based on the code of the original `sch_a` implementation.

The original scheduler was a generic source file, which applied for each generated simulator. Since we will construct DUV-specific schedules, our scheduler entiry is no longer generic: each DUV will have its own scheduler implementation, which is generated by the tool chain discussed in section 6.1.

This is to ensure similar observable behavior, while keeping development time low, the implementation of the new scheduler is mainly based on the original round robin based scheduler. The main difference in implementation is in the instantiation of arbiters. Where in the original design a "`for` ...`generate`" VHDL structure instantiates a number of generic round robin arbiters, this cannot be done generically in the new implementation. Each hypercell has its own specific schedule, so there is no generic entity which can be instantiated. Since we generate the scheduler for each DUV from within the Java tool chain, this is not a problem: we can easily instantiate the different entities from within the same tool chain.

The scheduling entities for each hypercell are different: each entity contains its own schedule and hypercell specific parameters. However, the interface and implementation are similar for each scheduling entity.

The interface for each scheduling entity is similar to the interface of the round robin arbiter: the inputs are a clock and reset signal, and a vector of unstable cells. The output is a valid flag and an identifier indicating which cell to schedule.

Every scheduling entity has several hypercell specific constants which determine the schedule: a `SCHEDULE` array holds a list of cells which have to be executed in order: this is the offline determined schedule. The length of this list is the worst case makespan.

Furthermore, each scheduling entity keeps track of its own state consisting of the current delta cycle and a vector of previously unstable cells. The delta cycle determines from where in the schedule we should start searching for an unstable cell. At a reset signal, the delta cycle is set to 0. At each simulator clock tick, it is incremented. Since the scheduling entities do not receive a reset signal at the end of a system cycle, we detect a new cycle ourselves: if we had no unstable cells in the previous clock cycle, and now all cells are marked as unstable, we restart our schedule by setting the current delta cycle back to 0.

```
process(nrst,clk)
begin
  if rising_edge(clk) then
    if nrst = '0' then
       deltacycle <= 0;
    else
       if request_r = ALL_STABLE and
          request = ALL_INSTABLE then
         request_r <= request;
         deltacycle <= 0;
       elsif deltacycle < CYCLES then
         request_r <= request;
         deltacycle <= deltacycle+1;
       end if;
    end if;
  end if;
end process;
```

Figure 6.1: Implementation of the state book keeping for a scheduling entity

Like in the round robin arbiter, the selection of a cell to schedule is implemented using only (asynchronous) combinational logic. We walk over the array containing the schedule, and we select the first cell which is unstable which has an index which is greater or equal to the current delta cycle.

Such a scheduler can be generated for each DUV-specific hypercell. The seq_hils also contains an additional "Stimuli Generator" hypercell. There is just one SG in the seq_hils, and it should always be evaluated first. We have a specialized scheduler entity for the SG which is much more basic than the other schedulers: it enables execution of the SG whenever it is unstable, without requiring any bookkeeping. The SG_SCH entity has an interface which is similar to the interface of all other scheduling entities, but the implementation is much easier: the entire implementation is given by figure 6.3.

```
process(deltacycle,request)
begin
  valid <= '0';
  for i in 0 to SCHEDULE'high loop
    if i >= deltacycle then
      if request(SCHEDULE(i)) = '1' then
        grant_int <= SCHEDULE(i);
        valid <= '1';
        exit;
      end if;
    end if;
  end loop;
end process;
```

Figure 6.2: Implementation of the scheduling logic for a single scheduling entity

```
grant <= 0;
valid <= request(0) AND nrst;
```

Figure 6.3: Complete scheduling implementation for the SG scheduler

With a scheduling entity for each hypercell (including the special SC scheduler) coupled together in a `SCH` Scheduler entity which is a modified version of the original code, we have a complete drop-in replacement for the original scheduler entity with our new combined online and offline scheduling algorithms.

### 6.2.3 Synthesis results

In order to compare the required hardware resources of the new scheduler with the round robin arbiter, both scheduling entities (i.e. a separate round robin arbiter and an online scheduling entity) have been isolated and synthesized using Mentor Graphics' Precision. A round robin arbiter for scheduling 64 cells was synthesized. The online scheduler entity contained a schedule for 64 cells with a makespan of 133 delta cycles.

Unfortunately, due to a problem with the tool, floor planning of the entities failed: the tool kept randomly crashing during the process. Only RTL synthesis of both entities was done. Therefore, both the required FPGA area and the prospected clock frequency are indicative and not reliable.

Both entities were synthesized for execution on a Xilinx Virtex 4 FPGA

(4VLX200FF1513). The synthesis of the round robin arbiter led to a RTL design requiring 263 CLB slices and 6 latches. The expected clock frequency is about 160MHz. Synthesis of the new online scheduling entity led to a RTL design of 968 CLB slices and 78 latches, with an expected clock frequency of approximately 270MHz.

Since the online scheduler contains a hypercell specific schedule, it was expected that it would be larger than the round robin arbiter. The size of the scheduler does not only grow with increasing number of cells per hypercell, but also with the makespan of the entire system: the scalability is not as good as that for the round robin arbiter.

As far as the RTL synthesis can show, the timing for the online scheduling entity does not form a problem for use in the seq_hils.

Since there is a great degree of freedom in implementation, is might be possible to build a more area-efficient version of the scheduling entity (e.g. by restructuring the simulator to detect a new system cycle globally instead of per scheduler entity), but that will remain future work.

## 6.3 Testing and results

In order to test the performance of the implementation, two complex (although somewhat synthetic) designs were simulated using a generated seq_hils. Both designs are available as EDIF netlist. The seq_hils is completely generated using the tool chain. The generated seq_hils simulators are simulated using Mentor Graphics' Questasim.

In the next sections, we will introduce the two test cases, and show the timing results gathered from simulation with both the old and the new schedulers.

### 6.3.1 Two test cases

The first test case is a design of a large Network-on-Chip. The design consists of 64 GuarVC routers in a $8 \times 8$ Mesh topology. No processing elements are connected to the network. The stimuli-generator in the seq_hils is used to inject packets into the network: each router sends several "best effort" packages (containing 15 words of data) at once to routers 3 to 5 nodes away in the network. We insert this bulk of packages into the network at system cycle 16, and simulate until all routers are done processing this data.

Although this is not a realistic test case (since there are no processing elements), it is a nice stress-test on the routers, and the massive amount of interconnected data flows also provide a heavy-load test for the simulator.

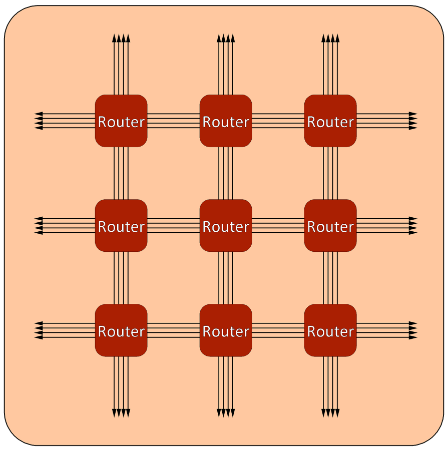


Figure 6.4: Design of a Network-on-Chip in a mesh topology, without connected processing elements

In total, this first test case has 64 cells mapped onto 1 hypercell.

The second test case is a more complex design. Again, we have an 8x8 mesh topology Network-on-Chip. But this time, 63 of the routers have a processing element connected to it: a ZPU general purpose processor. Each ZPU is also connected to a block of memory, from which it fetches the program instructions.

The memory contains the program which is to be executed on the ZPU: the program reads an integer from the NoC, adds 1 to this value, and sends it to the next node in the chain. The chain of ZPUs is 63 elements long: the 64th position in the network has no processing element; the test bench uses that position to monitor the behavior: each integer inserted into the network from the empty spot, should come back increased by 63.

Although the program running on the ZPUs is not useful in a real-world application, the situation of a 8x8 network on chip with processing elements sending data over the network is quite representative for the group of applications which could be simulated using the seq_hils.

The entire design of 64 routers, 63 ZPUs and 63 blocks of memory (190 cells in total) will be mapped on 3 different hypercells. We simulate 255 system cycles of this design: since executing a single instruction on the ZPU takes about 4 cycles, this is enough to run the prolog of the program once and the main loop (fetching data from the network, doing a calculation and sending the value onto the network) several times.

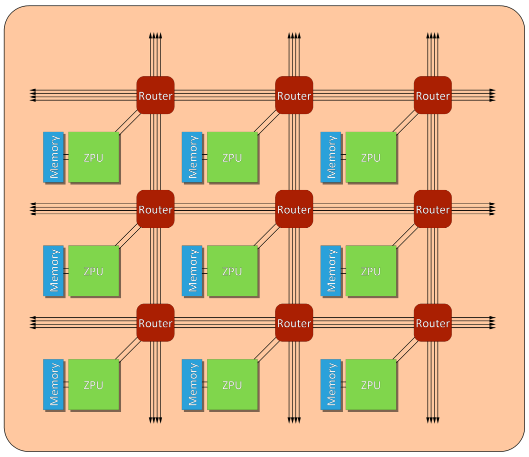


Figure 6.5: Design of a Network-on-Chip in a mesh topology, with ZPU processing elements connected

### 6.3.2 Test results

For both test cases, the seq_hils simulation environment was generated using the tool chain. The seq_hils was simulated twice for both test cases: once with the original round robin arbiter and once with the new scheduling in place. For each system cycle we simulate, we count the number of delta cycles required to stabilize the system.

The results for both testcases are plotted in figures 6.6 and 6.7. Each figure shows the number of delta cycles per system cycle, and contains both runs shown in different colors.

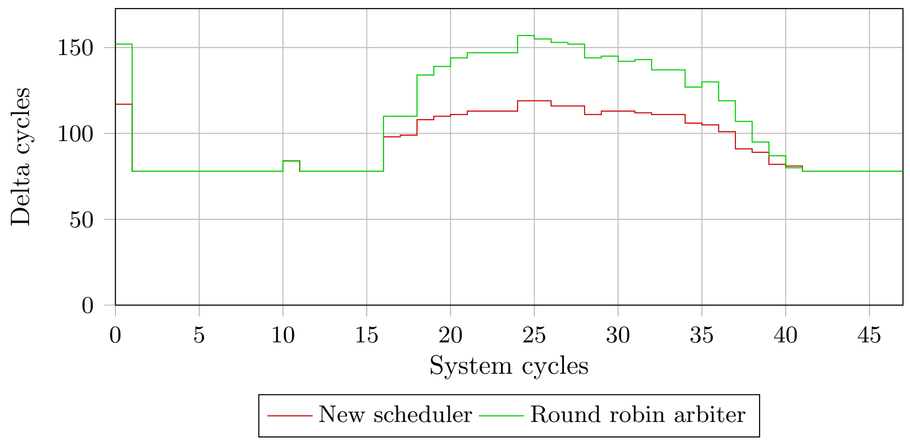


Figure 6.6: Delta cycles required to stabilize each system cycle in the simulation of an 8 × 8 NoC without processing elements, but with synthetic packet injection

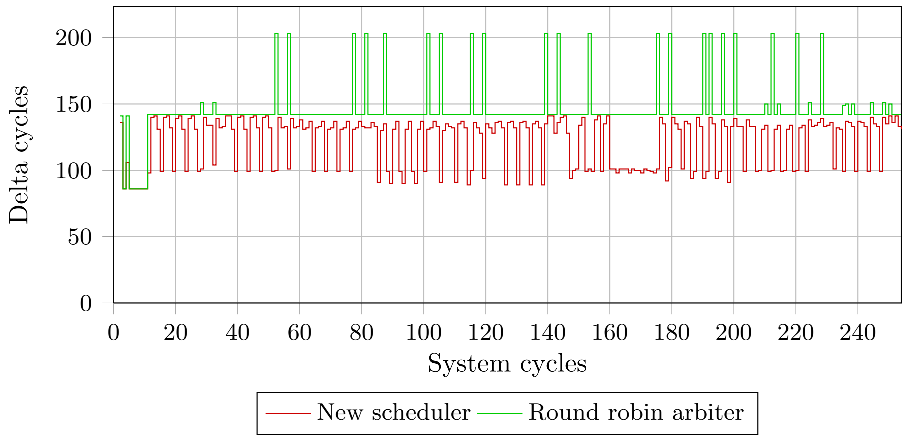


Figure 6.7: Delta cycles required to stabalize each system cycle in simulation of a large NoC with 63 ZPU tiles

From these graphs we can see that in just one system cycle, our scheduling algorithm is 1 delta cycle slower (cycle 40 in figure 6.6). In all other system cycles, we perform either as good as the round robin arbiter, or much better.

A nice note about simulation of the second test case: our worst-case makespan from the offline scheduling (and thus the base for the online scheduler) is 133 delta cycles: with flushing of the pipelines included, this worst-case is almost 3% better than the average number of delta cycles required by the round robin arbiter. It is even 30% better than the peaks observed by scheduling with the round robin arbiter!

On average, the performance increase provided by the new scheduler in simulation of the first test case is 12,8% and 16,7% for the second test case. This is quite close to what was expected based on the metasim results we got earlier.

## 6.4 Conclusion

In this chapter we discussed the implementation of the online and offline scheduling algorithms which were chosen during the previous chapters.

The offline scheduling was implemented in the Java tool chain, and was able to schedule the large test designs without a significant delay on the entire seq_hils generation process.

The online scheduling was implemented in a couple of VHDL entities. Those entities form a drop-in replacement for the original round robin arbiter based scheduler: the top-level RTL entity has an identical interface and behaves similar to input. The scheduling itself is completely combinational and thus adds no additional pipeline stages to the simulator. Although a complete synthesis was not performed, RTL synthesis of the new scheduler showed a higher demand on hardware usage on the FPGA, but also indicated that no significant slow down the simulation clock frequency is to be expected.

Simulating two larger test designs showed that the offline scheduler provided a worst-case schedule that has a makespan which is already slightly better than the average number of cycles required by the round robin arbiter for stabilization of the design. With the online scheduler in place, we gain an overall speed-increase of approximately 15% with both test cases. This closely matches the expected improvement from the metasim simulation in chapter 5. We therefore expect this new scheduler to be generic improvement to the seq_hils, which speeds up other possible simulations in the seq_hils as well.

CHAPTER 7

# Conclusions and Future work

During this project we designed and implemented a new scheduling approach for scheduling the seq_hils. This chapter will give an overview of what was achieved, and summarizes the topics open for future work.

## 7.1 Conclusions

Rutgers [17] developed a hardware simulation environment, which incorporates time multiplexing to allow the hardware design to be run completely in a FPGA. One of the subjects that required future work was the scheduler: the implemented round robin arbiter does not take any dependencies between cells into account. It was expected that a proper scheduler could drastically improve performance. In this thesis we took the challenge of improving the seq_hils's scheduler.

The DUV is partitioned by the developer into different cells. Similar cells are mapped onto a small set of hypercells. Since the cells are considered to be Mealy machines, each cells output depends on its inputs and its current state. Dependencies between cells consist of interconnections between cells and combinational paths through cells.

In order to be able to effectively test the performance of several scheduling approaches, we started with developing a simulator which simulates the simulators behavior: metasim. It incorporates a Monte Carlo simulation technique to simulate a non-behavioral probabilistic model of the cells in the DUV. The input for the simulator is the derived dependency graph of the DUV, it simulates the evaluation of a number of system cycles, and returns the average number of delta cycles per system cycle required to stabilize the DUV with the selected scheduling algorithm.

For the deducting of the dependency graph, a new algorithm was developed: each top-level input and output port of the DUV form a node (annotated with the cell they belong to) in the dependency graph. Interconnections between the cells are explicitly available and are copied into the dependency graph. Combinational path through the cells are deducted using a breadth-first search from each input port through the netlist of a cell. The search is cut-off at a flipflop. If an output port can be reached, the connection between input and output is also added to the dependency graph.

The final dependency graph will contain all possible data paths through the DUV. If all deducted data dependencies are taken into consideration for scheduling, we can construct a schedule offline which can guarantee stabilization of the system.

For the scheduling of the seq_hils we have implemented an hybrid online/offline scheduling approach. We chose for a computational intensive initial scheduling offline. This schedule will be optimized during simulation by a simple (low overhead) online scheduler.

We have shown that finding an optimal schedule is only feasible for small instances. For larger instances both computation time and memory usage increase exponentially. However, effective heuristically approaches exist. We introduced two heuristical approaches for offline scheduling based on the dependency paths, which both have a linear time and space complexity. This allows us to try both heuristical approaches on large designs, and choose the schedule with the best makespan, within tenths of seconds. The offline scheduling approaches were tested using the metasim simulator, and showed that a worst-case schedule (i.e. a schedule constrained by all data dependencies) would require 3 to 10% less delta cycles to stabilize a system cycle than the round robin arbiter would require.

As an online optimization approach, we suggest to start with the worst-case offline schedule, and use runtime information about which cells are instable, to skip over parts of the schedule if cells are already stable. Simulations using metasim, show that this approach can do another 5 to 10% percent performance increase on average. Based on the simulation results we expected the final performance increase to be about 12 to 15%.

The suggested algorithms were implemented in the tool chain which generates the seq_hils. The netlist analysis for dependency deduction and the offline scheduling heuristics are implemented as subtasks of the tool. The online scheduling entities are written in VHDL, and form a drop-in replacement of the original scheduler.

The implementation is tested with two realistic complex designs (although the system-input was synthetic in nature): two $8 \times 8$ mesh topology network on chip design. One design contains no processing elements; a test bench injects packets into network. This testcase contained 64 cells, which were mapped onto a single hypercell. The second design contains 63 ZPU processing elements with a memory element connected to each ZPU: the 190 cells were mapped onto 3 different hypercells.

Simulation of the seq_hils for both test cases showed a significant performance increase: Our new scheduler performs equally or outperforms the round robin arbiter in each system cycle (except for one, in which we are 1 delta cycle slower). In both cases the average number of delta cycles required to stabilize the system was decreased by about 18

The actual performance increase matches the expected performance increase based on previous metasim simulation results. It thus shows that our simulation model was accurate enough and our conclusions based on the simulation were valid.

Although there was no initial indication of what performance increase would be realistically be reachable, the final results are not as spectacular as hoped for. Nonetheless, more improvements on the scheduling are not expected to gain more than a few percent: the new scheduling is not optimal, but its performance is about as good as it gets.

Concluding, we can say that the goal of implementing a new scheduling algorithm for the seq_hils which, based on the knowledge about existing dependencies between cells, would outperform the round robin arbiter, is achieved. The new scheduler is a drop-on replacement for the old scheduling approach, and no user interaction is required to use the new scheduler. We have focused on decreasing the average number of delta cycles per system cycle, and achieved an overall performance increase of about 15%, which is expected to hold in a wide range of simulated designs.

The exact performance increase depends on the structure of the DUV, but since we explicitly use knowledge of data dependencies within the design, we can assume that more complex designs (with more dependencies and more combinational paths through cells) will have more profit from the new scheduler.

## 7.2 Future work

During the project of scheduling the seq_hils several interesting options were encountered, but not yet examined. These options were left open as future work. This section will summarize those options.

The first option left open for future work, is the online scheduling algorithm. The currently implemented algorithm is still very simple (but shows good results though). Several more complex approaches for online scheduling were thought of: gathering statistical information at runtime, might allow for better specialized optimizations over a longer run time. Implementation however might cause a huge overhead on required FPGA area. It might also be possible, to construct a more complex offline schedule which allows the online schedule to not just skip over cells in the schedule, but actually find an efficient path through to stabilization. Future research is needed to determine the feasibility of these approaches.

As an opposite option to more advanced online scheduling, we also opted to not do any online scheduling at all. Evaluating the cells in the order determined by an offline scheduler makes it possible to remove large parts of the overhead in the currently implemented logic: this will probably require more delta cycles per system cycle on average, but the system would be more flexible and scalable, and less logic might allow higher clock frequencies to be achieved. The metasim tests results suggest that the performance (the number of delta cycles required) would be comparable with (and perhaps even slightly better then) the original round robin based scheduler. Stripping the logic from the seq_hils and doing some tests is required to determine what the actual consequences will be.

Also, the RTL synthesis of the current VHDL implementation of the online scheduling, indicated that the required hardware area is much bigger than what was used by the round robin arbiter. However, the same algorithm could probably be implemented in many different ways: there might be a much smaller implementation possible. Improvements in the required hardware are accomplishable.

Furthermore, Rutgers [17] has proposed some scheduling related future work which still applies:

Currently, for each system cycle, each cell is evaluated at least once, since it is assumed that the internal state changes at each clock cycle. If we can check if the internal state actually is changed, we could increase performance even further without considerable modifications to the scheduling.

The partitioning of the DUV is currently a manual step in the process of generation a seq_hils. The ultimate goal is to work toward a complete automated tool. The offline scheduling results can take an important role in the automatical partition of the DUV. As shown in section 4.2, the dependency deduction and the offline scheduling give a quite tight upper bound on the required number of delta cycles for stabilization. This expected makespan is a good quantifier for a partitioning. This quantification can be used to pick the best partitioning from several options, and help in optimizing the hypercell instantiation.

APPENDIX A

# Simulation results for online scheduling algorithms

The different online scheduling approaches have been simulated using our metasim simulation environment. The individual results are listed in this appendix.

## A.1 Relative performance distribution

The histograms A.1 to A.8 show the distrubution of the relatative performance of the different online scheduling algorithms, which were gathered during the metasim simulation described in section 5.2. The relative performance of each run is obtained by indexing the required number of delta cycles to stabilize, with those required by the round robin arbiter.

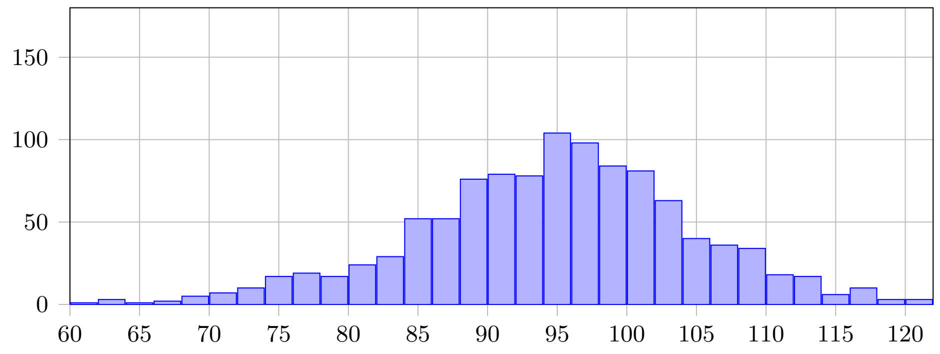


Figure A.1: Histogram of relative performance increase using the coaleascing heuristic offline scheduling only

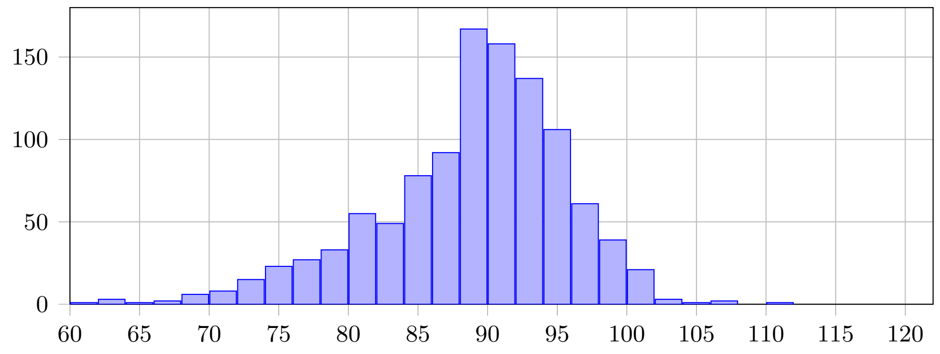


Figure A.2: Histogram of relative performance increase using the coaleascing heuristic offline with locally optimize online scheduling

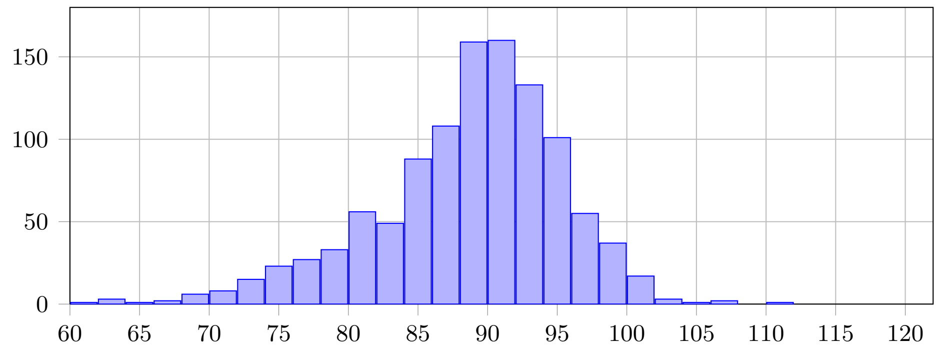


Figure A.3: Histogram of relative performance increase using coaleascing heuristic offline with globally optimize online scheduling

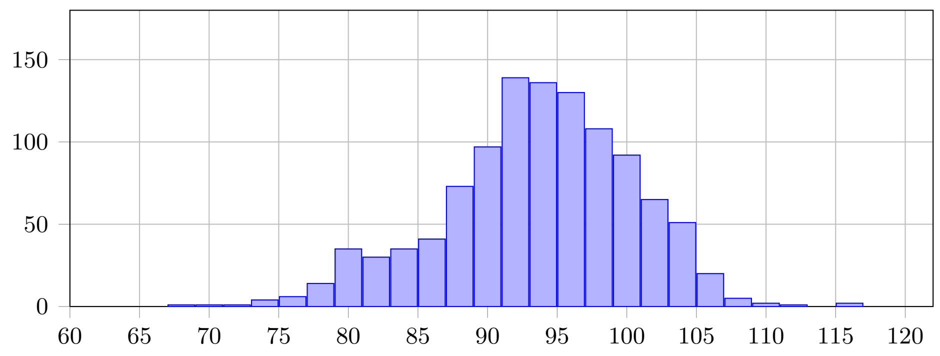


Figure A.4: Histogram of relative performance increase using coaleascing heuristic offline with fix best-case online scheduling

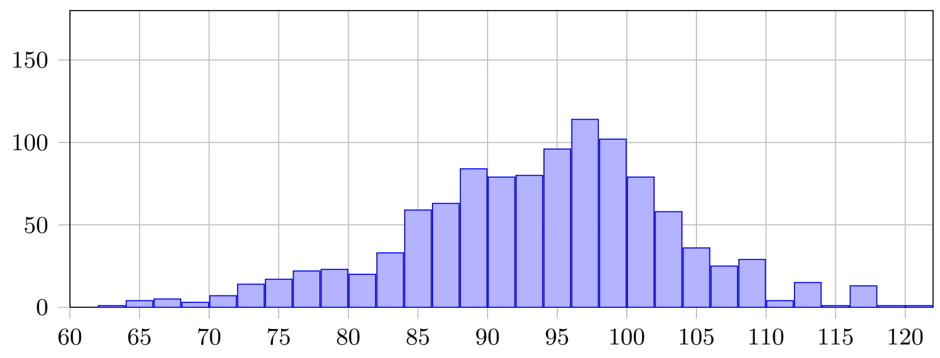


Figure A.5: Histogram of relative performance increase using SCS-based heuristic offline scheduling only

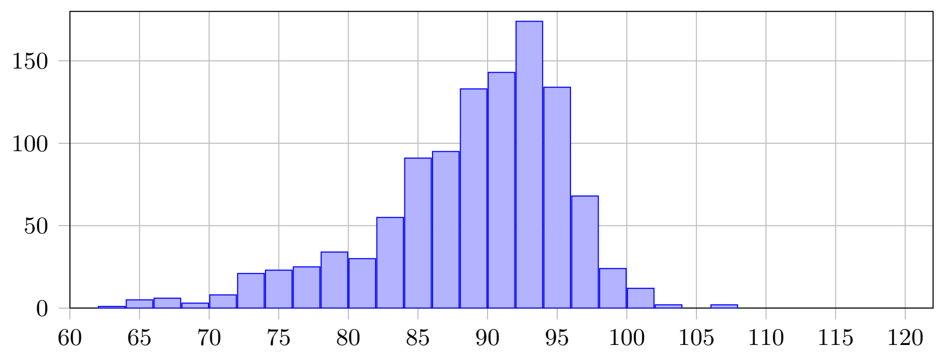


Figure A.6: Histogram of relative performance increase using SCS-based heuristic offline with locally optimize online scheduling

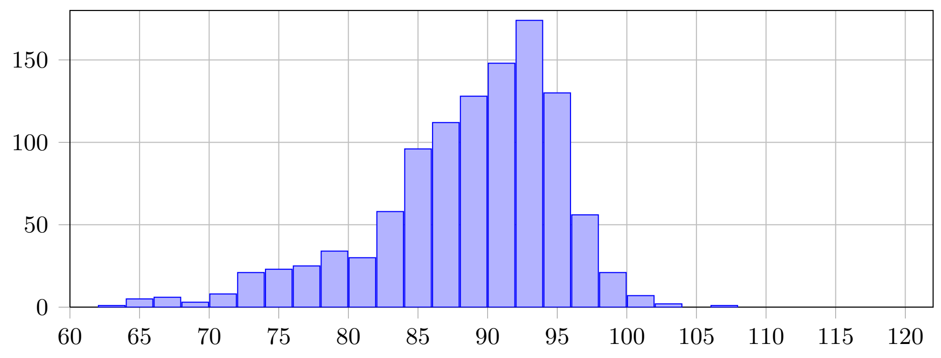


Figure A.7: Histogram of relative performance increase using SCS-based heuristic offline with globally optimize online scheduling

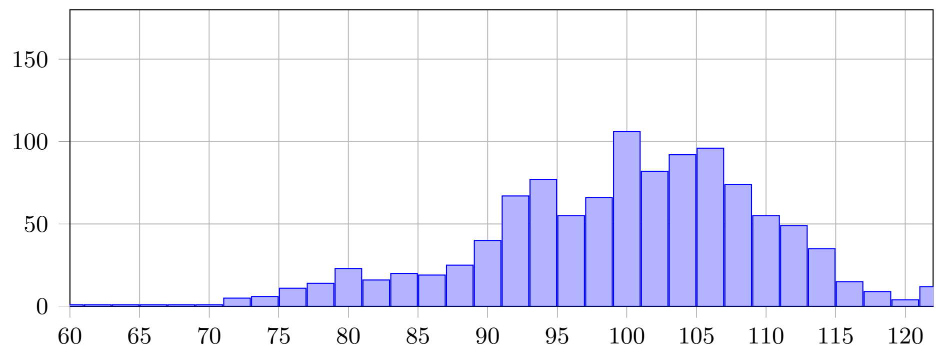


Figure A.8: Histogram of relative performance increase using SCS-based heuristic offline with fix best-case online scheduling

## A.2 Absolote performance results

The tables A.1 to A.3 show the average number of delta cycles required for stabilisation by each scheduling algorithm for the different test cases. The results are vertically grouped into four groups for the four different ranges of input designs. The following designs correspond to the four groups which were simulated:

- 50× a random structure consisting of 12 cells with 8 ports per cell
- 50× a random structure consisting of 25 cells with 20 ports per cell
- 20× a random structure consisting of 48 cells with 30 ports per cell
- 1× a $4 \times 3$ torus topology network on chip

| Parallel hypercells | Pipeline stages | Prob1 | Round robin arbiter | |
|---|---|---|---|---|
| # | # | % | $\mu$ | $\sigma$ |
| 1 | 1 | 1 | 21.03 | 0.19 |
| 1 | 1 | 10 | 21.66 | 0.23 |
| 1 | 1 | 100 | 25.01 | 0.73 |
| 1 | 7 | 1 | 34.43 | 0.26 |
| 1 | 7 | 10 | 36.36 | 0.39 |
| 1 | 7 | 100 | 44.19 | 1.20 |
| 2 | 7 | 1 | 23.69 | 0.13 |
| 2 | 7 | 10 | 26.28 | 0.35 |
| 2 | 7 | 100 | 31.05 | 0.61 |
| 1 | 1 | 1 | 45.90 | 0.43 |
| 1 | 1 | 10 | 46.47 | 0.50 |
| 1 | 1 | 100 | 50.35 | 1.40 |
| 1 | 7 | 1 | 58.72 | 0.50 |
| 1 | 7 | 10 | 59.93 | 0.65 |
| 1 | 7 | 100 | 66.56 | 1.85 |
| 2 | 7 | 1 | 36.24 | 0.24 |
| 2 | 7 | 10 | 37.49 | 0.44 |
| 2 | 7 | 100 | 42.30 | 1.03 |
| 1 | 1 | 1 | 87.96 | 0.58 |
| 1 | 1 | 10 | 88.44 | 0.60 |
| 1 | 1 | 100 | 91.80 | 1.34 |
| 1 | 7 | 1 | 99.78 | 0.60 |
| 1 | 7 | 10 | 101.05 | 0.67 |
| 1 | 7 | 100 | 106.30 | 1.65 |
| 2 | 7 | 1 | 56.95 | 0.29 |
| 2 | 7 | 10 | 57.61 | 0.41 |
| 2 | 7 | 100 | 61.28 | 0.88 |
| 1 | 1 | 1 | 21.81 | 0.00 |
| 1 | 1 | 10 | 22.54 | 0.00 |
| 1 | 1 | 100 | 24.00 | 0.00 |
| 1 | 7 | 1 | 35.71 | 0.00 |
| 1 | 7 | 10 | 38.84 | 0.00 |
| 1 | 7 | 100 | 46.30 | 0.00 |
| 2 | 7 | 1 | 24.83 | 0.00 |
| 2 | 7 | 10 | 28.60 | 0.00 |
| 2 | 7 | 100 | 33.63 | 0.00 |

Table A.1: Round robin based simulation results

| Parallel hypercells | Pipeline stages | Prob1 | Heuristic (Offline only) | | Heuristic (Optimize locally) | | Heuristic (Optimize globally) | | Heuristic (Fix worst-case) | |
|---|---|---|---|---|---|---|---|---|---|---|
| # | # | % | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| 1 | 1 | 1 | 17.86 | 1.54 | 17.56 | 1.28 | 17.56 | 1.28 | 17.40 | 1.06 |
| 1 | 1 | 10 | 18.07 | 1.55 | 17.73 | 1.33 | 17.73 | 1.33 | 18.26 | 1.04 |
| 1 | 1 | 100 | 19.46 | 1.95 | 19.42 | 1.95 | 19.42 | 1.95 | 22.74 | 2.09 |
| 1 | 7 | 1 | 31.91 | 1.81 | 30.26 | 1.89 | 30.26 | 1.89 | 28.95 | 0.86 |
| 1 | 7 | 10 | 32.56 | 1.68 | 31.28 | 1.59 | 31.28 | 1.59 | 34.37 | 1.08 |
| 1 | 7 | 100 | 35.10 | 2.68 | 34.96 | 2.53 | 34.96 | 2.53 | 43.28 | 1.60 |
| 2 | 7 | 1 | 25.28 | 1.47 | 21.82 | 0.81 | 21.44 | 0.67 | 21.51 | 0.58 |
| 2 | 7 | 10 | 25.91 | 1.21 | 23.89 | 0.57 | 23.64 | 0.63 | 26.59 | 0.71 |
| 2 | 7 | 100 | 27.70 | 2.17 | 27.00 | 1.30 | 26.96 | 1.28 | 31.90 | 0.67 |
| 1 | 1 | 1 | 44.78 | 1.68 | 44.04 | 1.52 | 44.04 | 1.52 | 43.58 | 1.55 |
| 1 | 1 | 10 | 45.14 | 1.65 | 44.22 | 1.52 | 44.22 | 1.51 | 44.01 | 1.52 |
| 1 | 1 | 100 | 47.12 | 2.08 | 47.08 | 2.11 | 47.08 | 2.11 | 47.76 | 2.36 |
| 1 | 7 | 1 | 56.80 | 3.13 | 55.32 | 2.53 | 55.32 | 2.53 | 54.63 | 2.19 |
| 1 | 7 | 10 | 57.08 | 2.90 | 55.72 | 2.37 | 55.72 | 2.37 | 57.72 | 2.67 |
| 1 | 7 | 100 | 58.64 | 2.73 | 58.62 | 2.72 | 58.62 | 2.72 | 66.40 | 3.82 |
| 2 | 7 | 1 | 35.86 | 2.19 | 34.62 | 1.90 | 34.34 | 1.86 | 33.53 | 1.22 |
| 2 | 7 | 10 | 36.36 | 1.89 | 35.49 | 1.59 | 35.26 | 1.59 | 37.84 | 1.34 |
| 2 | 7 | 100 | 38.64 | 1.98 | 38.64 | 1.98 | 38.64 | 1.98 | 44.32 | 1.67 |
| 1 | 1 | 1 | 84.20 | 2.66 | 83.45 | 2.44 | 83.45 | 2.44 | 83.95 | 2.06 |
| 1 | 1 | 10 | 84.40 | 2.77 | 83.61 | 2.47 | 83.61 | 2.47 | 84.29 | 2.27 |
| 1 | 1 | 100 | 86.00 | 3.21 | 85.95 | 3.11 | 85.95 | 3.11 | 87.50 | 3.37 |
| 1 | 7 | 1 | 92.35 | 2.39 | 90.90 | 2.36 | 90.90 | 2.36 | 90.22 | 2.61 |
| 1 | 7 | 10 | 92.50 | 2.50 | 91.12 | 2.39 | 91.12 | 2.38 | 92.75 | 2.72 |
| 1 | 7 | 100 | 93.90 | 2.95 | 93.75 | 2.59 | 93.75 | 2.59 | 100.45 | 3.38 |
| 2 | 7 | 1 | 54.05 | 2.06 | 53.40 | 2.01 | 53.15 | 2.06 | 52.68 | 2.03 |
| 2 | 7 | 10 | 54.15 | 1.99 | 53.59 | 1.87 | 53.33 | 1.95 | 55.33 | 2.29 |
| 2 | 7 | 100 | 55.30 | 2.17 | 55.25 | 2.09 | 55.25 | 2.09 | 61.45 | 2.40 |
| 1 | 1 | 1 | 23.00 | 0.00 | 23.00 | 0.00 | 23.00 | 0.00 | 23.00 | 0.00 |
| 1 | 1 | 10 | 23.00 | 0.00 | 23.00 | 0.00 | 23.00 | 0.00 | 23.00 | 0.00 |
| 1 | 1 | 100 | 24.00 | 0.00 | 24.00 | 0.00 | 24.00 | 0.00 | 24.00 | 0.00 |
| 1 | 7 | 1 | 35.11 | 0.00 | 32.95 | 0.00 | 32.92 | 0.00 | 33.38 | 0.00 |
| 1 | 7 | 10 | 40.00 | 0.00 | 36.54 | 0.00 | 36.56 | 0.00 | 40.28 | 0.00 |
| 1 | 7 | 100 | 43.00 | 0.00 | 43.00 | 0.00 | 43.00 | 0.00 | 48.00 | 0.00 |
| 2 | 7 | 1 | 30.00 | 0.00 | 25.00 | 0.00 | 24.07 | 0.00 | 24.61 | 0.00 |
| 2 | 7 | 10 | 31.00 | 0.00 | 28.19 | 0.00 | 28.00 | 0.00 | 30.00 | 0.00 |
| 2 | 7 | 100 | 37.00 | 0.00 | 37.00 | 0.00 | 37.00 | 0.00 | 34.00 | 0.00 |

Table A.2: Simulation results of our own heuristical approach

| Parallel hypercells | Pipeline stages | Prob1 | SCS based (Offline only) | | SCS based (Optimize locally) | | SCS based (Optimize globally) | | SCS based (Fix worst-case) | |
|---|---|---|---|---|---|---|---|---|---|---|
| # | # | % | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| 1 | 1 | 1 | 18.12 | 1.64 | 17.60 | 1.40 | 17.60 | 1.40 | 17.94 | 1.27 |
| 1 | 1 | 10 | 18.15 | 1.63 | 17.62 | 1.40 | 17.62 | 1.40 | 18.14 | 1.24 |
| 1 | 1 | 100 | 18.72 | 1.51 | 18.72 | 1.51 | 18.72 | 1.51 | 20.20 | 1.70 |
| 1 | 7 | 1 | 32.92 | 2.11 | 30.74 | 1.80 | 30.74 | 1.80 | 36.35 | 2.46 |
| 1 | 7 | 10 | 33.39 | 1.81 | 31.48 | 1.54 | 31.48 | 1.53 | 37.65 | 1.92 |
| 1 | 7 | 100 | 35.26 | 1.90 | 35.26 | 1.90 | 35.26 | 1.90 | 42.88 | 1.81 |
| 2 | 7 | 1 | 25.33 | 1.67 | 21.99 | 0.94 | 21.55 | 0.69 | 28.30 | 1.75 |
| 2 | 7 | 10 | 25.68 | 1.22 | 23.43 | 0.71 | 23.02 | 0.59 | 29.14 | 1.05 |
| 2 | 7 | 100 | 26.80 | 0.94 | 26.80 | 0.94 | 26.80 | 0.94 | 31.72 | 0.83 |
| 1 | 1 | 1 | 45.02 | 1.92 | 43.70 | 1.35 | 43.70 | 1.35 | 44.10 | 1.37 |
| 1 | 1 | 10 | 45.06 | 1.88 | 43.75 | 1.37 | 43.75 | 1.38 | 44.34 | 1.42 |
| 1 | 1 | 100 | 45.48 | 1.84 | 45.48 | 1.84 | 45.48 | 1.84 | 47.24 | 1.90 |
| 1 | 7 | 1 | 58.44 | 2.27 | 56.20 | 1.83 | 56.20 | 1.83 | 62.25 | 2.49 |
| 1 | 7 | 10 | 58.57 | 2.14 | 56.50 | 1.74 | 56.51 | 1.72 | 63.13 | 2.25 |
| 1 | 7 | 100 | 59.60 | 1.88 | 59.60 | 1.88 | 59.60 | 1.88 | 68.44 | 3.26 |
| 2 | 7 | 1 | 36.48 | 1.37 | 35.06 | 0.97 | 34.70 | 0.81 | 40.97 | 1.34 |
| 2 | 7 | 10 | 36.70 | 1.22 | 35.42 | 0.89 | 35.06 | 0.78 | 41.58 | 1.06 |
| 2 | 7 | 100 | 37.74 | 1.37 | 37.74 | 1.37 | 37.74 | 1.37 | 45.24 | 1.68 |
| 1 | 1 | 1 | 83.65 | 3.05 | 82.25 | 2.53 | 82.25 | 2.53 | 83.25 | 2.59 |
| 1 | 1 | 10 | 83.65 | 3.05 | 82.25 | 2.53 | 82.25 | 2.54 | 83.38 | 2.64 |
| 1 | 1 | 100 | 83.85 | 3.13 | 83.85 | 3.13 | 83.85 | 3.13 | 85.80 | 3.17 |
| 1 | 7 | 1 | 97.60 | 2.46 | 95.60 | 1.74 | 95.60 | 1.74 | 101.65 | 1.71 |
| 1 | 7 | 10 | 97.60 | 2.46 | 95.73 | 1.77 | 95.72 | 1.77 | 102.06 | 1.82 |
| 1 | 7 | 100 | 98.00 | 2.63 | 98.00 | 2.63 | 98.00 | 2.63 | 105.60 | 2.97 |
| 2 | 7 | 1 | 55.60 | 1.56 | 54.20 | 1.21 | 53.90 | 1.18 | 60.75 | 1.09 |
| 2 | 7 | 10 | 55.60 | 1.56 | 54.34 | 1.15 | 54.04 | 1.12 | 60.98 | 1.29 |
| 2 | 7 | 100 | 55.85 | 1.77 | 55.85 | 1.77 | 55.85 | 1.77 | 63.50 | 1.75 |
| 1 | 1 | 1 | 23.00 | 0.00 | 23.00 | 0.00 | 23.00 | 0.00 | 22.00 | 0.00 |
| 1 | 1 | 10 | 23.00 | 0.00 | 23.00 | 0.00 | 23.00 | 0.00 | 22.99 | 0.00 |
| 1 | 1 | 100 | 24.00 | 0.00 | 24.00 | 0.00 | 24.00 | 0.00 | 24.00 | 0.00 |
| 1 | 7 | 1 | 37.74 | 0.00 | 32.97 | 0.00 | 32.96 | 0.00 | 35.87 | 0.00 |
| 1 | 7 | 10 | 40.87 | 0.00 | 36.65 | 0.00 | 36.66 | 0.00 | 42.15 | 0.00 |
| 1 | 7 | 100 | 44.00 | 0.00 | 44.00 | 0.00 | 44.00 | 0.00 | 48.00 | 0.00 |
| 2 | 7 | 1 | 26.95 | 0.00 | 24.99 | 0.00 | 24.97 | 0.00 | 25.63 | 0.00 |
| 2 | 7 | 10 | 29.99 | 0.00 | 28.07 | 0.00 | 28.01 | 0.00 | 31.00 | 0.00 |
| 2 | 7 | 100 | 33.00 | 0.00 | 33.00 | 0.00 | 33.00 | 0.00 | 35.00 | 0.00 |

Table A.3: Simulation results with SCS based scheduling

# Bibliography

[1] Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[2] Zeev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joe D. Rutledge. Hss–a high-speed simulator. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 6(4):601–617, 1987.

[3] Richard Buchmann and Alain Greiner. A fully static scheduling approach for fast cycle accurate systemc simulation of mpsocs. December 2007.

[4] Srihari Cadambi, Chandra S Mulpuri, and Pranav N Ashar. A fast, inexpensice and scalable hardware acceleration technique for functional simulation. In *DAC 2002*, 2002.

[5] R. Gary Parker. *Deterministic Scheduling Theory.* Chapman & Hall, 1995.

[6] Denis Hommais and Frédéric Pétrot. Efficient combinational loops handling for cycle precise simulation of system on a chip. 1998.

[7] Wen-Qi Huang and Zhi Huang. Algorithm based on taboo search and shifting bottleneck for job shop scheduling. *Journal of Computer Science and Technology*, 19(6):776–781, 2004.

[8] Tao Jiang and Ming Li. On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, 1994.

[9] Nicholas Metropolis and S. Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[10] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[11] Kang Ning and Hon Wai Leong. Towards a better solution to the shortest common supersequence problem: the deposition and reduction algorithm. In *Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences - Volume 1 (IMSCCS'06)*, 2006.

[12] David A. Penry and David I. August. Optimizations for a simulator construction system supporting reusable components. In *In Proceedings of the 40th Design Automation Conference*, 2003.

[13] Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam. A new optimized implementation of the systemc engine using acyclic scheduling. In *Proceedings of the Design, Automation and Test in Europ Conference and Exhibition (DATE'04)*, 2004.

[14] Frédéric Pétrot, Denis Hommais, and Alain Greiner. Cycle precise code based hardware/software system simulation with predictable event propagation. 1997.

[15] Michael L. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer, 2005.

[16] S.B. Roodenburg. *Research in preparation of scheduling the sequential hardware-in-the-loop simulator*. University of Twente, 2009.

[17] J.H. Rutgers. *Advanced automated sequential hardware-in-the-loop simulator generation*. University of Twente, 2008.

[18] Andreas Schirmer. *New Insights on the Complexity of Resource-Constrainerd Project Scheduling*. Christian-Albrechts-Universit́at zu Kiel, 2005.

[19] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb. Tiers: Topology independent pipelined routing and scheduling for virtualwire compilation. In *Field-Programmable Gate Arrays, 1995. FPGA '95. Proceedings of the Third International ACM Symposium on*, pages 25–31, 1995.

[20] James E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[21] Russell Tessier, Jonathan Babb, Matthew Dahl, Silvina Hanono, and Anant Agarwal. The virtual wires emulation system: A gate-efficient asic prototyping environment. In *Proceedings of the 1994 ACM Workshop on FPGA's (FPGA '94)*, Februari 1994.

[22] P.T. Wolkotte. *Exploration within the Network-on-Chip paradigm.* University of Twente, 2008.