TU/e technische universiteit eindhoven

Department of Electrical Engineering
Section Design Technology (ICS/ES)

Master's Thesis

Concurrency in Computational Networks

S. Stuijk
(446407)

Supervisors:	dr. ir. A.A. Basten
	prof. dr. ir. J.L. van Meerbergen
Date:	October 2002

Abstract

Future generations of embedded multi-media systems will have an increasing need for compute platforms that combine high compute power with low energy consumption. To meet with these requirements, multi-processor systems must be used. These systems are in nature concurrent, and this concurrency in the architecture should be exploited. This requires that the concurrency is used in the mapping trajectory from the system specification to the hardware architecture. The concurrency in an application should therefore be extracted, and made explicit, in the models that are used to specify a system. To support the extraction of concurrency from an application, the model must contain a concurrency model. This concurrency model should support formal reasoning about concurrency.

This thesis present a model of computation that can be used to explicitly specify the concurrency in an application. This model of computation, the computational-network model, is based on the Kahn process network model. The computational-network model is extended with a concurrency model. This concurrency model is comprised of five main measures supported by a set of detailed measures. These measures provide an insight in the concurrency properties of the specified system. The thesis presents also a first implementation of the presented model of computation and concurrency model.

The thesis presents a case study that shows that all of these five measures are meaningful and do not overlap. These experiments suggest further that the measures are sufficient for obtaining good results; no more measures are needed. The design case shows also that it is possible to perform a concurrency optimization that is architecture independent and to obtain results similar to an optimization performed by an experienced designer that optimizes the application for a given architecture.

Acknowledgments

Over the last 9 months I had the opportunity to work in an excellent and inspiring environment, which eventually resulted in this thesis. I would like to thank Jef van Meerbergen, my graduation professor, for providing me that option. I would also like to thank Twan Basten, my supervisor, for giving me the opportunity to develop my own ideas. Though I always tried to come up with proven and watertight ideas, he always managed to find some bottlenecks to urge and motivate me to search for better solutions. From the early stages of this thesis until the final version, he was always able to help me structure my thesis and give useful feedback on how it could be improved. I also want to thank him for the many fruitful discussions we had about the subject. I also owe thanks to Johan Lukkien for taking place in my graduation commission and the useful suggestions that he gave me for finding related work.

I would also like to thank Jan Hoogerbrugge and Paul Stravers. They created the CAKE architecture and simulations environment which I used to prove that my models are working. I also want to thank Erwin de Kock for giving me a draft version of an article about a JPEG decoder case study that he performed. He also gave me the source code of all JPEG decoders presented in this article and he was the principal developer of the YAPI library. The article, the code and YAPI have proven to be very useful in this project.

Finally, I would like to thank Dirk and Etienne, my roommates, and all people of the ES group that are present every day in the groups coffee room for the many nice and interesting discussions we had during the coffee breaks.

Contents

Chapter 1

Introduction

Motivation. Multimedia systems are characterized by an ever-increasing need for compute power. This compute power is needed for the processing of large amounts of data such as images, video and speech. The demand for growing compute power is combined with high energy-efficiency constraints and is satisfied by integrating many average-speed and energy-efficient processing elements on a single chip. The integration of processing elements is resulting in the development of both heterogeneous and homogeneous multi-processor systems. These multi-processor systems are inherently concurrent, as they contain many processing elements that operate in parallel. With a trend toward larger and complexer multi-processor architectures, there will be even more parallelism to exploit. This requires that the parallelism available in an application, which is mapped onto the multi-processor system, is made visible in the mapping trajectory.
A number of different formal models of computation are used in concurrency theory. They express concurrency using formal languages [3], partial orders [30] or automata [7] at different levels of abstraction. Prime examples are Dataflow graphs [23], Petri nets [29] and Kahn process networks [13, 14]. These models of computation make the parallelism in an application visible. They are however not able to answer questions such as: How concurrent is my application? Does the specification maximally exploit the concurrency inherent in the application? How must the specification be modified to better exploit the concurrency in the application? Does the specification optimally exploit the concurrency inherent in the target architecture?
As mentioned, next-generation multi-media systems will use multi-processor architectures to meet the high performance and low energy constraints set by these systems. The concurrency in an application must be made explicit in the specification to exploit the concurrency inherent in these architectures. Such a specification should allow formal reasoning about the concurrency in the application and about how to exploit this concurrency. To date, no formal models exist that are able to do this without fully implementing the system.
This thesis describes a model of computation and accompanying concurrency model that allows formal reasoning about the concurrency in an application without fully implementing the system.

Overview. This thesis is divided in seven main parts: problem definition, model of computation, concurrency model, implementation, design exploration method, experimental results, and conclusions and recommendations. In Chapter 2, we describe the problem definition. Chapter 3 presents a model of computation, namely the model of computational networks, for describing a computation that is performed in a distributed system. A formal concurrency model is presented in Chapter 4. This model can be used to analyze the concurrency in computational networks. In Chapter 5, the implementation of the concurrency model is discussed. A design exploration method that uses the concurrency model is presented in Chapter 6. Chapter 7 demonstrates a design case. This case shows how to convert a computational network into another computational network that better exploits the available concurrency. Finally the conclusions and recommendations are discussed. This includes shortcomings of the concurrency model. Also, further improvements in the software implementation are discussed.

Chapter 2

Problem Definition

2.1 Introduction

Future generations of embedded multi-media systems such as hand-held computers, mobile phones, gaming devices, car navigation systems, etc., will have an increasing need for compute platforms that combine high compute power with a low energy consumption. The compute power is needed for the processing of large amounts of data such as images and video, and for applications such as speech recognition and synthesis that can be used to improve user interfaces. The systems must be energy-efficient because they often have only a limited energy supply, typically consisting of a rechargeable battery. Furthermore, a high energy dissipation means that a lot of heat is generated which, in turn, requires a lot of cooling. However, cooling technology takes a lot of space, is relatively expensive, and is susceptible to disturbances.
Engineers are looking for ways to meet both the high compute power and low energy dissipation requirements. One common aspect of many approaches is the exploitation of parallelism in one way or another, as the requirements cannot be achieved via single-processor technology. The following example explains why, and it shows why multi-processor technology can meet the requirements.

Example 2.1 Consider a system that has to realize a computation (e.g., JPEG or MPEG decoding). The first solution might be to realize this computation using a system that consists of a single processor (single-processor system). This single-processor system has a switching capacitance C, and runs at a supply voltage V with a frequency f. The power dissipation of the single-processor system is equal to: $P = fCV^2$ [6].
The computation can also be realized using a system with two processors in parallel (multi-processor system). These processors can operate at a frequency $f/2$. The same amount of work is then still done in the same time as in a single-processor system; assuming the overhead of the multi-processor system is negligible. However, the switching capacitance of the multi-processor system is $2C$. The supply voltage of the multi-processor system is $V' < V$. This voltage can be lower because of the lower switching delay frequency used in the multi-processor system. The power dissipation of the multi-processor

system is then equal to: $P' = fCV'^2$.
The ratio between the power dissipation of the multi-processor system and the single-processor system is:

$$\frac{P'}{P} = \frac{fCV'^2}{fCV^2} = \frac{V'^2}{V^2}$$

The power dissipation of the multi-processor system drops with the square of the difference between the voltage of the multi- and single-processor. Note that both solutions have the same compute power, but only a different energy consumption. ■

This example shows that when a multi-processor system is used, the energy consumption is reduced. A multi-processor system realizes in this way the low energy constraints imposed by future embedded multi-media systems. It can also full-fill the need for high compute power by using enough processors in the system. The single-processor technology can meet the high compute power constraint, but this comes with a higher energy consumption than in the multi-processor solution.

There is, besides the high compute power and low energy dissipation constraint, another reason for considering concurrency. This is the wiring delay problem [27], the delay introduced by on-chip wiring is getting so large in future IC designs that only a very small percentage of the die will be reachable during a single clock cycle. The whole die can as a result no longer be used for one single processor. Locality must be introduced in the architecture to be able to use the whole die. Locality means that a lot of processors will be integrated on a single die. This results in multi-processor systems-on-chip.

The trends described above will lead to systems in which a lot of parallelism is available. The important question is now how this parallelism can be exploited. This requires that the concurrency in the application is made explicit and is adapted to the parallelism in the multi-processor system onto which it is mapped. To solve this problem, we have to answer three questions. First, what do future system architectures look like? Second, how can the concurrency in an application be made explicit without the need to fully implement the system? Third, how do we map a specification onto a multi-processor system? Section 2.2 tries to answer the first question by describing the trends in modern system architectures. Section 2.3 describes the problems that arise when the second question is answered. The third question, the problem of multi-processor mapping is discussed in Section 2.4.

2.2 System Architectures

Modern system architectures try to exploit parallelism in one way or another. Good examples are instruction-level parallelism (ILP) in super-scalar and very-long-instruction-word (VLIW) processors and data-level parallelism (DLP) and task-level parallelism (TLP) in (multi-)processor architectures. ILP generally concerns general purpose CPUs, DLP is most common used for both single processor and multi-processor architectures,

and TLP for multi-processor architectures. In the multi-processor approach a system is constructed out of a number of coarse-grain components working together to get the job done. This leads to both heterogeneous [35] and homogeneous [34] multi-processor systems.
The previous section shows that in order to meet the high compute power and low energy consumption constraints of next generation embedded multi-media systems, multi-processors systems must be used. These systems are in nature concurrent. This concurrency must be exploited. This requires that task- and data-level parallelism is used as these allow to execute different parts of a computation concurrently on different processors.

2.3 Application Specification

Next-generation embedded multi-media systems will often realize so-called streaming applications in which data transformations play a dominant role (such as video processing). Programming techniques are required to model the functional behavior and desired timing- and energy-related properties of these applications. The previous section shows that these applications will be executed on multi-processor systems. To use these systems, they must be able to exploit the task- and data-level parallelism in the application. This requires that the parallelism in the application is specified, so that it can be used in the multi-processor system.
To model the functional behavior and desired timing- and energy-related properties of these applications, novel programming techniques are required. These programming techniques should provide insight in concurrency-, timing-, and energy-related properties of a system at the specification level, without the need to fully implement an application. An important aspect of these programming techniques is that they must be able to specify the concurrency in an application and allow formal reasoning about it. The next chapter presents a programming technique, the computational-network model, that can be used to explicitly specify the concurrency in an application at the specification level. This model is based on the existing Kahn process network model [13, 14]. The computational-network model is extended with a concurrency model, which makes it possible to reason formally about the task- and data-level parallelism in an application.

2.4 Multi-Processor Mapping

Section 2.2 discussed the trends in modern systems architectures. These systems will be multi-processors systems that contain concurrency. Section 2.3 discussed that in order to exploit this concurrency, it must be extracted from the application and made explicit in the specification. One of the important issues facing modern multi-processor designers is now the development of effective techniques for the distribution of the specified, concurrent program over multiple processors. These techniques must optimize execution time, memory usage, and energy consumption.
Research on mapping abstract models of computation (e.g. Kahn process networks) onto

multi-processor systems is still in a very early state [8, 40]. A good example of this is a JPEG case study performed at Philips Research [18]. The article describes a JPEG decoder that is mapped onto a multi-processor system. The techniques used in this case study require that the target architecture is known before the mapping is started. A simulation model of the system architecture is needed in the complete mapping trajectory to verify that the mapping is optimal. This makes that the used techniques are only valid for a given architecture. It also requires extensive simulation, which make the design exploration expensive. To overcome these problems, we think that a global framework needed to do this mapping looks like the mapping trajectory shown in Figure 2.1.

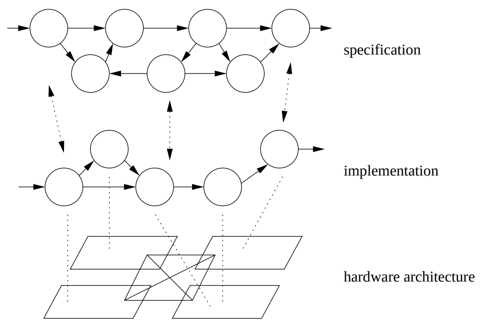


Figure 2.1: Mapping

The mapping trajectory starts with a computational network that specifies the behavior of the application. This computational network is mapped onto an implementation. The implementation is also a computational network. The difference between the specification and the implementation computational network is that the former is to a large extent target-architecture independent, whereas the latter takes all the relevant aspects of a given architecture into account, and is optimized toward desirable properties concerning execution time, power dissipation, etc. The final step is a mapping from the implementation to the hardware architecture. It concerns the efficient execution of sequential code on various sorts of processing elements and is covered by compiler technology.

2.5 Motivating Example

In the introduction of this chapter it is stated that future embedded multi-media systems use multi-processor architectures. The trends in these architectures and the required programming and mapping techniques for them are discussed in the previous sections. One of the important messages of these sections is that concurrency must be made explicit in the specification of an application. The granularity of the concurrency in the specification must be adapted to the concurrency available in the system architecture to provide a good mapping. To be able to do this, a model is needed that allows formal reasoning about the concurrency in a specification. This section presents an example that illustrates this need for a formal concurrency model. This example serves as our prime motivation to perform the research described in this thesis.

Example 2.2 Figure 2.2 shows a model for the computation performed by an application. The computation is divided in 3 tasks. These tasks are performed after each other, as indicated by the arrows connecting the tasks. There is an arrow connected to task 1 that is not connected to another task. This models that task 1 gets the data needed to perform its computation, task, from the environment. The arrow leaving task 3 models the output of this task 3. The output produced by task 3 is the output of the computation performed by the application.

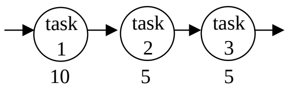


Figure 2.2: A computation split in three tasks.

The numbers below the tasks in Figure 2.2 represent the work (e.g. processor time used) of a task when it is executed on a processor for a computation. Section 2.4 stated that at the implementation level there is a specification that can be mapped on a multi-processor system by only considering the efficient execution of sequential code on a processor. This requires that the tasks at the implementation level of the mapping trajectory have a one-to-one mapping with the processors in the system architecture.
The goal of a multi-processor system will often be to keep all processors busy. As one task is mapped onto one processor, it requires that the total computation is divided evenly over the tasks. In other words, all tasks have to do the same amount of work. The tasks in Figure 2.2 do not meet with this requirement as the first task does the same amount of work as the tasks 2 and 3 together. Figure 2.3 shows two solutions to this problem. Solution I combines the tasks 2 and 3. The two resulting tasks perform both 50% of the work. This solution can be mapped onto 2 processors. The approach used in solution II is different. It does not combine two tasks, but splits task 1 in two new tasks. These tasks 1a and 1b perform both half of the work performed by the original task 1. The tasks in this solution perform all 25% of the total work needed for the computation and would require a multi-processor system with 4 processors.

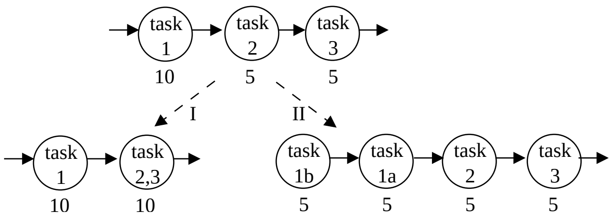


Figure 2.3: Two solutions with a balanced workload for the computation.

An important question is now which solution, I or II, is better. This will first depend on the systems architecture. If this architecture does not contain 4 processors, then only solution I can be chosen. However if the number of processors in the system architecture is not fixed yet, we can make a trade-off. Solution II will require more processors, more area, than solution I. On the other-side, solution II contains 4 tasks. If these four tasks run on four different processors, then it is possible to let each task execute on a different computation, set of input data. Solution II can then perform its computation on 4 different sets of input data, while solution I can perform in the same time the same computation only on 2 different sets of input data. Solution II has thus a higher throughput. We can thus say that solution II is more concurrent than solution I. ■

This example shows different that solutions can have different concurrency properties. To compare different solutions, a model of concurrency is needed that allows a system designer to analyze the concurrency in an application in a formal way. This concurrency model must help a system designer in answering questions like: How concurrent is my application? Which of these solutions contains the most concurrency in its specification?

2.6 Conclusion

This chapter shows that concurrency will play an important role in next-generation embedded multi-media systems. These multi-media systems will be multi-processor systems, as that is the only way in which the high-performance and low-power constraints of next-generation compute platforms can be met.
Multi-processor systems are in nature concurrent, and this concurrency in the architecture should be exploited. This requires that the concurrency is used in the mapping trajectory from the system specification to the hardware architecture. The concurrency in an application should therefore be extracted, and made explicit, in the models that are used to specify a system. To support the extraction of concurrency from an application, the model must contain a concurrency model. This concurrency model should support formal reasoning about concurrency. In this way, concurrency can be extracted from an application in a uniform way. To support the designer in the concurrency extraction, the concurrency model must indicate to what extent the concurrency is extracted from an application and it should suggest how the specification of a system must be modified to better exploit the concurrency available in the application.
The next chapter of this thesis presents a model of computation that can be used to specify a system. Concurrency is made explicit in this computational model. Chapter 4 describes a concurrency model that can be used with this computational model and has the above mentioned properties. The rest of this thesis discusses the implementation of the computational model and the concurrency model. It further shows the use of these models in a case study that implements a JPEG decoder.

Chapter 3

Model of Computation

3.1 Introduction

A number of different formal models of computation are used to model distributed computations. They express concurrency using formal languages [3], partial orders [30] or automata [7] at different levels of abstraction. Prime examples are Dataflow graphs [23], Kahn process networks [13, 14], Petri nets [29], Statecharts [10] and Process Algebra [3]. Section 3.2 introduces a computational model at a higher level of abstraction than these models. The relation with the previously mentioned models is also discussed in this section. Concurrency is made explicit in this model of computation, called the computational-networks model. Section 3.3, extends the computational-networks model with a notion of partially ordered events in time.

3.2 Computational Networks

Informally speaking, a parallel computation is organized in the following way: some autonomous computing nodes are connected to each other in a network by point-to-point connections. Computing nodes exchange information through these connections. These connections are the only way by which the computing nodes may communicate. A given node computes on data coming along its input connections to produce output on some or all of its output connections. The computing nodes are at any given time computing or waiting for information on one of their input connections.
Such a parallel computation can be represented as a directed graph. The nodes of this graph represent the computing nodes. The edges represent the communication connections between the computing nodes; an incoming edge represents an input connection, an outgoing edge represents an output connection.
This informal definition of a parallel computation is used to construct the computational-network model. A computing node is modeled in the computational-network model as a *compute node.* This compute node is a "component" that has a set of input ports and a set of output ports. The input ports hold the input data. This input data is modeled using strings of data-elements. The execution of the compute node can be imagined

by the compute node "reading" or "observing" these input strings and generating the appropriate output strings. This is done following the *transformation* that describes the behavior of the compute node. The output ports will hold the strings after applying the transformation to the input strings. This defines a compute node in an informal way; Definition 3.1 defines a compute node in a computational network in a more formal way.

Definition 3.1 (Compute node) A compute node is a tuple (I, O, t) where

i) I is a set of input ports;

ii) O is a set of output ports;

iii) t is a transformation. A transformation describes how a compute node computes a (tuple of) strings on its output ports using a (tuple of) strings on its input ports.

A compute node computes a (tuple of) strings on its output ports using a (tuple of) strings on its input ports. These strings are produced and consumed by other compute nodes in the parallel computation or by the environment of the computation. The "communication" of these strings between the different compute nodes is done in the computational-network model using *connections*. Data-streams are transferred in-order over connections.

Definition 3.2 (Connection) A connection is a pair (p, q). The connection transfers data-streams in-order (fifo) from port p to port q.

The definition of a compute node and a connection enables us to construct a *network component*. This network component realizes a parallel computation. It contains a set of compute nodes that are connected to each other using connections. The compute nodes in the network component communicate with each other using the connections.
A computation is only useful if the network component will eventually produce a result on some (set of) output port(s). These results are produced on the output port(s) of the compute node(s) in the network component. These output ports must not be connected to the input ports of other compute nodes. The unconnected output ports of the compute nodes serve as the output ports of the network component. A compute node might then communicate its results not only to other compute nodes in the network component, but also to the environment. The parallel computation might also require the use of input from the environment of the network component. The network component should therefore contain a (set of) input port(s). An unconnected input port of a compute node serves then as an input port of the network component. Definition 3.3 defines this network component in a formal way.

Definition 3.3 (Network component) A network component NC is a tuple (N, C, I, O) where

i) N is a set of compute nodes;

ii) C is a set of connections;

iii) Every connection in C connects an output port of a compute node to an input port of a compute node;

iv) Every port of every compute node is connected to at most one connection;

v) I is the set of input ports of the network component, being defined as those input ports of the compute nodes in N not connected to a connection in C;

vi) O is the set of output ports of the network component, being the unconnected output ports of the compute nodes in N.

An example of a network component is shown in Figure 3.1. The network component shown here contains three compute nodes a, b and c. They are connected to each other via three connections labeled c_1, c_2 and c_3. The input port of compute node a is unconnected and thus an input port of the network component. This input port is represented by a diamond. An output port of a network component is represented by a small box. Compute node c provides the only output port of this network component.

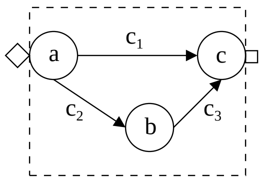


Figure 3.1: An example of a network component

It is now possible to construct a model for a parallel computation using a single network component. This network component does not contain any hierarchy. Hierarchy might however be useful in a larger computational network, as it allows abstraction from primitive operations taking place in the system. The computation in a network component could be seen as a primitive operation. A parallel computation may consist of the computations taking place in a number of these network components. The network component itself is for the rest of the system a black box with a set of input ports, a set of output ports and a defined behavior i.e., to the outside world it looks like a compute node. Definition 3.4 defines a *computational network*, which realizes this hierarchy. The definition follows the same lines as Definition 3.3.

Definition 3.4 (Computational network) A computational network is defined as a tuple (NC, C, I, O) where

i) NC is a set of network components;

ii) C is a set of connections;

iii) Every connection in C connects an output port of a network component to an input port of a network component;

iv) Every port of every network component is connected to at most one connection;

v) I is the set of input ports of the computational network, being defined as those input ports of the network components in NC not connected to a connection in C;

vi) O is the set of output ports of the computational network, being the unconnected output ports of the network components in NC.

A computational network contains a set of network components that perform computations. The computational network contains further a set of input ports and a set of output ports. These are used to communicate with the outside world. The network components are connected to each other by connections in the computational network. The unconnected input ports and output ports of the network components serve as the input ports and output ports of the computational network. An example of a computational network containing two network components A and B is shown in Figure 3.2. The two network components are connected to each other with the connection c_1. This connection connects the input port of compute node v in network component B to the output port of compute node u in network component A.

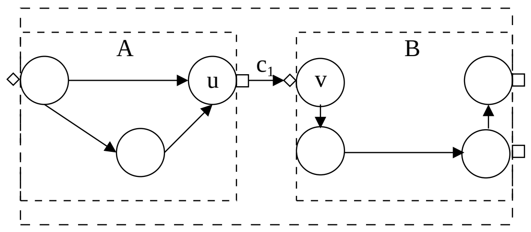


Figure 3.2: An example of a computational network

The combination of Definitions 3.3 and 3.4 gives a modular model of computation that allows two levels of abstraction. It is straightforward to generalize this to a truly hierarchical model by assuming that the compute nodes in a network component can be seen as network components themselves. However, this is not done because it would unnecessarily complicate the remainder.

The concept of the computational-network model introduced in this section is as follows: compute nodes read a (set of) strings from their input ports. These strings are transformed according to the transformation defined by the compute nodes. The result of this transformation is again a number of strings that are written to the output ports of the compute nodes. The compute nodes communicate with each other using connections.
Section 3.1 lists a number of formal models of computations that are used to model distributed computations. We now discuss how some of these computational models can be modeled in the computational-networks framework. The first model of computation that was mentioned is the Kahn process networks (KPNs). KPNs are easily modeled in our framework. The functions in KPNs become our compute nodes. The data streams that are used in KPNs to communicate between the functions are realized by sending these streams over the connections in the computational-network model. The computational-network model can also model Dataflow graphs (DFGs). The actors in a DFG become our

compute nodes and the communication is done using data streams. The computational-network model can further model a subclass of Petri nets, the marked graphs. The transitions in these marked graphs become our compute nodes and the places become the connections.

3.3 Executions

A computational network, which was introduced in the previous section, consists of a set of network components. These network components consist on their turn of a set of compute nodes n_1, n_2, ... , n_M which together perform a computation. Each compute node performs therefore a sequence of actions which are modeled as a totally ordered sequence of events. A compute node can *write* on its output ports, as a (set of) strings, the result of the transformation performed to the (set of) strings it *reads* from its input ports.
The events that can occur in a computational network during a computation are classified into the following three types:

1. **write event** Such an event models a write operation in which a compute node writes on one of its output ports;

2. **read event** Such an event models a read operation in which a compute node reads from one of its input ports;

3. **internal event** Such an event models the execution of an action or a sequence of actions; these actions must not include read or write operations.

The execution of a compute node is a totally ordered sequence of internal, read and write events that take place during the execution of the compute node. The read and write events impose further a partial order on the events taking place in the computational network as a whole. The relation between the different events is called the *causality relation* or *happened before relation* and denoted by $\prec$. The causality relation is defined according to [22].

Definition 3.5 (Causality relation) Let E be a set of events produced by the execution of a computational network. For e, $e' \in E$, $e \prec e'$ holds if and only if

i) e and e' are events in the same compute node and e precedes e',

ii) e is a write event and e' is the corresponding read event, or

iii) there exists an e'' such that $e \prec e''$ and $e'' \prec e'$.

Two events e and e' are said to be causally related if and only if $e \prec e'$ or $e' \prec e$ holds. If neither $e \prec e'$ nor $e' \prec e$ holds, these events are concurrent. Let $e_1, e_2, \ldots, e_k$ be a sequence of events such that $e_i \prec e_{i+1}$ for $1 \leq i \leq k-1$. Such a sequence is called a *causal path* from event e_1 to event e_k.

In an implementation it makes sense to associate different delays with the different events that can take place in the system. The communication that takes place in the system will also have a delay associated with it. The main idea behind the computational network is that it can model a distributed computation and that it allows reasoning about different aspects of the system. A number of these aspects are influenced by the timing properties of the events that take place in the system and the causal path of the events in the computation. To allow reasoning about causality and some timing aspects on a relative high level of abstraction without referring to implementations/physical time, we use logical clocks. To make it more practical, we introduce delays for the events that take place in compute nodes and delays for the connections.

Lamport's logical clocks [22] can be used to create an ordering that is consistent with causality (Definition 3.5) for all events that occur during a computation in the computational network. Lamport's system of logical clocks assumes a set of logical clocks, one per compute node. The logical clock's assign to every event a time-stamp that contains the value of the logical clock at the moment the event occurred. Every event is performed within a single logical clock value; there is no delay associated with this event. The clock of a node is incremented once between two events. Furthermore, communication imposes a causality relation that must be respected by the logical clock, which means that the clock of a reading node is updated based on the time-stamp of the read event. There is no delay associated with the communication itself.
To reason about timing aspects without referring to implementations, we associate a delay with the events that take place in the compute nodes and the communication over the connections. To model this delay, we extend the Lamport's logical clocks with a function d, called the *delay function*. This function associates a delay with every event that occurs during a computation in the computational network.
Formally, the delay function maps a set of events E plus the set of connections C to the set of natural numbers, denoted $\mathbb{N}$. Formally, $d : E \cup C \rightarrow \mathbb{N}$. The Lamport's logical clocks can be modeled by assigning a delay of one to every event and a delay of zero to every connection.
The time-stamping mechanism based on Lamport's logical clocks, in the remainder simply referred to as the time-stamping mechanism, contains a function t. It is called the *global logical clock* and maps a set of events E to a totally ordered set $\mathbb{N}$ with ordering $<$. Formally, $t : E \rightarrow \mathbb{N}$ such that $e \prec e' \Rightarrow t(e) < t(e')$. The global logical clock can be computed via a set of counters, the local logical clocks. Each compute node in the computational network maintains a different counter. Let t_i denote the counter maintained by compute node n_i.
When a compute node n_i executes an event, it updates first its local clock t_i and then assigns a time-stamp t to the event. This time-stamp is the value of the local logical clock after the event is executed.
The protocol used to update the clock t_i of a compute node n_i is the following:

1. When n_i executes an internal event or a write event e, the clock value t_i is advanced by setting $t_i := t_i + d(e)$.

2. When n_i executes a read event e, where y is the time-stamp of the corresponding write event and c_j is the connection over which the event was received. The clock is advanced by setting $t_i := \max(t_i, t + d(c_j)) + d(e)$.

The local logical clock change depends in case 1 of the protocol only on the delay associated with the event executed. The change of the local logical clock in case 2 of the protocol is more complex. It is of course determined by the delay associated with a read event, but also by the causality relation imposed by the corresponding write event and the delay associated with the connection. If the logical clock of the compute node is already larger than the logical clock value of the compute node in which the write event occurred at the time of this write event plus the delay associated with the channel, then the logical clock is advanced by the delay associated with the read event. Otherwise, the logical clock of the compute node is changed to the value of the logical clock of the compute node at the time the corresponding write event occurred. This logical clock value is then raised by the delay associated with the connection and the read event.

The time-stamping mechanism based on Lamport's logical clocks provides a method for deriving an ordering of all events that take place during a computation in the computational network. This ordering is consistent with causality and the delay associated with the events that take place in the computation. The time-stamping mechanism can be used to analyze the ordering and abstract timing of events that take place in a computation. The remaining part of this section introduces a number of measures, based on the time-stamping mechanism. The measures represent a number of timing related properties of a computation in a computational network.
First, we define the *execution time* (Definition 3.6) of a compute node and the *total execution time* (Definition 3.7) of a computational network. The execution time of a compute node represents the amount of logical time that a compute node is executing for a computation. The duration of the total computation in the computational network is expressed in the *total execution time.*

Definition 3.6 (Execution time) The execution time, T_e^n, of a compute node n in which the set of events E_n occurs, is defined by Equation 3.1.

$$T_e^n = \sum_{e \in E_n} d(e) \tag{3.1}$$

Definition 3.7 (Total execution time) The total execution time, T_E, is defined as the time needed for an execution. In other words, the total execution time is equal to the largest value of the local logical clocks of all compute nodes at the end of the execution.

The execution time of a compute node indicates how long, expressed in logical time, the compute node was running. The total execution time measures how much time was needed to execute the complete computation on the computational network. A compute node need not be executing during the total execution time. The compute node may be idle because it has finished its execution or it is waiting for data on one of its input ports. The *idle time* of a compute node is defined as follows:

Definition 3.8 (Idle time) The idle-time, T_i^n, of a compute node n is defined by Equation 3.2.

$$T_i^n = T_E - T_e^n \tag{3.2}$$

An executing compute node performs read, write and internal events. The read and write events are needed to communicate with other compute nodes. The internal events are needed to perform the transformation of the compute node. This transformation can be seen as the computation performed by the compute node on the data read on its input ports. The logical time needed to perform this transformation is expressed in the *computation time* (see Definition 3.9).

Definition 3.9 (Computation time) The computation time, T_c^n, of a compute node n in which the set of internal events I_n occurs, is defined by Equation 3.3.

$$T_c^n = \sum_{e \in I_n} d(e) \tag{3.3}$$

In the remainder of this thesis we use the word *computation* exclusively for the transformations performed on the strings of data in the network. This does not include the communication of the strings of data. When we refer to the combination of computation and communication we use the word *execution.* The computation performed by a compute node refers thus to the internal events that have occurred in the node to perform the transformation. The execution of a compute node refers to the combination of internal, read and write events that occur in a node.

A computation can be displayed graphically in an *event diagram*, such as the one shown in Figure 3.3. There are four compute nodes a, b, c and d in this figure. The clock ticks at which a read, write or internal event is occurring are indicated by black circles with respectively an arrow leaving it, entering it or with no arrow connected to it. An event that takes more clocks ticks is indicated by a black ellipse over all clock ticks at which the event occurred. An arrow indicating a read or write event is connected to the first clock tick at which the event occurred. In Figure 3.3 we have Lamport's logical clocks, i.e., $d(e) = 1$ for event e and $d(c) = 0$ for connection c. At logical time 1, there occurs a read event in compute node a, this is a read event in which the compute node reads data from the environment – outside world. This is represented by a black circle with an arrow entering it, just as in a normal read operation. But the other side of the arrow is not connected to another event in the event diagram, this indicates that it is a read from the environment. A write to the environment is represented by an arrow leaving a black circle but not entering another black circle. Such a write event can be seen at logical time 12 in compute node d. The total execution time of this computational network is equal to 12, as at that logical clock value the last event occurs. The compute node a has an execution time of 4, an idle-time of 8 and a computation time of 2.

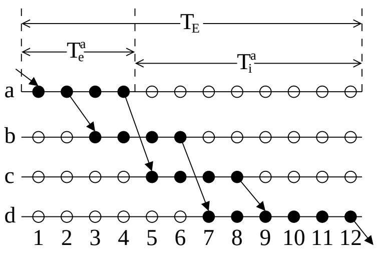


Figure 3.3: Example of an event diagram

As explained, the idle-time T_i^n represents the logical clock values at which no event is being executed in a compute node n_i. The node is said to be *idle* at these clock values. The clock value at which this occurs is called an *idle time* of the node. Idle times are represented in Figure 3.3 by unfilled circles.

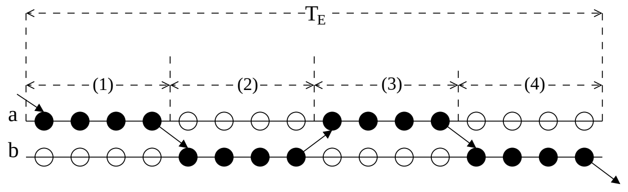


Figure 3.4: Idle events in a compute nodes

Figure 3.4 shows the event-time diagram for a computation in a computational network that consists of two compute nodes a and b. It shows all the computation needed for a single input from the environment. Compute node a performs events at the logical clock values 1 through 4 (period (1)) and from 9 through 12 (3). The compute node is idle from the logical clock values 5 through 8 (2) and from 13 through 16 (4). The reason for the compute node being idle in period (2) and (4) is different. In (2), the compute node is waiting for data from compute node b. In (4), the compute node has finished its execution, while the computation has not yet finished. Another execution of compute node a for a new input could already start during period (4). This is a desired property as we are aiming at streaming applications in which the same computation must be repeated many times on different inputs. Starting another computation is not possible during period (2), as the execution has not yet finished. The idle time of period (2) is unavoidable in the computation – the compute node cannot execute for this computation or another computation –, while the idle time of period (4) is not. The question is now what causes the idle time and whether or not the idle time is unavoidable in the computation. To answer the first question, we have to see why an idle time can be present:

1. The execution of a compute node n is not started yet, but the computation is;
2. The execution of a compute node n is finished, but the computation is not;
3. The execution of a read event of a compute node n is delayed because the required data from another compute node is not yet available.

The idle time caused by case 3 is important for the behavior of the computational network and should be removed to get a more concurrent solution. It may be possible to remove this idle time by changing the order of events in the causal chain(s) leading through the read event causing the problem. The idle time caused by cases 1 and 2 are not important for the computational network, as those idle times will be used for the execution of the compute node for another input. To make a distinction between the idle time that is important for the computational network and the idle time that is not important, we introduce the *communication idle time* (Definition 3.10). This communication idle time is equal to the idle time caused by case 3.

Definition 3.10 (Communication idle time) The communication idle time, T_{ci}^n, of a compute node n is defined as the number of idle times of n after the first read or write event occurred and before the last read event occurred.

The time that a computation occupies a compute node, the node cannot execute for another input, is called the *run-time*. This time includes the execution time (see Definition 3.6)and communication idle time (see Definition 3.10) of a compute node, as those two times together determine the number of logical clock values at which the compute node is occupied by a computation.

Definition 3.11 (Run-time) The run-time, T_r^n, of a compute node n is given by Equation 3.4.

$$T_r^n = T_e^n + T_{ci}^n \tag{3.4}$$

The last measure that is defined is the *sequential execution time.*

Definition 3.12 (Sequential execution time) The sequential time of an execution, T_{SE}, is the time needed to run the computation sequentially. This time is defined to be equal to the sum of the execution times of all compute nodes in the computational network:

$$T_{SE} = \sum_n T_e^n \tag{3.5}$$

The sequential execution time is found by adding the execution times of all compute nodes in the computational network. This approximates the execution time of a sequential version of the computation. This approximation is in general not exact, as a sequential execution might have different delays associated with the read and write events, as these transform into local variable operations. There will also be control statements needed in the sequential execution to perform the correct computation. This will introduce some extra delay. It is however assumed that the sum of the execution times will be a reasonable measure of the execution time of a sequential version of the computation.

3.4 Conclusion

This chapter has presented a model of computation that can be used to model distributed computations. This model is comprised of two parts. The first part, the computational-network model, can be used to specify a concurrent system. The second part models executions and can be used to analyze the events that take place during a single execution.

The computational-network model operates at a high level of abstraction. This allows the modeling of all kinds of different properties of systems in a very natural way. The computational-network model introduces a natural way to explicitly specify concurrency in a system by introducing different compute nodes in a computational network. These compute nodes communicate with each other by means of connections. The computational networks model allows for the modeling of different models of computation in its framework.

Section 3.3 has introduced a classification for the different events that can occur in a distributed computation and introduces the causality relation among these events. These events can be ordered using a time-stamping mechanism based on Lamport's logical clocks. This time-stamping mechanism gives an ordering of the events taking place in a distributed computation and introduces an abstract notion of time into the computational-network model. The notion of time is extended with a number of different time measures that express the activity of a compute node and the computation.

The remainder of this thesis studies a formal concurrency model that is based on the computational-network model and the notion of time which have been introduced in this chapter.

Chapter 4

Concurrency Model

4.1 Introduction

The previous chapter introduced the computational-network model. In this model, autonomous computing nodes are modeled as compute nodes, which communicate with each other using point-to-point connections. The compute nodes perform a concurrent computation on the data. Concurrency in the computation is made explicit in this way. This chapter describes a formal concurrency model that can be used to analyze the concurrency in a computational network. Section 4.2 discusses the basic assumptions that are made in the concurrency model about the computational network. Section 4.3 presents the concurrency model that is comprised of five main measures supported by a set of detailed measures. A set of computational networks analyzed using the concurrency model is presented in Section 4.4. The relation with other concurrency models is discussed in Section 4.5 Finally, Section 4.6 contains some concluding remarks.

4.2 Assumptions

Concurrency in a computational network is influenced by many things. It is for instance influenced by the way the computation is divided over the different compute nodes in the computational network. If all compute nodes have to do more or less the same amount of work for a computation, then it might be possible to do much of the computation concurrently. If not, then a small number of the compute nodes has to do most of the job, while the other compute nodes are idle (not executing). It will be clear that in the latter case there is less concurrent activity possible in the computational network. This concurrency property is one of those that we are interested in, because changes in the computational network may influence it. Therefore, it will be taken into account in the concurrency model that is presented in the next section.
The communication structure may also influence the concurrency in the computational network. This can happen in a number of ways. First, the length of the string of data that can be present on a connection at one moment in time may influence it. Assuming a connection can hold only a string of data of finite length, then a write event may be

blocked because a connection is full. This will influence the concurrency, as the compute node cannot continue its computation until there is enough space on the connection to complete the write event. This effect is caused by the amount of memory allocated for the connection [2] and not by the computational network itself. This effect from the environment should not be taken into account in the concurrency model. It can be left out with the following assumption.

Assumption 4.1 The strings of data on the connections of the computational network may be of infinite length.

A compute node can also be blocked if it cannot read the data from its input ports. If these input ports are connected to the environment, then it implies that the compute node must wait until the environment has written the data to the connection. As we do not want to study the effects of the environment on the concurrent behavior of the computational network, this should not be possible. This is realized using the following assumption.

Assumption 4.2 The strings of data on the input ports of the computational network are of infinite length. These strings of data are infinite concatenations of the strings of data required for one computation and are present on the connection before the connected compute node wants to read them.

This assumption implies that the computational network performs infinitely many times the same computation on the same data set. In this way, it is possible to analyze the behavior of the computational network as if it is running on an infinitely large set of inputs. These inputs are all the same. In practice, this will not be true. I.e., a JPEG decoder will decode a set of different images. The images need not be the same. However, their data streams will be comparable. It is the idea that the input used in our assumption approximates the average input of the computational network.
The concurrency is also influenced by the mapping of compute nodes on processors. The nodes of a network will be executed on a set of processors. A (run-time) scheduler will assign the nodes to processors on which they execute. If there are less processors than compute nodes, the scheduler will have to choose which compute node to execute when. This effect is caused by the environment – the (run-time) scheduler – and not by the computational network itself. We are not interested in effects on the concurrency that are caused by the environment. They must therefore be left out of the concurrency analysis. This is done using the following assumption.

Assumption 4.3 There are infinitely many processing nodes available.

This assumption implies that a compute node that wants to execute can do this as soon as the required data becomes available. Only the absence of required data on its input ports can stop a compute node from executing. Effects of scheduling a compute node on a processor are in this way left out of the model.
The final assumption that is made is the following.

Assumption 4.4 The computation time of a single computation is finite.

This assumption implies that the strings of data for one computation, which are present at the input ports of the computational network, are of finite length. This makes it possible to analyze, in finite time, the behavior of the computational network.

4.3 Concurrency Measures

A computational network that realizes a computation has certain concurrency properties. The goal of the concurrency model is to provide measures for these properties. To be able to derive these measures, we must first identify which properties must be considered. To do that, we consider what a computational network does. In short: a computational network performs a transformation on the data streams communicated over the connections. The compute nodes in the computational network are busy with the computation – applying transformation – and communication – reading and writing to the connections. Considering concurrency, we want that all compute node have to do an equal share of the computation. This balanced workload provides the option to have a low scheduling overhead. The compute nodes must also perform the computation as fast as possible – each compute node individually and the computational network as a whole. Finally, it is required that not too much of the time is spent on communication. To get numerical values for all of these properties, we introduce five different concurrency measures in the following sections. Each section treats one measure. It explains which concurrency property the measure captures, why it is needed and how the measure must be interpreted. An important goal for the concurrency measures is that the values are in the range [0,1], in which a value 1 means that the measured concurrency property is optimal and a value close to 0 means that it is very bad. This makes it simple to compare different computational networks.
In the remainder it is assumed that $CN = (NC, C, I, O)$ is a computational network with NC a set of network components and C a set of connections. $nc = (N, C, I, O)$ denotes a network component in NC and n is a compute node in the network component nc.

4.3.1 Computation Load

A computational network consists of a set of compute nodes that communicate with each other. Each compute node performs a transformation on the data it reads from its input ports. The result of this transformation is written to the output ports of the compute node. The transformation can be seen as the computation performed on the data by the compute node. The compute node will need a certain amount of time to perform this computation. The compute node will also need a certain amount of time to communicate the data with other compute nodes.
The goal of a computational network is that the strings of data that are on the input ports of the computational network are transformed into strings of data on the output ports of the computational network. This transformation should be done efficiently. This means that all of the compute nodes working must spend as much effort as possible on

computing the result. This implies that every compute node should spend a large amount of its time on the computation and not on communication.
One could argue that introducing more compute nodes may result in the possibility to do more of the transformation in parallel. This would then lead to a faster transformation of the strings of data. However, communication takes time and this time needed for communication may outweigh the time gained by splitting compute nodes.
We can conclude from the observations presented above, that the ratio between the time spent on computation and the time spent on communication is important when considering concurrency. This observation leads to the first measure in the concurrency model, the *computation load.*

The computation load has to express the ratio between the time spent on the transformation and the time spent on the transformation and communication together. This ratio can easily be computed for a single compute node. The time spent on the transformation is then expressed by the computation time of the compute node (see Definition 3.9). The time needed for both the transformation and communication is expressed by the execution time (see Definition 3.6). The computation load of a compute node n is then given by Equation 4.1.

$$computation\,load_n = \frac{T_c^n}{T_e^n} \tag{4.1}$$

The value of the computation load is in the range [0,1]. A value of 1 means that the compute node is all of its time busy with the transformation, while it does not spent time on the communication. A value of 0 means on the other hand that all of the time is spent on communication.

To get a value for the computation load of a network component, we take the average computation load of the compute nodes in the network component nc (See Equation 4.2). In this way, we get a notion of the time spent on the transformations in the compute nodes compared to the time they spent on transforming and communicating the strings of data.

$$computation\,load_{nc} = \frac{\sum\limits_{n \in N} computation\,load_n}{|compute\ nodes\ in\ nc|} \tag{4.2}$$

Equation 4.2 associates a computation load with a network component as defined in Definition 3.3. A computational network consists of a set of network components. To get a computation load for the computational network, we could simply take the average computation load of the network components. The problem with this solution is that the average is taken over the network components without considering the number of compute nodes in a network component. The number of compute nodes in a network component does not influence the measure. This is not a desired property. Consider for instance the situation in which a computational network contains two network components. One has a computation load of almost 1 and contains 10 compute nodes. The other network

component contains only 1 compute node and has a computation load of almost 0. If the computation load of the computational network is calculated by taking the average over the computation loads of the network components, then it would have a value of around 0.5. We might conclude from this that the compute nodes spend on average as much time on the transformation, as they spend on the communication. This is obviously not true, as most of the compute nodes spend almost all of there time on the transformation. In this way, it would be possible to regroup the compute nodes over the network components and obtain different numbers. This does not make sense, as the time spent on the communication and transformation of each compute node and the computational network as a whole does not change.
It will be clear that the number of compute nodes in a network component must be taken into account, when computing the computation load of a computational network. Therefore, we associate with each network component a *size*. The size of a network component is equal to the number of compute nodes in the network component (See Equation 4.3).
The computation load of the computational network is given by Equation 4.4. This measure weights the computation load of the network components by their size.

$$size_{nc} = |compute\ nodes\ in\ nc| \tag{4.3}$$

$$computation\ load = \frac{\sum\limits_{nc \in NC} computation\ load_{nc} \cdot size_{nc}}{\sum\limits_{nc \in NC} size_{nc}} \tag{4.4}$$

The main concurrency measure in the concurrency model is the computation load measure of a computational network (Equation 4.4). The computation load measures for the network component (Equation 4.2) and compute node (Equation 4.1) are so called detailed concurrency measures. These detailed measures provides an insight in the concurrency properties of the different network components and compute nodes.

Example 4.1 Given the computational network of Figure 4.1. This computational network consists of one network component. The network component contains four compute nodes a, b, c and d.

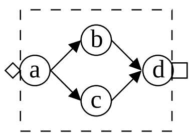


Figure 4.1: Computational network

The computation performed by this computational network starts with compute node a that reads data from its input port. It performs a transformation on this data. The result is written to its output ports, that are connected to the input ports of the compute nodes

b and *c*. These compute nodes read the data output by compute node *a* from their input ports, apply a transformation to it and write the result to their output ports. Compute node *d* will then read these strings of data and perform also a transformation on it. The results of this transformation are output by compute node *d* and the computational network, as the output port of compute node *d* is the output port of the computational network.
The event diagram for this computation is shown in Figure 4.2.

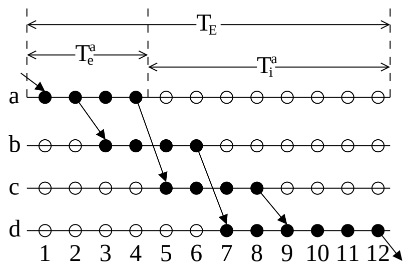


Figure 4.2: Event diagram

The event diagram shows that all compute nodes have idle time. For instance, compute node *a* has an idle-time of 8 logical clock values. This idle-time does not belong to the communication idle time (see Definition 3.10). As all idle-time for *a* occurs after the last write event. The idle time of all other compute nodes is also no communication idle time. These compute nodes idle before the first read event and after the last write event. The computation load of the compute nodes is found using Equation 4.1 and is listed in Table 4.1.

compute node	computation load
a	0.25
b	0.5
c	0.5
d	0.5

Table 4.1: Computation load of compute nodes.

The computation load of the computational network is now found using Equation 4.4. The result is listed in Table 4.2.

measure	value
computation load	0.44

Table 4.2: Computation load of computational network.

The value of the computation load indicates that the compute nodes in the computational network spent almost the same amount of time on performing the transformation as they spent on communication. ■

4.3.2 Execution Load

The computation load measure came from the observation that there must be a balance between the time spent on the transformation in the compute nodes and the time spent on communication between the compute nodes. The measure compares the number of internal events to the total number of internal, read and write events that occur during the execution of a transformation in a compute node. The measure does not consider that a compute node may also be idle as it is waiting for data on its input ports. This idle time has been called the communication idle-time (see Definition 3.10) in the previous chapter. It will be clear that in the optimal situation there is no communication idle-time. This might however not always be possible, but the communication idle-time should always be as low as possible. This observation leads to the introduction of the second measure in the concurrency model, the *execution load*.

The execution load must express the ratio between the time that a compute node is busy compared to the time that a compute node must wait. This ratio can easily be computed for a single compute node. The time that a compute node is busy is expressed by the execution time of the compute node (see Definition 3.6). The total time needed by a compute node is given by the run-time (see Definition 3.11). This run-time takes both the execution time and communication idle-time into account. The execution load of a compute node n is given by Equation 4.5.

$$execution\ load_n = \frac{T_e^n}{T_r^n} \tag{4.5}$$

The value of the execution load is in the range (0,1] assuming that at least one event occurs in a node. A value of 1 means that the compute node does not idle during its execution; it does not have to wait for data. A value close to 0 means that the compute node is almost all of its time waiting for data produced by other compute nodes.

To compute the execution load of a network component, we could follow the same approach as with the computation load and take the average execution load of the compute nodes in the network component. We would then ignore the fact that a compute node may have to idle before it can start a new transformation on a new set of data-strings because the compute node is faster than another compute node. To explain this, we must consider Assumption 4.2. This assumption says that the computational network must perform the same computation infinitely many times. Each compute node inside the computational network will perform the same events in the same order for every computation, as the input for every computation is the same. Therefore, the execution-time of every compute node will be the same for every computation. Consider now the situation in which two compute nodes a and b have a run-time of respectively 10 and 20 time-units and their executions depend on each other. If both compute nodes are executed for 10 computations, then a executes for 100 time-units, while b executes for 200 time-units. One could then be tempted to say that a has finished these 10 computations much earlier than b. But if infinitely many computations are run, they will require both infinite

time-units to execute. The time required for a single computation is for compute node a still half of the time required for compute node b. This can be seen as that compute node a has to wait after each computation until compute node b is finished. Compute node b is thus adding extra idle-time to the execution of compute node a. This example shows that if infinitely many computations are executed and a compute node does not have the longest run-time of all compute nodes in the computational network, it will have to idle before it can start a new computation.
This leads to the conclusion that the rate at with subsequent computations can be started is determined by the slowest compute node i.e., compute node with longest run-time. This longest run-time is called the *run-time of the network component* and is given by Equation 4.6.

$$T_r^{nc} = \max_{n \in N} T_r^n \tag{4.6}$$

To compute the execution load of a network component, we must scale the execution loads of the compute nodes inside the network component with their run-time compared to the run-time of the network component. In this way, we take the fact that compute nodes may have to idle before subsequent computation into account. The execution load of a network component nc is given by Equation 4.7.

$$execution\ load_{nc} = \frac{\sum\limits_{n \in N} execution\ load_n \cdot \frac{T_r^n}{T_r^{nc}}}{size_{nc}} \tag{4.7}$$

Using the execution load of a compute node (see Equation 4.5) and the run-time of the network component (see Equation 4.6), we get Equation 4.8 for the execution load of a network component nc.

$$execution\ load_{nc} = \frac{\sum\limits_{n \in N} \frac{T_e^n}{T_r^n} \cdot \frac{T_r^n}{T_r^{nc}}}{size_{nc}} = \frac{\sum\limits_{n \in N} T_e^n}{size_{nc} \cdot \max\limits_{n \in N} T_r^n} \tag{4.8}$$

The execution load measures takes in this way the average of the time that compute nodes are performing internal, read and write events during a computation. It indicates how well the workload (transformation and communication) is balanced over the different compute nodes in the network component. This execution load balance should be as high as possible, as that means that all compute nodes need around the same time to finish their work.

Example 4.2 We continue with the previous example (Example 4.1). The execution load of the compute nodes is found using Equation 4.5. It is listed in Table 4.3.

compute node	execution load
a	1
b	1
c	1
d	1

Table 4.3: Execution load of compute nodes.

All compute nodes have an execution load of 1. This indicates that a compute node does not have to wait for data of other nodes when it can start executing, it has no communication idle time. A compute node may have to idle before it can start a new computation. Because of that, we must scale the execution load of the compute nodes when the execution load of a network component is calculated. This *scaled execution load* is given by Equation 4.9. The scaled execution load of the compute nodes in the network component are shown in Table 4.4.

$$scaled\ execution\ load_n = execution\ load_n \cdot \frac{T_r^n}{T_r^{nc}} \tag{4.9}$$

compute node	scaled execution load
a	0.66
b	0.66
c	0.66
d	1

Table 4.4: Scaled execution load of compute nodes.

Table 4.4 shows that when we repeat the same computation infinitely many times on this computational network, the compute nodes a, b and c will be busy doing a transformation or communicating for 66% of their time, while compute node d is always busy. ■

The computation load of a computational network was found by scaling the computation load of the network components in the computational network, and then taking the average over it. The execution load of a computational network can be computed in the same way (see Equation 4.10). The reasons for doing this are the same as used in the computation of the computation load of a computational network. This execution load measure for the computational measure is one of the main measures in the concurrency model. The execution load measures for the compute node and computational network are used as detailed measures in the concurrency model.

$$execution\ load = \frac{\sum\limits_{nc \in NC} execution\ load_{nc} \cdot size_{nc} \cdot \dfrac{T_r^{nc}}{\max\limits_{nc \in NC} T_r^{nc}}}{\sum\limits_{nc \in NC} size_{nc}} \tag{4.10}$$

Using Equation 4.8, we find Equation 4.11 for the execution load of a computational network.

$$execution\,load = \frac{\sum\limits_{nc \in NC} \sum\limits_{n \in nc} T_e^n}{\max T_r^n \cdot \sum\limits_{nc \in NC} size_{nc}} \tag{4.11}$$

Example 4.3 This example continues with Example 4.2. The execution load of the computational network is now found using Equation 4.10. The results are listed in Table 4.5.

measure	value
execution load	0.75

Table 4.5: Execution load of computational network.

The value of the execution load indicates that the compute nodes in the computational network are 75% of the time active. This is expected as a new computation can be started every 6 logical clock values, making that the four compute nodes require 4 times 6 logical clock values to execute. They spend 18 logical clock values running of these 24 logical clock values. This makes that the are executing 18 of the 24 logical clock values. This value for the execution load means that when the computational network would be realized in a system where all compute nodes can execute as soon as the data becomes available (Assumption 4.3), then 75% of the time there will be 4 compute nodes executing. ■

This section ends with a concluding remark on the name of the measure, execution load. This name might give the impression that the measure is only dependent on the execution of the compute nodes in a network and depends thus solely on the read, write and internal events that take place in the compute nodes. This is not true, the measure depends also on the communication idle time of a compute node. This time is influenced by the communication delay of the connections and "happened before relations" introduced by the computation. These are an integral part of the execution and are therefore taken into account in the measure. One should therefore always remember that as a result not only the internals of the compute node but also the connections between the compute nodes influence the measure. It is interesting to note that this is not the case for the computation load measure.

4.3.3 Restart Interval

The previous section has argued that the compute node with the longest run-time is dominating the computation. This observation has been used to state that the computation must be distributed evenly over the compute nodes. There are two options to improve the execution load. First, the computation could be better distributed over the same number of compute nodes. Second, the computation could be distributed over a different number of compute nodes. An extreme case of this would be a computational network with just one compute node. This computational network would have a computation load and an execution load of one. However, it would be hard to argue that such a solution is always the optimal one from a concurrency point of view. The single node computational network can not start executing the compute node for another input before it has completely finished the computation for the previous input. The rate at which different computations can be started is as low as a sequential version of the computation. On the other hand, a solution with more compute nodes will probably be able to start a new computation sooner. The *restart interval* of that solution will be better, while the execution load can be the same if the communication overhead is kept to a minimum. The restart interval is therefore an essential property of a computational network, when considering concurrency. The restart interval is defined as the third measure in the concurrency model. It is given by Equation 4.12 for a compute node n, by Equation 4.13 for a network component nc and by Equation 4.14 for a computational network. The restart measure for the computational network uses the run-time of the network component (Equation 4.6) to find the maximum run-time of all compute nodes in the computational network.

$$restart_n = \frac{1}{T_r^n} \tag{4.12}$$

$$restart_{nc} = \frac{1}{\max\limits_{n \in N} T_r^n} = \min_{n \in N} restart_n \tag{4.13}$$

$$restart = \frac{1}{\max\limits_{nc \in NC} T_r^{nc}} = \min_{nc \in NC} restart_{nc} \tag{4.14}$$

The value of the restart measure is in $(0, 1]$, as the run-time of a compute node is in $[1, \infty)$. A large value for the restart measure means that the restart interval is small, computations can be started quickly after each other. On the other hand, a small value indicates that there is a long time between the starts of two computations.
The previous concurrency measures, computation and execution load, produce for every computational network meaningful values over the whole range from 0 to 1 if changes in the computational network are made. The restart measure will not that easily spread its values over the whole range. Only in the extreme cases – execution-time close to 1 or ∞ – it will produce values at the edges of the range. This is not a desired property, as one would want that the values of the measure are well spread over the whole range for all solutions of a given application. The problem is that it is cannot be decided what this range is in the case that we do not know the application in default. Therefore, it is not possible to solve this problem by changing the restart measure.
The following example shows what the implications of this drawback are and how it can be overcome partially.

Example 4.4 Let us consider two computational networks CN_1 and CN_2 that realize the same computation. The maximum run-time of the network components in the computational network CN_1 is 1000. The compute nodes in computational network CN_2 have been changed such that the same computation is still realized, but the maximum run-time of the network components in the computational network is now 100. The slowest compute node in CN_2 is thus 10 times as fast as the slowest compute node in CN_1.
The restart measure for CN_1 is equal to 0.001 and for CN_2 its equal to 0.01. Looking at the values of the restart measure its difficult to see that computational network CN_2 is much faster than computational network CN_1. This will even be harder if the execution-time of a computational network is 1000000 and another computational network that realizes the same application is two times faster. This relative difference in restart interval must be made visible when comparing different solutions of the same application. This can be done by normalizing the values of the restart measure over a set of designs with the largest value.
If we do this for the computational networks CN_1 and CN_2 we get a value of 0.1 for the computational network CN_1 and a value of 1 for the computational network CN_2. These values show clearly that the computational network CN_2 has 1/10 of the restart interval of CN_1.
The disadvantage of this approach is that the 1 value for CN_1 may suggest that the restart value is optimal whereas this is obviously not always the case. ■

4.3.4 Synchronization

A distributed system performs a parallel computation. The parallel implementation of the computation will in most cases be faster than a sequential implementation of the computation. This is because parts of the computation can be done in parallel. To what extent this can be done depends on the synchronization that is required between the different compute nodes in the computational network and how well the computation is balanced over the different compute nodes. To what extent the computation is well balanced over the different compute nodes is measured in the execution load. The influence of the synchronization is not captured in this, or one of the other concurrency measures. Synchronization is important when considering concurrency, because synchronization is limiting the execution of compute nodes and with that the number of compute nodes that can run in parallel. Synchronization constraints may impose the restriction that two compute nodes can only execute one after another, making the computation realized by these two compute nodes longer than the time required to execute each one of them. The synchronization determines in this way the time that a computation will take in a computational network. This time is measured as the total execution time (See Definition 3.7).
The goal of a parallel computation is often to get a solution faster than a sequential computation. This is often in literature referred to as speed-up [28]. This speed-up is limited by the synchronization, as it determines the total execution time. It is also limited by the communication overhead, but that is already covered in the computation

load. To capture this speed-up and limitations imposed on it by the synchronization, we introduce a new concurrency measure, the *synchronization*. The measure is given by Equation 4.15. The measure uses the inverse value of the speed-up achieved by the computational network compared to a sequential solution. This value is subtracted from 1 to meet with our objective that a value of 1 for a concurrency measure indicates a good solution from the concurrency point of view.

$$synchronization = 1 - \frac{T_E}{T_{SE}} \tag{4.15}$$

The range of values for this measure is in $(-\infty, 1]$ in which a value close to 1 indicates a good solution and 0 a solution that is as fast as the sequential computation. Negative values indicate that the computation is even slower than a sequential computation.

Example 4.5 This example uses the same computational network and event diagram as used in Example 4.1.

The execution time (see Definition 3.6) of the four compute nodes can be determined using Figure 4.2. This execution time is equal to 4 for the compute nodes a, b and c and equal to 6 for compute node d. The compute node in the computational network with the longest execution time is compute node d. The figure shows further that the total execution time (see Definition 3.7) is equal to 12. Using Equation 3.5, we find a sequential execution time of 18.

Using Equation 4.15, we find the following value for the synchronization:

$$synchronization = 1 - \frac{12}{18} \approx 0.33$$

This value for the synchronization indicates that the computation using the given computational network is 33% faster than a sequential computation. ■

The previously introduced concurrency measures, computation load, execution load and restart contained measures for the compute nodes, network components and computational network. The synchronization measure is only given for the whole computational network. This is done because it is impossible to determine the computation time of a network component. It cannot be decided with the used time-stamping mechanism whether an idle-time is caused by a causality relation imposed by an event in a compute node inside the network component or by a causality relation imposed by an event in a compute node outside of the network component. In other words, the total computation of a network component may depend on an event outside the network component. This is undesirable as the measure must be independent of its environment. As it is not possible to determine computation times for the network components, it is not possible to determine a synchronization measure for them. This problem can be solved with vector time [9, 26]. Vector time provides the possibility to decide whether an event in a compute node of a network component depends on an event in a compute node in another network component. It was decided to not use vector time, as its is much more compute intensive than the time-stamping mechanism. The idea is that the time-stamping mechanism already provides enough information, so that the overhead of computing vector times is not justified.

4.3.5 Structure

The previous sections have introduced four concurrency measures that use the time-stamping mechanism based on Lamport's logical clocks to measure certain concurrency properties of the computational network. The structure of the computational network plays only an implicit role in these measures. The structure of the computational network reveals the task and data-parallelism that is present. The task-parallelism is made explicit in the computational network through the compute nodes that can perform transformations in parallel. The data-parallelism is transformed in task-parallelism by introducing separate compute nodes for the different data streams.
A simple computational network of a streaming application will consist of a single chain of compute nodes from the input ports of the computational network to the output ports of the computational network. This can be seen as a path through the computational network that realizes a computation. In terms of microprocessors, this can be seen as a pipeline in which every compute node represents a stage of the pipeline. Just as in the pipelines of a microprocessor, it is possible in the computational network to have feed-forward or feed-backward connections in the computational network.
The strings of data in a more complex computational network, particularly those containing data-parallelism, will not all pass the same chain of compute nodes from the input ports of the computational network to the output ports of the computational network. There will be different paths from the input ports through the compute nodes to the output ports of the network. This implies that not all strings of data are transformed through the same chain of compute nodes; parts of the transformations, computations, are done in different paths – chains of compute nodes.
This observation is used to create a formal analysis method for the structure of the computational network. This analysis method is used to derive the fifth measure of the concurrency model.

The first step in the analysis of the structure of a computational network is to transform it into a graph. This graph transformation removes the hierarchy present in the computational networks model. During this project, it was not possible to find a graph transformation in which this information is preserved. The transformation is done in the following way. Let $CN = (NC, C, I, O)$ be a computational network with NC a set of network components and C a set of connections. This network can be represented as a graph $G = (V, E)$, with V a set of vertices and E a set of directed arcs. A vertex v represents a compute node n that is inside a network component $nc = (N_{nc}, C_{nc}, I_{nc}, O_{nc}) \in NC$. A directed arc e_{ij} between the vertices v_i and v_j represents a connection $c \in c_{nc} \in C$ from the output port of compute node v_i to the input port of a compute node v_j with nodes $v_i \in nc_i \in NC$ and $v_j \in nc_j \in NC$.

The vertices v that represent the compute nodes with input ports of the computational network are called the *start nodes.* This set of vertices is denoted as V_S. The vertices v that represent the compute nodes with an output port of the computational network are called the *end nodes* and are denoted as V_E.

Example 4.6 Given is the computational network of Figure 4.3. This computational network consists of one network component with 8 compute nodes. This computational network must be transformed into a graph representation before we can analyze its structure. All compute nodes in the network component are transformed into corresponding vertices. The edges between the vertices are found by considering the connections between the compute nodes. The graph representation of this computational network is shown in 4.4.

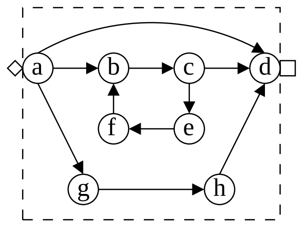


Figure 4.3: Computational network

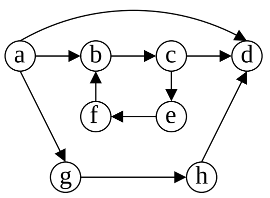


Figure 4.4: Graph representation

The input port of the compute node a is the only input port of the computational network. The vertex a that represents the compute node belongs therefore to the set of start nodes. The output port of the compute node d is the only output port of the computational network. The vertex d that represents the compute node d forms therefore the set of end nodes. ■

Using the graph model of the computational network, we define a *path* as follows.

Definition 4.1 (Path) A path S in a graph $G = (V, E)$ is a sequence $S = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices from a vertex v_0 to a vertex v_k for which holds:

i) $v_0 \in V_S$;

ii) $v_k \in V_E$;

iii) $(v_{i-1}, v_i) \in E$ for $i = 1, 2, ..., k$;

iv) v_i at most 2 times in S for $i = 0, 1, 2, ..., k$.

A path represents a chain of compute nodes through the network components of a computational network. This chain of compute nodes starts at a compute node of which at least one input is an input port of the computational network. The chain ends at a compute node with an output port of the computational network. The definition of a path assumes that all nodes in a computational network are on a path from V_S to V_E. This assumption is realistic, as we are aiming at streaming applications. Normally, these systems require input data (e.g. image, video stream) and they produce output data.

The computational networks that specify these systems will therefore have at least one input port and one output port.
A path that does not go through a cycle in the graph is a path that contains no duplicate nodes, as each vertex of the graph is present at most once in the path. The graph may however contain cycles. The last constraint of the path definition guarantees that in that case that a path can go at most once through a cycle in the graph.

Example 4.7 This example continues with the computational network of the previous example.
The graph contains four paths: $p_1 = \langle a, d \rangle$, $p_2 = \langle a, b, c, d \rangle$, $p_3 = \langle a, b, c, f, e, b, c, d \rangle$, $p_4 = \langle a, g, h, d \rangle$. The path p_1 goes directly from compute node a to compute node d. The path p_2 goes through the compute nodes a, b, c and d. The path p_3 goes also through these compute nodes, but it traverses also through the cycle $b \rightarrow c \rightarrow e \rightarrow f \rightarrow b$. ■

Example 4.7 shows the paths that are found in the computational network that is used in this example. The compute nodes that belongs to the path p_1 are a subset of the compute nodes that belong to the path p_2. The path p_1 is therefore called a *sub-path* of path p_2. Each path c_1 that contains a subset of the compute nodes found in another path c_2 is called a sub-path of path c_2. Path c_2 is then called a *super-path* of c_1 as it contains a super-set of the compute nodes found in path c_1.

The paths that are present in a computational network can be grouped in *computational paths*. These computational paths are defined in Definition 4.2. A computational path is a set of paths that all have the same input and output node of the computational network and contain only nodes from some super-set of compute nodes. In other words, it groups all paths that belong to the same set of subsequent transformations.

Definition 4.2 (Computational path) A computational path is defined as the tuple (u, u', C) with $u \in V_S$, $u' \in V_E$ and C a set of paths. For every path $\langle v_0, v_1, v_2, \ldots, v_k \rangle \in C$ holds $v_0 = u$ and $v_k = u'$. For the set of paths C holds:

i) For each $p_1, p_2 \in C$ holds that p_1 is a sub-path of p_2 or p_2 is a sub-path of p_1 (i.e., C is totally ordered using the sub-path relation);

ii) C is maximal, i.e., there is no path p not in C that can be added to C such that C is still totally ordered.

Example 4.8 This example uses the paths found in Example 4.7. The paths p_1, p_2 and p_3 of this example go through the same set of vertices $\{a, b, c, d, e, f\}$, they can therefore be grouped into one computational path $c_1 = (a, d, \{p_1, p_2, p_3\})$. The path p_4 goes through the vertices a, g, h. These vertices are not a subset of the vertices found in the other paths. Path p_4 does therefore not belong to the computational path c_1. The vertices found in path p_1 are however a subset of the vertices found in path p_4. These paths must therefore be grouped into another computational path $c_2 = (a, d, \{p_1, p_4\})$.
■

The computational paths in a computational network represent the different data flows that go through the system during the computation. Exploiting parallelism implies that it is tried to maximize the number of different data flows. These data flows should be as distant from each other as possible. In other words, they must share as little compute nodes as possible. Sharing a compute node implies that two data flows have to synchronize in order to execute the compute node that needs data from both computational paths. This synchronization hinders the exploitation of concurrency. Whether the computational network has a concurrent structure can therefore be seen by looking at the number of computational paths that go on average through a node.
This observation has lead to the definition of the *structure* measure. This measure is given by Equation 4.16. The measure is zero if all computational paths go through all nodes. This implies that there is no parallelism in the structure. A value close to one indicates that the structure of the computational network is very parallel.

$$structure = 1 - \frac{av}{c} \tag{4.16}$$

,with c = #computational paths and av = average number of computational paths through a compute node.

Example 4.9 The previous example shows a computational network that contains two computational paths. Both computational paths go through the compute nodes a and d. All other compute nodes in the computational path belong to only one computational path. On average there goes 1.25 computational paths through every compute node. The parallelism in the structure is equal to:

$$structure = 1 - \frac{1.25}{2} \approx 0.4$$

■

Example 4.10 This example uses the computational network of Figure 4.5. The computational network consists of one network component with 7 compute nodes. The computational network has one input port and one output port. The figure shows that there is one data stream in the computational network through the nodes a, b, c and d. There are two feed-backward paths connected to this data stream. One goes through the node e and the other one through the nodes f and g.
The graph representation of this network is shown in Figure 4.6 and the path that found in the graph are listed in Table 4.6.

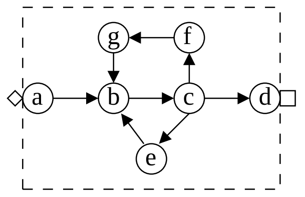


Figure 4.5: Computational network.

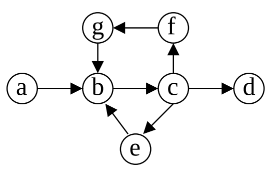


Figure 4.6: Graph representation.

path	
p_1	$\langle a, b, c, d\rangle$
p_2	$\langle a, b, c, e, b, c, d\rangle$
p_3	$\langle a, b, c, f, g, b, c, d\rangle$

Table 4.6: Paths in computational network.

The paths found in the graph representation of the computational network must be grouped in computational paths. The paths p_1 and p_2 must be combined in one computational path. Both paths start in node a and end in node d and p_1 is a sub-path of p_2. Path p_3 cannot be added to this computational path, as p_3 is no sub-path of p_2 and p_2 is no sub-path of p_3. The first computational path found is thus $c_1 = (a, d, \{p_1, p_2\})$. The path p_3 must be grouped in a different computational path. Path p_3 is a super-path of p_1. The path p_1 must thus also be grouped in this computational path. The computational path c_2 is then: $c_2 = (a, d, \{p_1, p_3\})$.
The paths p_2 and p_3 are both feed-backward connections of the path p_1. The definition of a computational path prevents that these feed-backward connections are combined in one computational path. The definition allows that a path is grouped in a computational path with at most one of its feed-backward connection. ■

4.4 Examples

Consider a computational network that has to perform the following function: $y := 24x + 30$. This function must be realized with a system that has the following three internal events, each with a delay of one logical clock value:

i) $v := 2u$

ii) $v := u + 1$

iii) $v := u + w$

The delay for a read or write event is also one logical clock value. The restart measure has been re-scaled, as is suggested in Example 4.4. One solution will therefore have a value of 1 for the restart measure.

• Solution 1
A first solution to the problem is the computational network shown in Figure 4.7. The computation of the three compute nodes a, b and c is shown in Figure 4.9. The communication delay of the computational network is taken to be zero. As a result, a read event can occur one logical clock value after the corresponding write event occurred.

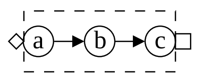


Figure 4.7: Computational network solution 1

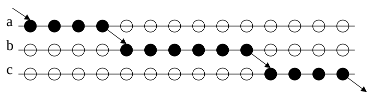


Figure 4.8: Event diagram solution 1

```
Compute node A:       Compute node B:       Compute node C:
   read(x, in)           read(t0, in)          read(t0, in)
   t1 = x + 1            t1 = t0 · 2           t1 = t0 + 1
   t2 = t1 · 2           t2 = t0 + 1           y = t1 · 2
   write(t2, out)        t3 = t1 + t2          write(y, out)
                         t4 = t3 · 2
                         write(t4, out)
```

Figure 4.9: Transformations compute nodes solution 1

All timing information needed for the concurrency model can be extracted from the event diagram shown in Figure 4.8. Numeric values for the concurrency measures can be determined using the Equations 4.4, 4.10, 4.14, 4.15 and 4.16. Their values are shown in the table below. Analysis of the structure of the network reveals that the network contains one computational path, which goes through the three nodes. The structure measure is therefore zero. This is shown in Table 4.7. The computation load of this solution is relatively low because the computation is small. This introduces a large communication overhead. This problem can not be solved and will be present in all presented solutions.

Table 4.7: Concurrency measures for solution 1

Measure	
computation load	0.56
execution load	0.78
restart	0.68
synchronization	0
structure	0

• **Solution 2**

In the previous solution, the execution load was not equal to one. This was caused by compute node b, which had an execution time longer than the other two compute nodes. This can be solved by splitting the compute node in two compute nodes $b1$ and $b2$. The resulting computational network is shown in Figure 4.10. The transformations performed by the compute nodes are shown in Figure 4.12. The compute nodes a and c are the same as in the previous solution. Compute node b has been split in the middle by inserting a read and write operation. The events that compute t_2 and t_3 in the previous example have also been reordered. This is done to avoid having to send both t_1 and t_2 over the connection. This shows that splitting a compute node is not trivial, as it may require re-ordering of the events.

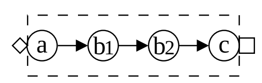


Figure 4.10: Computational network solution 2

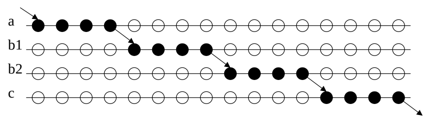


Figure 4.11: Event diagram solution 2

```
Compute node A:      Compute node B1:     Compute node B2:     Compute node C:
   read(x, in)          read(t0, in)         read(t2, in)         read(t0, in)
   t1 = x + 1           t1 = t0 · 2          t3 = t2 + 1          t1 = t0 + 1
   t2 = t1 · 2          t2 = t1 + t0         t4 = t3 · 2          y = t1 · 2
   write(t2, out)       write(t2, out)       write(t4, out)       write(y, out)
```

Figure 4.12: Transformations compute nodes solution 2

It is assumed that the communication delay is still zero in this solution. The timing diagram that presents the behavior of the computational network is shown in Figure 4.11.
Table 4.8 shows the values for the different measures of the concurrency model. If we compare these values with the results of solution one, we see that we have a higher execution load and restart measure. The execution load is one, which means that all compute nodes are always busy with some computation. Solution 2 has a higher value for the restart measure than solution 1, because the slowest compute node requires 4 logical clocks to execute, while this is 6 in solution 1. This balanced execution load comes with a price, the computation load dropped slight because of the extra communication.

Table 4.8: Concurrency measures for solution 2

Measure	
computation load	0.50
execution load	1
restart	1
synchronization	0
structure	0

The event diagram clearly shows that this solution differs with respect to concurrency from the previous solution. The execution load and restart measure of the concurrency model do both indicate this difference. Therefore, at least one of these two measures should be part of the concurrency model.

- **Solution 3**

The execution load is optimal in the previous solution. However, the synchronization indicates that the solution is equal to a sequential implementation of the algorithm. A single computation will therefore take the same time as a computational network that consists of a single compute node. The computational network realizes no speed-up of the algorithm compared to a sequential computation. This means that the execution of the different compute nodes needed for the computation are performed one after another; the compute nodes execute in series. This is an undesirable property, as the goal is to exploit the parallelism in the network. The computational network must be transformed in such a way that a speed-up is obtained and that the number of compute nodes that execute in parallel on one computation is raised. The resulting computational network is shown in Figure 4.13. The compute node declarations are shown in Figure 4.15. The events in the compute nodes have again been reordered. Compute node a has to write its result to two connections. Compute node c has to read the data from two connections. To still have the same execution time for all compute nodes, the addition in this node is moved to the compute nodes $b1$ and $b2$. This gives us again a computational network that realizes the required function. The required transformations to the compute nodes are not trivial, just as it was in solution 2. It is again assumed that the communication causes no delay.

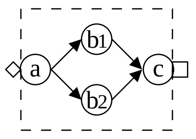


Figure 4.13: Computational network solution 3

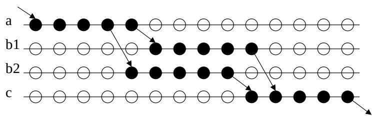


Figure 4.14: Event diagram solution 3

```
Compute node A:        Compute node B1:      Compute node B2:      Compute node C:
    read(x, in)            read(t0, in)          read(t0, in)          read(t0, in2)
    t1 = x + 1             t1 = t0 · 2           t3 = t0 + 1           read(t1, in1)
    t2 = t1 · 2            t2 = t1 · 2           t4 = t3 · 2           t2 = t0 + t1
    write(t2, out2)        t3 = t2 + 1           t5 = t4 + 1           y = t2 · 2
    write(t2, out1)        write(t3, out)        write(t5, out)        write(y, out)
```

Figure 4.15: Compute node declarations solution 3

The timing diagram that corresponds to the behavior of the computational network is shown in Figure 4.14. The figure shows that part of the computation is now performed in parallel (execution of compute nodes $b1$, $b2$). As a result, the synchronization will no longer be zero. Figure 4.13, shows a computational network that contains two computational paths. This implies that not every node is on the same computational path. There is thus parallelism present in the structure. The numeric value for the synchronization, structure and the other measures in the concurrency model are shown in Table 4.9. The restart measure is changed compared to solution 2. This is due to the introduced communication overhead.

Table 4.9: Concurrency measures for solution 3

Measure	
computation load	0.50
execution load	1
restart	0.80
synchronization	0.30
structure	0.33

The values of the concurrency measures of solution 3 are different from solution 2. This is correct, as solution 3 is clearly different compared to solution 2 with respect to the concurrency that is available in the computational network. This difference is indicated by the different values for the synchronization, structure and restart measure. This difference is indicated the best by the synchronization and structure measure. Therefore, at least one of these two measures should be present in the concurrency model.

• **Solution 4**

The previous solutions assumed that the communication causes no delay. Consider now the same computational network as in the previous solution, but with a communication delay of one logical clock value for all connections. The timing diagram for this situation is shown in Figure 4.16. Table 4.10 shows that the synchronization is the only measure that changes. It is lower than in the previous solution. This is because the communication delay lowers the number of events that can occur simultaneously.

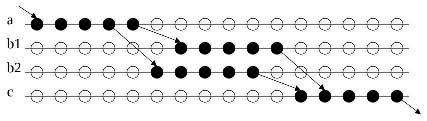


Figure 4.16: Event diagram solution 4

Table 4.10: Concurrency measures for solution 4

Measure	
computation load	0.50
execution load	1
restart	0.80
synchronization	0
structure	0.33

The difference between this solution and the previous one is indicated by the changed synchronization. Therefore, the synchronization must be present in the concurrency model.

The structure measure must also be present in the concurrency model, as this measure indicates the possibility to execute compute nodes in parallel, if the communication is not preventing this. Whether or not the communication prevents this is indicated by the synchronization.
The synchronization measure suggests in this solution that the computation is sequential, but the structure measure indicates that this is a coincidence. The computation is not sequential, but due to communication overhead it takes the same time as a sequential solution.

• **Solution 5**
As a last solution to the problem we construct a computational network that contains a single compute node. The transformation performed by the compute node is shown in Figure 4.17. This solution is created by combining the three compute nodes of solution 1. The sequential execution time, total execution time of the computational network, and the execution time and run time of the slowest compute node are all equal, as there is only one compute node in the computational network. This compute node has as a result an execution load of one and a bad synchronization measure of zero. Parallelism in the structure is not present. The numeric values for these and the other measures in the concurrency model are shown in table 4.11.

```
Compute node A:
   read(x, in)
   t1 = x + 1
   t2 = t1 · 2
   t3 = t2 · 2
   t4 = t2 + t3
   t5 = t4 + 1
   t6 = t5 · 2
   t7 = t6 + 1
   y = t7 · 2
   write(y, out)
```

Figure 4.17: Transformation compute node solution 5

Table 4.11: Concurrency measures for solution 5

Measure	
computation load	0.80
execution load	1
restart	0.40
synchronization	0
structure	0

It will be clear that there is a difference between the second solution and this one if concurrency is considered. This difference is indicated by the value of the restart measure in the concurrency model. To make the distinction between these two solutions, the restart measure must be present in the concurrency model.

4.5 Related Work

Section 4.3 extended the computational-network model with a concurrency model. This concurrency model contains five different measures to capture the different concurrency properties of a system. This section present a brief survey of other concurrency models found in literature. It will also discuss the relation between these models and our model. For an extensive discussion on models and languages for parallel computation, we refer to [33]. The article surveys parallel programming models and languages and assesses their suitability for realistic portable parallel programming. They argue that an ideal model should be easy to program, should have a software development methodology, should be architecture-independent, should be easy to understand, should guarantee performance, and should provide accurate information about the cost of programs. We agree with them that these criteria are important for a realistic model of parallel computation. It is our belief that the concurrency model is easy to understand and use. The idea behind the concurrency model is that a system designer must optimize the five concurrency measures. To what extend this is realized can directly be seen by looking at the measure and looking at how close they are to one. In the implementation, presented in the next chapter, we try to meet with the other criteria.

During a literature search conducted at the start of this project, we found basically two different type of concurrency models. In the first type, the concurrency models that try to measure the performance of a parallel program and compare that with the performances of a sequential program. Good examples of this are [37], [25], [28] and [15]. They simulate the parallel program and measure its run-time. This run-time is used to estimate the run-time of the parallel program if it is distributed over a set of processors and to estimate the run-time of a sequential version of the program. They compute then the speed-up realized by the parallel program. This speed-up is the ratio between the run-time of the sequential and parallel program. The most important difference between these concurrency models is the way in which they determine the run-time of the parallel and sequential program. The synchronization measure of our concurrency model is closely related to these concurrency models. The synchronization measure uses the same ratio to provide an insight in the speed-up achieved by the computational network compared to a sequential solution.
A second type of concurrency models are the more formal models. These models operate often on a higher level of abstraction than the before mentioned concurrency models. Examples of this are [11], [32] and [31]. The approach of Raynal [31] is interesting as there is a large resemblance with our approach. Both the model of computation used in our approach and Raynals approach uses a time-stamping mechanism based on logical clocks. Both approaches associate also a delay with the events executed in a node. The concurrency model proposed by Raynal is comprised of one measure. In terms of our model, it calculates the sum of the idle-time of all nodes that occurred in the

computation and normalizes this with the idle-time of all nodes that must occur because of the causality relation imposed on the events in the nodes. This measures the time that nodes are busy compared to the time that the are idle because of the causality relations. The measure is closely related to our execution load. The difference is that the execution load takes all communication idle-time into account and not only the idle-time caused by the causality relations.

Concurrency models related to our work can also be found in the field of systolic processors. Systolic processors are a class of pipelined array architectures proposed by Kung and Leiserson [19, 20]. A systolic array is a network of processors which rhythmically compute and pass data through the system. Systolic processors accomplish a speed-up of the computation. They are not suited for speeding-up the communication, but allow for the balancing of communication and computation [20]. Research to performance analysis and design optimization has been performed in the field of systolic processors [21]. For the optimality criteria they often consider the computation time, latency, throughput and the number of processors and size of communication network. We do not consider the number of processors or the size of the communication network in our concurrency model, but we do consider the other criteria.

4.6 Conclusion

This chapter introduced a formal concurrency model that can be used to analyze the concurrency in a computational network. The concurrency model is based on five different measures: computational load, execution load, restart, synchronization and structure. The computational load and execution load indicates to what extent the computation is balanced over the computational network and how much of the effort is put in transforming the data compared to communication. The synchronization provides an insight in the speed-up that is achieved by creating a concurrent solution compared to a sequential solution. The restart is used in the concurrency model to provide feedback over the rate at which new computations can be started on the computational network. The parallelism that is present in the structure of the computational network is identified, to some extent, using the structure measure. These measures are explained in Section 4.3. Three of the five main measures are derived from a set of detailed measures which describe the computation load, execution load and restart for compute nodes and network components. These detailed measures can be used to get a better insight in the concurrent behavior of the elements of the computational network.
Section 4.4 presents a number of computational networks that realize the same computation, but have different concurrency properties. It is shown that those differences are indicated by the concurrency measures. It is further shown that all measures are needed to make a clear distinction between the different solutions that are presented in this section.

The concurrency model was introduced in this chapter by observing what properties are important when considering concurrency in a computational network. It was not considered what properties are important for a system designer at the point that the concurrency measures were introduced. Let us discuss now how the measures relate to the properties of a system considered to be important by a system designer. A system designer will in general be concerned about the latency and throughput of the system. The latency is measured in the computational-network model in the total execution time. The latency relative to a sequential solution is used in the synchronization measure. To minimize the latency, one should try to optimize the synchronization. The throughput is related to the restart interval and as such to the restart measure. The system designer must optimize the restart measure in order to get an optimal throughput. The system designer can use the concurrency model to get feedback about the latency and throughput of the specified system.

Chapter 5

Implementation

5.1 Introduction

The previous chapter introduced a concurrency model that can be used to analyze the concurrency in a computational network. The computational network itself was introduced in Chapter 3. The following two sections introduce an implementation for the computational network and the time-stamping mechanism that is used in the model of computation. Section 5.4 presents a software program that implements the concurrency model. The software program takes a computational network as an input. It analyzes this computational network for its concurrency properties by simulating the computational network and analyzing the data produced by this simulation. The output of this analysis are the numerical values of the concurrency measures.
Simulating the network does not contradict with our desire for abstraction. The concurrency model is based on the time-stamping mechanism. This time-stamping mechanism requires a static list of events that occur in the computation. To get this static list of event, we have to simulate the computational network with a given input. To allow abstraction from this input, we must use multiply simulations and perform statistical analysis on them. This gives us the desired abstraction.

5.2 Computational Networks

The definition of a computational network allows the modeling of a number of different models of computation known from literature (see Section 3.2). Amongst them are the Kahn process networks (KPNs) and Dataflow graphs. These two models of computation are mainly used for the modeling of signal processing applications. The objective when developing the computational-network model was to get a model of computation that is suited for modeling embedded applications. These embedded applications are often signal processing applications. Therefore it seems logical to consider the possibility to extend an existing implementation of Kahn process networks or Dataflow graphs to get a suitable implementation of the computational-network model for a first experimental evaluation of the concurrency model. An advantage of this approach is that it allows the reuse of code that is written for YAPI.

YAPI, the Y-chart application programming interface, is a programming interface developed at Philips Research [16, 17]. It is used to model signal processing applications as process networks. The YAPI model of computation extends the existing model of KPNs with channel selection to support non-deterministic events. A C++ library is provided as an implementation of this model of computation. This makes YAPI suitable as a first implementation of the computational-network model. The compute nodes of the computational-network model become the processes of the YAPI/KPN model. A computational-network of the computational network model is equal to a process network in the YAPI/KPN model. The connections of the computational-network model are represented by the fifo's in the YAPI/KPN model.

5.3 Time-Stamping Mechanism

To implement the time-stamping mechanism based on Lamport's logical clocks, we need an implementation for the different delay functions used in this time-stamping mechanism. The time-stamping mechanism contains two delay functions. One that associates a delay with the communication of data over a connection. And one that associates a delay with each event that occurs in the compute nodes. The delay function for events in a compute node will be implemented using a delay function for internal events and a delay function for read and write events. The reason for using two delay function for events becomes clear when we discuss their implementation.
Section 5.3.1 describes the implementation of the delay function for internal events. The delay function for read and write events is described in Section 5.3.2. Section 5.3.3 discusses the implementation of the delay function for connections.

5.3.1 Delay Function for Internal Events

Computational networks are implemented using C++. This implies that a single event in a compute node is equal to a single C++ statement. The delay function for internal events must map each C++ statement on a delay value associated with that statement. This should be done in a way that the value of the delay represents the amount of work associated with performing the event.
The amount of work associated with performing an event is often expressed by the time needed to perform the event. This represents the resources of a system that are used by an event. This time can be measured using a wall-clock or the systems clock. These values are difficult to measure, as it is difficult to determine the exact times at which the event started and ended. It is difficult to determine the number of cycles that are needed to perform an event. Furthermore, it is very architecture dependent and thus not very abstract. However if we assume that the average statement found in the code of the compute node, normalized to the number of machine-instructions needed to execute it, takes the same number of cycles to complete, then it becomes possible to determine a

value for the time needed to perform an event that is a little more abstract. It remains to be seen whether it is sufficiently architecture independent to be useful in practice. However our first experiments show that the assumption holds.
The assumption now is that the number of instructions executed for an event is a good measure for the work associated with the event. The number of instructions needed for a statement can be determined using a cycle-accurate simulator. The problem with using this kind of simulator is that they have long run-times. A solution to this is the system used in VCC [5]. The Virtual Component Co-Design environment provides an option to annotate the C code of a program with the delay associated with executing the behavior on a given architecture. This gives a performance model for a behavior implemented in software without running it on the processor. This allows for fast simulation and rapid design exploration.
This is the kind of profiling that we propose to implement for the delay function for internal events. It associates a reasonably accurate delay with each statement, but allows for fast simulation as no instruction set simulator is used. The latter is needed because the concurrency optimization is performed in an early stage of the design where still a lot of architecture exploration will be performed. This requires that simulations can be performed reasonably fast.

To implement the above described profiling mechanism, we could use VCC to profile the statements inside the code of the compute nodes. The problem is that VCC is not publicly available; therefore we decided to create our own VCC like implementation of this profiling mechanism.
The required mechanism is quite simple. The information required to profile the code is the number of instructions needed for each statement in the source code. This information can be obtained in the following way. Consider the situation in which a single line in the source code contains exactly zero or one statements. Normally it is possible to relate each assembler operation to a line in the source code. As each line contains exactly zero or one statements, each assembler operation is related to a single statement. Counting the number of assembler instructions related to a specific line gives the number of assembler instructions related to a specific statement. Using this method, it is possible to assign a delay to each statement in the sequential code of a compute node.

The delay can be determined using a compiler that is available on the machine that runs the simulations used in the design process. This machine will in general contain a general purpose CPU. The final implementation may contain processors with a different instruction set. This implies that in the implementation there may be different delays associated with the events in the compute nodes. The determined delay values do depend on the architecture of the system.
The best way to overcome this problem is to use a compiler for the processors that are used in the implementation. In case of a heterogeneous system, this may require the use of multiple compilers. It is in that case required that a mapping of compute nodes to resources is made.

In the case that the homogeneous multi-processor system contains only general purpose CPUs like Pentiums or MIPS, it is assumed that the delays do not have to be calculated for a specific architecture. They can be determined using a compiler for another general purpose CPU that is not used in the system but is available on the machines that run the simulations.

The time-stamping mechanism was introduced in Section 3.3 to allow reasoning about causality and some timing aspects on a relative high level of abstraction without referring to implementations/physical time. To get accuracy, we relate the events to the number of instructions needed to execute these events on a processor. To get abstraction from the system architecture we assume that the influence of a specific instruction set of a processor does not have that much influence on the results. Our first experiments show that the proposed notion of time is both accurate and abstract enough to perform target architecture independent concurrency optimization.

5.3.2 Delay Function for Read and Write Events

For the delay function of internal events, we use the number of instructions needed to execute the event. The same approach could be used for read and write events. However, the read and write functions used in our software library as communication primitives will in general not be used in the actual implementation. The implementation will use a software library containing read and write functions optimized for the given system. The functions will then require a different number of cycles; they have a different delay. It is also possible that the implementation uses no read and write function at all. The communication primitives of YAPI may be replaced by different, more efficient, communication primitives.
Let us consider the case that the YAPI communication primitives are used. A compute node will produce some data that it places in a memory. When the compute node writes this data to the connection, it is copied from the memory location where it resides to the memory location used for the connection. On a read event of another compute node, the data is transferred from the memory location of the connection to the memory assigned to the compute node. The data is copied in this system a number of times. Option I of Figure 5.1 shows an example of this. The dashed arrows in this figure represent the different memory transfers involved in the communication excluding the required synchronization messages.
To avoid the copying of data, one can use different communication primitives. Option II shows a solution that can be used in a system with a shared memory. A processor is informed with semaphores about the memory location to which it can write or from which it can read. The run-time system that implements the communication primitives must guarantee that processors read from and write to the correct locations in the shared memory. A technique for mapping process network communication onto a multi-processor architecture with shared memory that uses semaphores is action synchronization [1].
A multi-processor system that contains no shared memory cannot use the communication

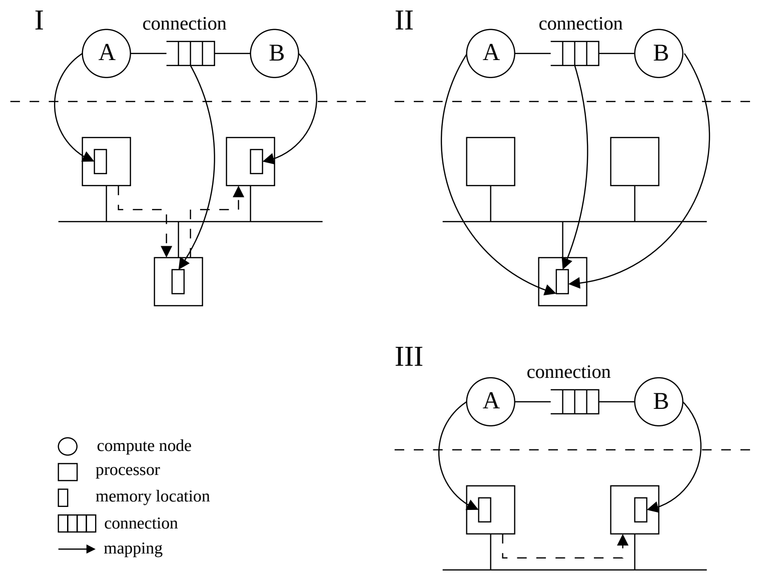


Figure 5.1: Data communication in multi-processor system.

primitives of option II. But instead of using the communication primitives of option I, it can use those of option III. In option III, the processors have their own local memory in which the data is stored. The compute node A will write the produced data to its own local memory. As soon as the processor running compute node B needs the data written by compute node A, it will request it from the processor running compute node A. The data is then directly copied from the local memory of the processor running compute node A to the processor running compute node B. The data needs to be copied only once in this solution. The processors however are responsible for not overwriting the produced data.
There is still another solution, Option IV. This option is not shown in the figure. Option IV assumes that all data output by a compute node will be needed at some point and only by one compute node. The data can in that case be copied to the receiver as soon as possible. In practice, this solution will fail in many implementations of computational networks for the following reason. The data produced by a compute node may be needed by multiple compute nodes, i.e., one-to-many connections will be present in implementation of computational networks. The producing compute node must output the data as many times as there are receivers in case option IV is used. This problem can be avoided if option III is used.

In an implementation, one will use a technique similar to option II or III for communication. Option I will not be used as it is always more expensive than option III. Option IV is not used for the above mentioned reason. Consider now the case that option II is used. The communicated data is already available in the memory when a read or write call is issued. The only thing that needs to be done is to communicate the location in the memory of it between the processors. This can be done in a constant time with

respect to the number of data elements transferred. If option III is used, there is also a constant time needed in the communication costs for setting up the communication channel. But the total communication costs will also depend on the time needed to send all data-elements over the physical communication medium.
To model all of the involved effects completely, one needs information about the communication architecture of the system. This requires for instance information on the average access times to the communication medium, size of the buffers inside the I/O units of the processors, mapping of compute nodes on processors etc.
When a design exploration is started, this information will not be available in this detail. The delay function for read and write events can therefore not take it into account. From the above observations, it becomes clear that the costs of communication / delay associated with communication is determined by a constant time needed to get access to the channel and a time that depends on the number of data-elements communicated. The first can be taken into account in the delay function for read and write events by a constant $a \geq 0$, the latter can be taken into account by a constant $b \geq 0$ that is multiplied by the number of data-elements communicated. The a, b constants must be integers. This guaranties that the delay has an integer value. This is required as the logical clock can only have integer values. For a read or write event e, we use the delay function of Equation 5.1.

$$d(e) = a + b \cdot \#elements\ communicated \tag{5.1}$$

The constants a and b are determined by the type of communication chosen. Values for these can be determined by running simulations on the targeted multi-processor system if this is known and available.

5.3.3 Delay Function for Connections

The previous section discussed that much of the costs of communication are not known when the concurrency optimization is performed. The delay function for connections can therefore not use much information about the architecture and its characteristics. It is however observed that the costs of communication of a single data-element over a communication medium depends on the size of this element. The bigger the element, the more time is needed on the communication medium, the higher the cost. It seems therefore logical to take the size of the data-element transferred over a connection as a basic measure for the delay associated with a connection. The size of a data-element is expressed in the number of bytes that are needed to store it in memory. To be able to scale the delay of the different connections, we introduce a constant $a \geq 0$ with which the size of the data-element is multiplied to get the value for the delay associated with it. The a constants must be an integer. This guaranties that the delay has an integer value. This is required as the logical clock can only have integer values.
Equation 5.2 gives the delay function for a connection c. Each data-element that is communicated through the connection is assigned this delay.

$$d(C) = a \cdot \#bytes\ of\ data\ element \tag{5.2}$$

5.4 Concurrency Analysis and Simulation Tool

5.4.1 Introduction

Chapter 4 has presented a concurrency model for computational networks that uses a time-stamping mechanism based on Lamport's logical clocks. The implementation of the computational-network model and time-stamping mechanism have been discussed in the previous sections. This section will add to that a software implementation of the concurrency model. This program is called the *Concurrency Analysis and Simulation Tool* or CAST for short.

The measures of the concurrency model are based on the ordering of the events that take place in the computational network (See Section 3.3). This ordering can be computed from a list of all events that are executed in the computational network. The structure measure in the concurrency model requires further information about the structure of the computational network. This information must be gathered before the concurrency measures can be calculated.
The computational network must be simulated to obtain the needed information. During the simulation, a trace must be generated that contains the information needed for the time-stamping mechanism. The simulation must also extract the information on the structure of the computational network.
The trace generated during the simulation contains information about the internal, read and write events that have occurred in the compute nodes of the computational network. An internal event is equal to a single C++ statement. This statement is mapped onto a delay using the delay function for internal events. This delay function takes the number of machine instructions executed as a value for the delay of the statement. The trace file could contain information about the statement executed, which could be mapped onto the appropriate delay when creating the event ordering. However, it is more convenient to map the statement onto the delay before creating the trace. The trace can then contain the delay associated with the executed statement instead of the statement itself. This requires however that the source code is annotated with the appropriate delays. The source code must also contain code to create the trace file and to extract the structure of the computational network.

There are now three steps visible that must be taken care of by the program. First, the source code describing the computational network must be annotated with the values of the delay that is associated with executing the code and code for creating a trace. This step is called the parser step. Second, the annotated source code must be simulated to obtain the trace and information about the structure of the computational network. This is done in the simulation step. In the third step, the analysis step, the trace is used to create an event ordering which on its turn is used to calculate the concurrency measures. The time-stamping mechanism must use the delay functions for communication events and connections to map the read and write events onto the appropriate delay.

The overview of CAST is shown in Figure 5.2. The figure shows the three different steps of the program and the input and output files of each of them. These steps and their input and output will be discussed in the following paragraphs. The .info-files generated by CAST contain the values for the main and detailed concurrency measures.
CAST uses, besides the files shown in the figure, a project file that specifies all of the settings for CAST. This XML file is named default `cast.xml`. Most of the settings contained in this file can also be controlled via the command line. For a more detailed description of these options and the command line interface of CAST, one should look at the brief user manual of the program in Appendix A.

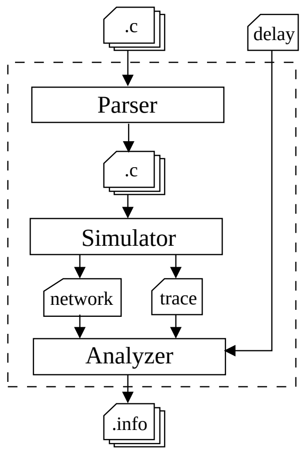


Figure 5.2: CAST overview

5.4.2 Simulator

An important step before the concurrency measures can be calculated is the simulation. The CAST simulator is responsible for creating a trace of all events that have occurred in the computational network during the computation. It must also gather information about the structure of the computational network.
The created trace file does not contain information on which statements (internal events) were executed. It contains only the delay associated with the execution of these internal events. Each time a statement is executed we do not log the delay associated with this event, but instead, we raise a counter associated with the compute node. This counter, the *delay counter*, counts the delay associated with all internal events that have occurred up to the point where the compute node is in the execution. If a read or write event occurs in the execution of the compute node, we do not have to log which internal events have occurred before this read or write event. It is sufficient to log the value of the counter, as that provides the required information for determining the event ordering.

There are mainly two options for creating the trace. The first option is that a trace file could be created for each compute node individually. The result is a set of trace files that have to be parsed when creating an event ordering. The events within the trace file respect the causality relations imposed on them, as these are always respected by the events that occur in a single compute node. However, the causality relation for the events between compute nodes – read and write – is not captured in the data structure. This causality relation must be derived from the trace files by correlating the traced events. This is a complex operation as it requires that all events of all trace files are scheduled in an ordering that respects the causality relation.
The second option is to use a single trace file for all compute nodes. The events that occur

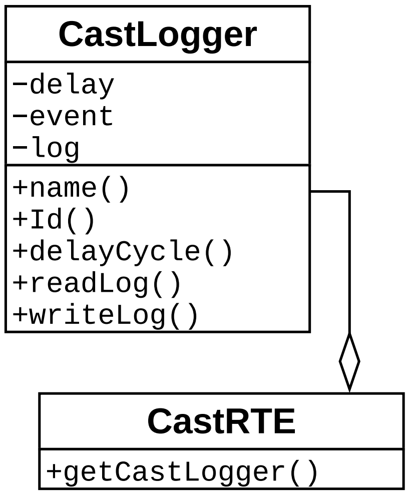


Figure 5.3: CAST run-time environment

within a single compute node are still logged in an order that respects causality. These events are mixed with the events that occur in other compute nodes. This ordering is determined by the order in which the YAPI scheduler executes the compute nodes. The scheduler will obviously do this always in an order that respect the causality relation. It is not possible to log a read event before the corresponding write event has occurred.

A run-time environment is required to log information in the trace and maintain the delay counter of every compute node. This run-time environment maintains a data-structure for every compute node known by the run-time environment. The UML diagram [38] of the run-time environment is shown in Figure 5.3. The run-time environment is available in the program through the global variable `castRTE` which is a CastRTE object. The `castRTE` variable maintains a list of CastLogger objects. A CastLogger object contains the delay counter for a compute node and a reference to the trace file.

A reference to the CastLogger object that belongs to a compute node can be obtained via the `getCastLogger` procedure of the CAST run-time environment. This procedure requires the name of the compute node to find the corresponding CastLogger object in the list of CastLogger objects maintained by the run-time environment. If the compute node is not found in the list, a new object is created for the corresponding compute node. The delay attribute of the CastLogger object is a 64 bit counter used to count the delay associated with the internal events executed in the compute node. This delay counter can be updated and accessed through the `delayCycle` procedure. The CastLogger object contains further an `event` attribute; this maintains the number of read and write events that have been performed by the compute node. A read or write event in a compute node is traced through the `readLog` and `writeLog` procedures of the object. These procedures add a line to the trace file, which identifies the event that has occurred in the compute node.

This line in the trace file must contain information whether the logged event is a read or write event. If this is the case, it must further contain information on the compute node and port that were involved in this event. With these, it is possible to identify which connection is used in the communication. This makes it possible to correlate the read and write events. Finally, the line must contain the value of the delay counter and the number of data elements that are transferred in the read or write event.

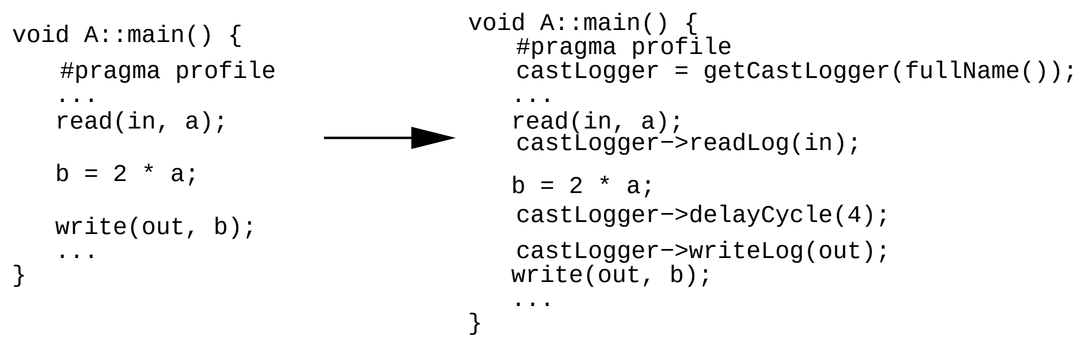
The CAST parser must make the necessary changes to the source code of the YAPI process network for the CAST run-time environment to work. This parser is discussed in the next section. The annotated YAPI process network created by the parser is the input for the CAST simulator. The CAST simulator will first compile the annotated YAPI process network to obtain an executable. This is done using a standard GCC compiler for the platform on which the simulation must be performed. After that, the simulator executes the binary with the arguments for the simulation that were passed to CAST. The binary will execute and run like a standard YAPI program. It will use the YAPI run-time environment to execute the process networks and processes in it. The only difference is that the binary will also use the CAST run-time environment. This run-time environment is used to create a trace of all events that take place in the computational network and an XML file describing the structure of the computational network. This structure information describes all computational networks, compute nodes, ports and their connections found. All these elements are assigned a number, called an *id*. This is a unique number that identifies an element throughout the whole system. These id's are assigned such that the compute nodes in the computational networks have the lowest numbers, starting from zero. How these id's are used is explained when the analyzer step is discussed.

5.4.3 Parser

The first stage in CAST is called the parser stage. It takes as an input a set of C++ source files with the YAPI process network. The parser must annotate the code in these files in order to allow tracing of the internal, read and write events that occur in the compute nodes.
Using the CAST run-time environment, it is possible to create a trace of all events that occur in the computational network during a computation. It is required for the run-time environment that each compute node can access its CastLogger object. One method to implement this is to add an attribute to the object describing the compute node. During initialization of the object it is possible to initialize this attribute with a reference to the CastLogger object. This requires that the CAST program is able to understand a large portion of C++ syntax in order to determine which objects must be identified. Another solution would be to modify the YAPI run-time environment and combine this with the CAST run-time environment. This requires that YAPI is changed. This makes it harder to use new versions of YAPI in the future, as all changes to YAPI have to be redone for each new version of YAPI developed at Philips.
There is a simpler solution that requires slightly more run-time, but only needs a very

simple parser in the CAST program. This solution adds a variable `castLogger` to every function that belongs to a compute node. This `castLogger` variable is valid within the scope of the function and is initialized by calling the `getCastLogger` procedure of the CAST run-time environment with the name of the compute node. The name of the compute node is obtained by calling the `fullName` procedure available in YAPI.
To simplify the parser, we have chosen to implement this last solution in CAST. The parser in CAST must now be able to identify where a function starts and ends in the source code. The parser must now be able to identify the different functions in the source code and modify the body of the functions. The functions that must be modified are member functions of the compute node classes. To determine whether a function belongs to a compute node class, the source code file and all its include files must be analyzed. This is needed to determine which function belongs to which class and whether that class is derived from a compute node class. Doing this in an automated fashion would require a complex parser that has to understand a lot of the C++ syntax. There is a simple solution to overcome this problem. This requires that the user informs the CAST parser that a function must be annotated with delay information. This is the only change that the user has to make to the source code of a YAPI program in order to analyze it using CAST. This annotation works as follows. Whenever the CAST parser finds a `profile` pragma within the body of a function, it will annotate the source code with the appropriate delay for each statement. The CAST parser outputs first the statement and then a call to the CAST run-time procedure `delayCycle` with the delay associated with the statement. This is done until an `end profile` pragma is found or the end of the function body is found.
The CAST parser adds also calls to the `readLog` and `writeLog` procedures of the CAST run-time environment. These are inserted to log the read and write events that it finds in the function body. If a read function call is found in the original source code, this read call is output by the CAST parser followed by a call to the `readLog` procedure of the CAST run-time environment. The call to the CAST run-time environment must be placed after the read call. This ensures that it is performed after the moment that the data was available on the connection. If the read call is logged before it is performed, there is the risk that a read event is logged of which the corresponding write event has not yet occurred. A write event can be logged just before or after it is performed if the connection can store enough data elements to never block a write event. In practice, this will not be true. If a write event would be logged after it has occurred, a read event on the corresponding data may already have occurred and be logged. To prevent this, the CAST parser inserts the `writeLog` procedure before the call to the write procedure.
Figure 5.4 contains a code fragment before and after the CAST parser step.

If an iteration or selection statement is found and no compound statement (open/close brackets) is present around the statement(s) within the iteration or selection, it must be inserted by CAST. This is because the CAST parser may add new statements that have to be executed together with the statements within the iteration or selection. The parser must be able to decide whether or not to insert the compound statement. To help the parser with this, the original source code is pre-processed. After this pre-processing



```
void A::main() {
   #pragma profile
   ...
   read(in, a);

   b = 2 * a;

   write(out, b);
   ...
}
```

```
void A::main() {
   #pragma profile
   castLogger = getCastLogger(fullName());
   ...
   read(in, a);
   castLogger->readLog(in);

   b = 2 * a;
   castLogger->delayCycle(4);

   castLogger->writeLog(out);
   write(out, b);
   ...
}
```

Figure 5.4: Annotated source code.

step, the opening bracket of a compound statement is on the same line in the file as the iteration or selection statement. The parser needs to analyze only one line of the file to decide whether or not to insert the compound statement.
The implementation of the delay function for internal events has been discussed. It requires that a single line of code contains exactly zero or one statements. This requires that the source code is reformatted to meet with this requirement. To perform the necessary pre-processing, CAST uses a program called `indent`. `Indent` is a source code formatting program that is installed on many Linux systems and is available for free. As this program is easily available, it was decided to use it instead of adding a reformat function to the CAST program.

The result after the CAST parser is run on the source code files is a set of annotated files. The original source code files have first been reformatted for CAST. This format is hard to read for humans, as it may contain long lines of code. To create more human-readable code, the output of the CAST parser is reformatted using the GNU coding style in conjunction with the `indent` program. This creates readable source code, which is placed in the `cxx` directory of the project directory.

The implementation of the delay function for internal events is discussed in Section 5.3.1. The delay of an internal event is determined by the number of machine-instructions needed to execute the internal event on a given processor. The problem with this approach is that the final implementation may use different processors, with a different instruction set. The delays used in the delay function are in that case not correct. The section discussed an option to overcome this problem. That requires the use of a compiler for the platform on which a compute node will be mapped. To support this, CAST has an option to generate delay information for internal events using a different compiler than the standard compiler. To make this work, small changed to the code of the parser must be made. The parser must know which compiler to use for the platform.
CAST supports besides the standard compiler also a MIPS compiler. CAST can be instructed to use this MIPS compiler via the project file. CAST generates in that case assembler code from the source code with the MIPS compiler. This assembler code is

used to obtain the delay for the internal events that occur in the compute nodes and annotate the source code. Note that the actual simulation can be done on a different platform than the platform used in the parser step; the parser creates only annotated source code.

5.4.4 Analyzer

The simulator has created a trace, `cast.trace`, of all events that have taken place in the computational network during the simulation. It has also created an XML file, `network.xml`, that describes the structure of the computational network. These two files and the delay settings provided by the user form the input of the analyzer. The analyzer creates an internal data-structure from the network description. It also reads the delay settings and sets the constants of the delay functions of the read and write events and the communication. The trace file is used to create an event ordering. The result is stored in the data structure and is used to calculate the different concurrency measures.
This section describes in detail how all of these steps are performed. Before doing so, we have to make one remark on the implemented concurrency model. The analyzer currently implemented in CAST does not use the hierarchy in the concurrency model as presented in Chapter 4. It uses a flat model for the computational network. This means that the analyzer assumes that all compute nodes belong to one network component, this network component is the only network component in the computational network. The analyzer does not have to store the size attributes of the different network components, as there is only one network component. The formulas for the concurrency measures of the network component and computational network can be simplified in this case. The execution load and computation load can simply take the average over the loads of the compute nodes. The synchronization, restart and structure measure do not change.

Internal Data Structure

The analyzer starts with parsing the network description in the file `network.xml`. The result of this is a reference to a ComputationalNetwork object. This object contains references to objects for all computational networks, compute nodes, ports and connections in the computational network. The UML description of these objects is given in Figure 5.5.

The ComputationalNetwork class and the ComputeNode class both contain a set of InPorts and a set of OutPorts that represent the input ports and output ports of the computational network and compute node. The InPort and OutPort class are derived from a Port class. A Port object stores references to the connectors in which a port serves as a source and as a destination. A connector can be linked to the port through the link procedure. A Connector object contains a reference to a Connection object. The connection class implements the behavior of the connection as defined in Definition 3.2. The read and write procedures must be used by the ports of the compute nodes to read and write data to the connection. This data is stored in a fifo that is implemented as a linked list. Data-elements can now be add and removed in constant time from the start and end of the fifo.

The computational-network model contains network components. These are not present in the UML description of the data structure used by the analyzer. These network components are not necessary as it is possible to generalize network components to computational networks. A network component can simply be seen as a computational network that contains compute nodes. This generalization is used in the data structure of the analyzer.

All classes mentioned so far are derived from the Id class. This Id class stores the name of an object. The complete hierarchical name of the object can be obtained through the `fullName` procedure. The `id` procedure gives the id of the element. The Id object contains further a reference to the parent of the element. For instance, the parent of a compute node is the computational network it belongs to.

The ComputationalNetwork, ComputeNode, Port and Connection class are derived from the EventSystem class. A object of this class stores all data for the time-stamping mechanism. It contains the constants used for the different delay functions. These constants can vary per port, compute node or connection. They are stored in the EventSystem object for each element. The EventSystem object contains also procedures to count the number of read and write events that occur during the execution of the computation. The delay associated with the execution of the internal events in a compute node is also counted. These values and the time measures of Section 3.3 can be accessed through a number of procedures present in the class.

The EventSystem class contains further the procedures `setPrimary` and `isPrimary`. To explain these two procedures, we must consider the problem of how to simulate a computational network. The computational-network model describes a computational network as an element that contains a set of input ports and a set of output ports. The computational network communicates with the outside world through these ports. To simulate a computational network, we have to connect these ports to other computational networks or compute nodes that read data from or write data to the connections of the computational network that is analyzed. The computational networks and compute nodes that simulate the environment are also simulated when the analyzed computational network is simulated. Assumption 4.2 says that the data written by the environment may not cause delay in the computational network that is analyzed. This assumption is satisfied if the local logical clock of the compute nodes that simulate the environment stay at zero during their execution. Compute nodes to which this applies are called *primary nodes*.
Which compute nodes are primary nodes must be specified in CAST. This is done by adding an XML *primary node* tag to the delay settings. The syntax of this tag is described in the following paragraph along with the rest of the delay settings.

Delay Settings

The delay function for read and write events and the delay function for connections need a number of constants to calculate the delay associated with a read or write event or communication. The values for these constants, as well as the definition of primary nodes is done in the delay settings. These delay settings are stored in an XML file. Normally, this is done in the project file used by CAST (`cast.xml`). For an example see Figure 5.6.

The tags of the different delay settings are inside a **`delaysettings`** element. There are two different elements allowed inside this **`delaysettings`** element. The first is the *primary* element. It contains one attribute, *node*, whose value is equal to the hierarchical name of a compute node that is a primary node. There may be an unlimited number of primary nodes defined in the delay settings.

The second element allowed inside the delay settings is the *delay* element. This element contains always a *type* attribute. The value of this attribute is equal to 'rw' or 'connection' and defines whether the delay settings must be used for the delay function of read and write events or for the delay function of connections. If the type attribute is 'rw', then the delay element must contain an element d with a value 'a,b'. a and b are the integer constants used in the delay function. If no comma is found, it is assumed that the b constant is zero. It must further contain a *node* attribute. The value of this node attribute is equal to the hierarchical name of the compute node to which the setting applies. If the value is equal to 'default', then it applies to all compute nodes in the computational network for which no setting is specified.
If the type attribute has a value 'connection', then the delay element must contain also a d attribute with the value for the constant a of the delay function for connections. It must further contain a *connection* attribute with the name of the connection to which the setting applies. The value of the connection attribute may also be 'default'. The settings applies in that case to all connections for which no setting is specified in the delay settings.

CAST assumes at start-up that the a constant of the delay function for read and write events is 1. It takes a value of 0 for the b constant of this function. The delay function for connections uses a default value of 1 for the a constant. These settings can be overwritten by the default settings specified in the delay settings. These values can on their turn be overwritten by values for specific compute nodes and connections in the delay settings.

Event Ordering

The time-stamping mechanism requires that all events traced during the simulation are ordered using the algorithm described in Section 3.3. The EventSystem object of each compute node contains the required counters and procedures for the time-stamping mechanism. An Event object is used to send the time-stamp at which a write event occurred in a compute node over a connection to the compute node that reads the written data.

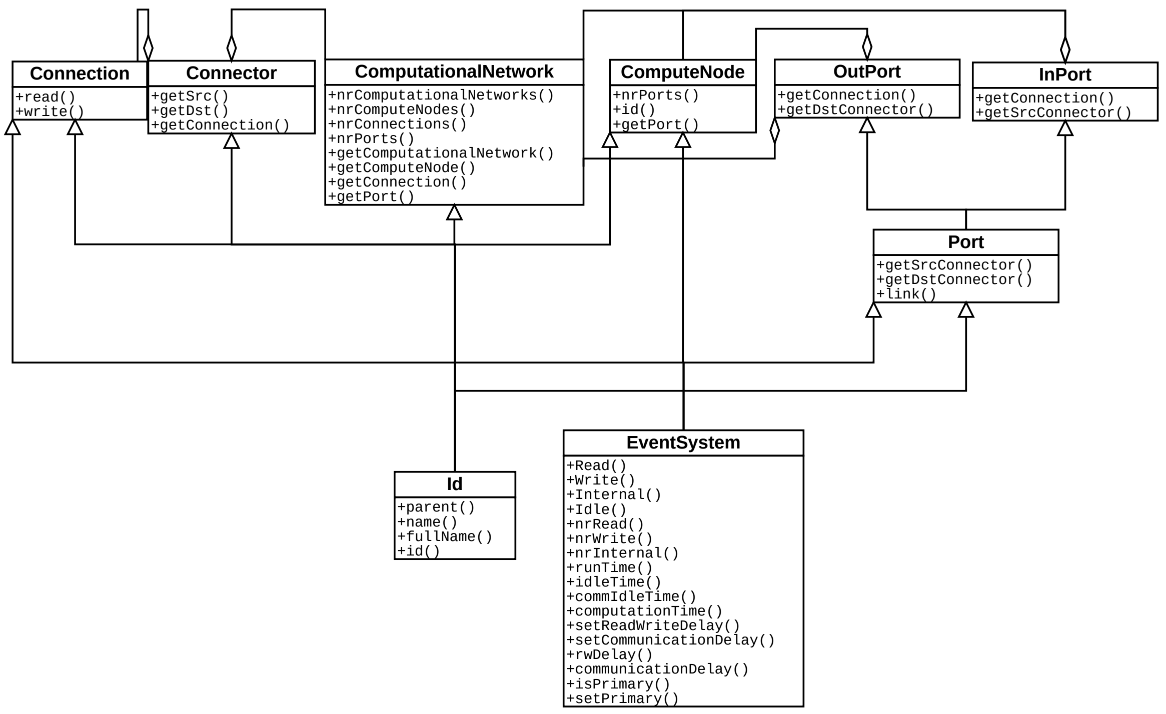


Figure 5.5: UML description of computational-network model.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cast SYSTEM "cast.dtd">
<cast version="1.0">
   <delaysettings>
     <primary node="jpeg.frontend"/>
     <primary node="jpeg.backend"/>
     <delay type="rw" d="10" node="default"/>
     <delay type="connection" d="100" connection="jpeg.pixels"/>
     <delay type="rw" d="1" node="jpeg.vld"/>
  </delaysettings>
</cast>
```

Figure 5.6: Example of the delay settings.

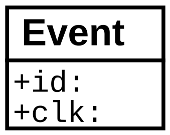


Figure 5.7: UML description of Event class.

The UML description of the Event class is shown in Figure 5.7. When a compute node writes this Event to a connection, it contains a value identifying the event in the compute node, and a value `clk` that is equal to the time at which the event occurred. The connection updates this clk value when it receives the event object. The connection adds to it the delay associated with the connection. This is done using the delay function for connections.

The event ordering mechanism must first find the compute node, port and connection involved in a read or write event. Then it can perform the event. It must also make sure that delay for all internal events that occurred in this node between the previous read or write event and this read or write event in the node is added to the local logical clock. If necessary, to respect the causality relation, idle time must be inserted.

To find the compute node involved in an event, CAST could use the internal data structure described above. The problem with this is that for each event it must traverse through the computational network to find the correct compute node. This cannot be done in constant time. It is possible to access the correct compute node in constant time. To do that, CAST must create a list `computenodes` that contains references to all the compute nodes in the computational network ordered with their id. This list uses the property that the id's of the compute nodes have the lowest number, starting from zero. This makes it possible to access through an array directly a compute node when its id is known. The id of the compute node is simply the location in the array where the reference to the compute node is stored.

The event ordering mechanism can now read a single line from the trace. This line contains the id of the compute node involved in the read or write event and the name of the port involved in the read or write event. Using the id of the compute node and the name of the port, it is possible to find the compute node, port and connection involved in this read or write event. The line of the trace file contains further a value that is equal to the delay of the internal events that were performed in the compute node before this read or write event occurred. This value is used to update the local logical clock of the compute node with the delay associated with the internal events that have been executed so far in the compute node.
Consider the case that the event logged on this line of the trace is a write event. The time-stamping mechanism creates an event object that contains a time-stamp equal to the local logical clock. This event object is then written to the connection using the `write` procedure of the connection. The event object is written as many times as there

are data-elements involved in the write event to the connection. The connection adds to the time-stamp of each of these Event objects the delay associated with the connection. The event objects are then added in-order to the list that implements the connection.
In the case that a read event is logged on this line of the trace, the time-stamping mechanism has to behave slightly different. It must first use the `read` procedure of the connection object to read as many event objects from the connection, as there are data-elements involved in the event. The time-stamp contained in the last event object must then be used to decide how the local logical clock must be updated. The compute node may have to add idle time to the local logical clock to respect the causality relations.

When all events are ordered, the event system of each compute nodes can be queried for the number of read and write events that have occurred in the compute node. The delay associated with these events and the internal events that have occurred in the compute node can also be obtained through the event system. The different time measures of the compute node can also be obtained using the event system of the compute node. The event system of the port objects contains values for the number of times a read or write was performed on the port. The event system of a connection object contains values for the number of data elements read and written to this connection. The event system contains now enough information to calculate the concurrency measures.

Computation Load

The computation load of the computational network is found by taking the average over the computation load of the compute nodes in the computational network. CAST computes therefore the computation load of all compute nodes, that are not primary nodes, in the computational network and takes the average over them. The compute nodes are found using the `computnodes` list. The values for the computation time and execution time can be obtained through the event system of the compute nodes.

Execution Load

The execution load of the computational network can be computed in the same way as the computation load. The only thing is that it requires the value of the longest run-time of all compute nodes in the computational network. This value can be found by traversing the list once and determining the maximum run-time of all compute nodes.

Restart

The restart measure is calculated in CAST by traversing all compute nodes in the `computenodes` list once and determining the largest run-time. This maximum run-time is then used to calculate the restart measure using Equation 4.14.

Synchronization

The synchronization measure is found using Equation 4.15. The required value for the sequential execution time is found by traversing the `computenodes` list and taking the sum of the execution times of all compute nodes. The total execution time of the computational network is calculated using Equation 3.2 and the values for the execution and idle-time of a compute node. The first compute node in the list can be used for that. It is then not necessary to find the largest local logical clock value of all compute nodes in the computational network.

Structure

To compute the structure measure of the concurrency model, CAST must first find all computational paths in the computational network. It will therefore first create a list of all paths that are in the computational network. This is done by starting in a compute node that is a primary node. Then, it finds all compute nodes that are immediately reachable from this primary node. A compute node n_j is immediately reachable from a compute node n_i if there is at least one connection from n_i to n_j and this connection is a so-called *data channel*. A connection is a data channel if the number of data elements transferred over the connection during the execution of the computational network is a certain fraction (default 10%) of the maximum number of data-elements communicated over a connection in the computational network. In this way, we try to exclude connections between nodes that are used to send control information (e.g. image size, number of color components) as these do not belong to the main data-streams in the network. The structure measure should focus on the main data stream, data channels, and not on the other connections. The communication fraction can be set in the CAST project file using a *param* element with *commfrac* as the value of the name attribute. For example, the element `<param name="commfrac" value="0.01"/>` sets the communication fraction to 0.01. This means that a connection is selected as a data channel if this connection communicates 1% of the data-elements communicated by the connection that communicates the most data-elements during the execution of the computational network.
For each compute node that can be reached from the primary node, CAST creates a partial path containing this compute node. After that, it is determined which compute nodes are immediately reachable from each of these compute nodes. Each compute node that can be immediately reached from one of these compute nodes is added to the path from that compute node. This process is continued for each path until a path cannot continue from a compute node because there are no immediately reachable compute nodes left taking into account the restriction in Definition 4.1 that a node can occur at most twice in a path. If the last compute node reached is a primary node, then the path is stored in the list of paths found in the computational network, else it is discarded. The discovery of new paths ends as soon as all paths from all primary compute nodes are found.

CAST has now created a list of all paths in the computational network that consist of data channels. The next step is to remove all paths that only go through primary nodes, as these paths do not belong to the computational network. After that, CAST creates a list of all computational paths that can be found in the computational network. This is done by taking a path from the list of all paths. It is then checked whether this path belongs to a computational path that is already in the list of computational paths. If so, the path is added to that computational path. If not, a new computational path is created with this path. If a new computational path is created, it is checked which of the already grouped paths belongs to this computational path. If a path does belong to this newly created computational path, it is added. This process is continued until all paths are grouped into on or more computational paths.

This gives the analyzer a list of all computational paths that are present in the computational network. To compute the structure measure, it must know how many computational paths go through each compute node. To compute this, it is then counted how many times each compute nodes occurs in different computational paths. This is used to calculate the average number of computational paths through a compute node. With that number and Equation 4.16, CAST calculates the structure measure.

Statistical Analysis

Assumption 4.2 says that the strings of data at the input ports of the computational network are infinite concatenations of the strings of data for one input – execution of the computational network. To get an idea of how the computational network behaves in this streaming environment it is important that the supplied strings of data represent the average input that the computational network will see. To support this, CAST has an option to calculate the concurrency measures over a set of simulations. The input of each simulation can be different, but the average over all inputs should represent the average input that the computational network will see.
The statistical analysis module of CAST is run when CAST is executed with the option: `-a-statistics`. It asks the user, which simulations should be analyzed. The program computes for these simulations the average, variance, minimum and maximum of all concurrency measures.
Note that this option can also be used to create averages over a computational network that is simulated with different delay settings for the channels or read and write events. This makes it possible to get an idea about the behavior of the computational network for different communication delays.

5.5 Conclusion

This chapter has presented an implementation for the model of computation introduced in Chapter 3 and the concurrency model of Chapter 4. The computational-network model is implemented using the YAPI/KPN model and C++ library developed by Philips Research. An advantage of reusing YAPI is that there is immediately a large set of code that can be reused for the computational-network model. The time-stamping mechanism in the model needs an implementation for the different delays functions in it. The delay function for events was implemented through two different delay function. One for the internal events, and one for the read and write events. The delay function for

the internal events uses the number of machine instructions needed to execute a given C++ statement to estimate the delay of the event. The delay function for the read and write events uses a linear function that estimates the costs, delay, associated with the communication. These costs are partially determined by the time needed to call the read or write function and partially by the time needed for the memory transfer. It is discusses in Section 5.3.2 that these costs depend on the chosen architecture and the chosen implementation for the communication primitives. The delay function for connections, which is also a part of the time-stamping mechanism, uses a function that estimates the delay based on the size of the data-element being communicated.
Section 5.4 presents a software implementation for the different delay functions and the concurrency model. It discusses how the YAPI/KPN models must be modified before using it with the Concurrency Analysis and Simulation Tool. This requires only minor changes to the original program. The parser, simulator and analyzer used in CAST are discussed, as well as the implementation of the different time-stamping functions. The last part of this section discusses the statistical analysis function of the program.
The next chapter presents a design exploration method that uses the different concurrency measures. Values for these measures can be determined using CAST, as will be shown in the design case of Chapter 7.

Chapter 6

Design Exploration Method

6.1 Introduction

This chapter describes the design exploration method. This method is used to transform a computational network that specifies the functional behavior of a computation to a computational network that specifies the computation and explicitly specifies the available concurrency. The aim of the design exploration method is to create a computational network that has a balanced execution load and communication behavior tailored to the system.
The method consists of five different steps that are explained in detail in the following sections. It starts with creating a computational network with as much task- and data-parallelism as possible. This is done by splitting the compute nodes. The communication behavior is then changed in the next step. In this step, the granularity of communication is tuned to get a more balanced communication load. The last two steps of the method consist of merging compute nodes. This removes some of the task- and data-parallelism, but it balances the execution load of the compute nodes in the computational network.
Four of the five step of the method optimize one of the five concurrency measures. The detailed measures that come with this measure are used to derive this optimum. The other concurrency measures are used to prevent the optimization of the computational network for only the concurrency property optimized in that step. The design exploration method arrives in this way at a point where the computational network as a whole is optimal considering the different concurrency properties.

6.2 Starting Point

The design exploration method starts with a computational network that functionally specifies the computation. This computational network will contain some parallelism, as it will contain different compute nodes for the coarse-grain tasks carried out in the computation. It may also contain some data-parallelism that is already specified by the designer. The exact level of granularity with respect to the specified concurrency is not relevant for the method, as it will first split the computational network in a version that contains as much task- and data-parallelism as possible. Only then will it start recombining the compute nodes in the computational network. The more parallelism already specified, the less must be extracted in the first two steps of the method.

6.3 Task Splitting

In the first transformation, called task splitting, we increase the task-parallelism in the computational network. This is done by splitting the slowest compute nodes – i.e., the longest running tasks – in a number of compute nodes that are grouped in a single computational path. In terms of microprocessor design, this is equal to creating a pipeline for the slowest operations/tasks. To select possible candidates for the task splitting, we consider the compute nodes that have a low value for the restart measure. The most important one to consider in this case is the compute node with the lowest value for the restart measure, as this compute node is limiting the throughput of the computational network. The effectiveness of this transformation is measured by the restart measure for the computational network.

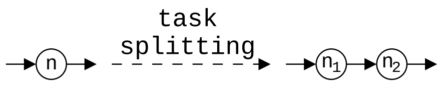


Figure 6.1: Task splitting.

6.4 Data Splitting

The result of the previous step is a computational network that exploits the task-parallelism that is present in the application. The computation that is realized using this computational network may contain also data-parallelism. Section 4.3.5 discussed that this data-parallelism can be made explicit by creating different computational paths for the strings of data that can be transformed in parallel.
The data-parallelism of every compute node must be made explicit by introducing separate compute nodes for the strings of data that can be transformed in parallel. That is done by splitting every compute node which contains data-parallelism. A split compute node is replaced by a set of compute node which perform in parallel the same transformation on different strings.

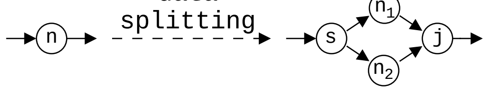


Figure 6.2: Data splitting.

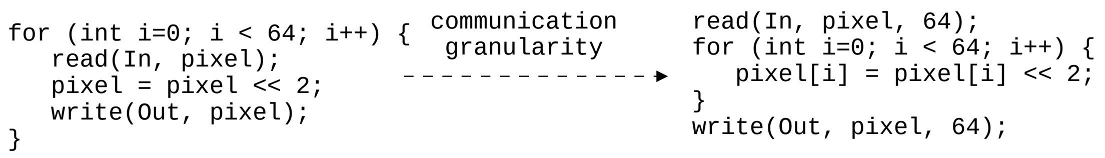
Figure 6.2 shows a compute node n that is transformed in the data splitting step. It

results in two compute nodes n_1 and n_2. These perform the same transformation on the strings of data as the compute node n. The compute node s, *split node*, writes the strings of data originally intended for compute node n now partially to compute node n_1 and partially to compute node n_2. This split node must control to which connection it will write the data. The transformation has also introduced a compute node j. This compute node, called *join node*, is responsible for synchronizing the strings of data output by the compute nodes n_1 and n_2. The join node must write to its output the same string of data as the compute node n did before the data splitting transformation.
The split will - in a later step of the method - often be merged with the compute node that is before it in the computational path. This compute node will as a result often require more run-time. It must control to which connection it outputs the strings of data. The same situation will hold for the join node. This node will often be merged with the compute node that follows it on the computational path. This compute node will also often require more run-time as it must synchronize the strings of data that are communicated over the different connections.

The goal of the data splitting step is to extract the data-parallelism from the application. The structure measure can be used to determine the amount of data-parallelism in a given computational network. The closer the measure comes to one, the more data-parallelism is present.
All compute nodes in the computational path must be considered during the data splitting step. It is namely important that the new computational paths that are created in this step share as less compute nodes as possible. The system designer must concentrate on the compute nodes through which a high number of computational paths go, as these points require synchronization of the different strings of data in the computational network. This synchronization takes time and may result in communication bottlenecks. As a result, these nodes do not use the possibly available data-parallelism to the same extent as the compute nodes that are on fewer computational paths.

6.5 Communication Granularity

The third transformation in the design exploration method is the communication granularity step. The granularity of communication is tuned in this step to get a more balanced communication load. This is done by decreasing the number of read and write calls of a compute node. This requires that the number of data elements communicated per call is increased.
This transformation is based on the observation that read and write calls are expensive in software [18]. The exact costs for the read and write calls are determined by the system architecture and communication primitives used (See Section 5.3.2). These numbers may be unknown when the communication granularity step is performed. One can then associate a delay with the read and write calls that expresses that these operations are more expensive than a normal memory operation or function call and perform statistical analysis for different delays.
The effect of communicating larger blocks of data is measured by the computation load.



```
for (int i=0; i < 64; i++) {
   read(In, pixel);
   pixel = pixel << 2;
   write(Out, pixel);
}
```

```
read(In, pixel, 64);
for (int i=0; i < 64; i++) {
   pixel[i] = pixel[i] << 2;
}
write(Out, pixel, 64);
```

Figure 6.3: Communication Granularity.

This measure must increase to a point where the compute nodes are mainly busy with the actual transformation and not with communication. The candidates for the communication granularity step are selected by selecting the compute nodes that have a low computation load, as these compute nodes spend most of their time on communication and not on computation.

6.6 Data Merging

The data merging step is the next transformation in the design exploration method. The aim of this step is to get a collection of computational paths that all have to do the same amount of work. This is realized by creating a computational network in which the compute nodes of every computational path require the same amount of time to perform the computation. Meeting this requirement will result in a raised execution load. It will ideally also lead to a situation in which the execution of the compute nodes of different computational paths before a join node takes the same amount of time. This leads to a situation in which the join node does not have to wait long before the different datastreams are synchronized. To get a better idea whether this situation holds, one should calculate the average execution load for every computational path individually. This is at the moment not supported by CAST, but the detailed measures output by CAST allow one to do so.
The transformation performed in the data merging step is the inverse of the transformation in the data splitting step. It usually only involves different combinations of compute nodes than the onces created in the data splitting step.

6.7 Task Merging

The last step in the design exploration method is called task merging. The goal of the transformations performed in this step is to create a computational network in which all compute nodes are all executing during as much time of the total execution time as possible. Every compute node should be busy during most of the computation. To what extent this is realized is indicated by the execution load of the computational network.
Merge candidates for the task merging step are the compute node that are neighbors of each other on the computational path and that together have an execution load of less than one, or marginally higher than one if no other options are available. The task merging step merges the compute nodes along a computational path and reduced in this

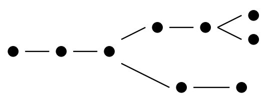


Figure 6.4: A design tree resulting from a design exploration.

way the available task-parallelism, but raises the execution load of the computational network. The operation looks like the inverse of the task splitting step.

6.8 Conclusion

This chapter has presented a design exploration method that aims at creating a computational network that has a balanced execution load and communication behavior tailored to the system. This is done using a five step approach. The task splitting, data splitting, communication granularity and task merging step aim at respectively optimizing the restart, structure, computation load and execution load measure of the concurrency model. Each of these steps requires both the main and detailed measures of the concurrency model. The data merging step is the only measure that not aims at optimizing one of the five main concurrency measures.

The design exploration method starts with a computational network that functionally specifies the computation. This design is transformed in the different steps of the design exploration method. In most of these steps, one design is used as a starting point for one new design. This new design is then used as the starting point for the next design. It is however possible that a design serves as the starting point for multiple designs. For instance in a situation in which the execution load can be optimized by either removing or preserving all data-parallelism after the communication granularity step. To explore both situations, two different designs must be made, one for each situation. These designs use the final design created in the communication granularity step as a starting point. The design exploration method results then in a design tree that relates all design made during the exploration. Figure 6.4 shows an example of such a design tree. The figure shows the different designs (dots) made in a design exploration and their relation. Branches in the design tree can be made everywhere during the design exploration. There are however some point at which it is very likely that a branch is created. The final design of the communication granularity step is one that is very likely to serve as a branch point. The reason for this is that when a design contains both task- and data-parallelism there will probably be different solutions to remove some of the parallelism and get a good execution load. A second branching point is the final design created in the data splitting step. This design contains much parallelism. The costs of communication will determine to what extent this parallelism can be preserved, as it determines the communication overhead. The communication granularity step changes the communication overhead by reducing the parallelism. The balance between this communication overhead and the parallelism is determined by the costs of communication. Different communication costs may therefore lead to different designs. The starting point of the design exploration can serve as a third branching point. A system architecture that contains processors with an instruction set for specific task will have different restart measures for some nodes than an architecture that does not have these processors. This situation can lead to different decisions in the task splitting step and with that to different designs.

The design exploration method is used in the design case presented in the next chapter. The design exploration in the case study gives a design tree as described in this conclusion.

Chapter 7

Experimental Results

7.1 Introduction

This chapter presents a case study that uses the design exploration method. The case study derives in a structured way an implementation of the JPEG decoder [12] that has a balanced workload and good communication behavior. This structured approach results in two computational networks that implement the JPEG decoder as a computational network. The development is done independent of any architecture.
We mapped subsequently these computational networks on a multi-processor architecture to demonstrate that the presented approach works. For this we used the CAKE architecture and CAKE simulator [34]. We compare between the results using our approach and the results obtained in a case study performed at Philips Research [18], where the JPEG decoder was manually mapped onto the CAKE architecture.

7.2 Case Study: JPEG Decoder

7.2.1 Introduction

The JPEG decoder has been selected as a case study for two reasons. First, this application is used as a case study in the Dataflow Group of the Open SystemC Initiative [36]. Their aim is to standardize the modeling of process networks. Process networks and computational networks are closely related subjects that aim at making concurrency explicit. It seems therefore logical to use the same example as this important research group does. Second, this application is used in a case study of Philips Research [18]. The results and sources of the various designs created in that case study are available at the university. This makes it possible to compare the results of both case studies by analyzing both of them with the concurrency model. In the case study of Philips, the designs were mapped onto the CAKE architecture [34]. This environment is also available at the university. This makes it possible to also compare the results of both case studies on a multi-processor platform. The possibilities to compare the results of the two case studies make the JPEG decoder an interesting option to choose.

JPEG images are stored and processed either with or without data inter-leaving. When there is no data inter-leaving, each component is stored and processed separately - a RGB image would be stored as three separate images, one for red, one for green and one for blue. For more efficient storage and processing, the color components can be interleaved. Inter-leaving means that the data for the three color components is not stored as three separate images in the coded image, but they are placed in parts after each other in the coded image.

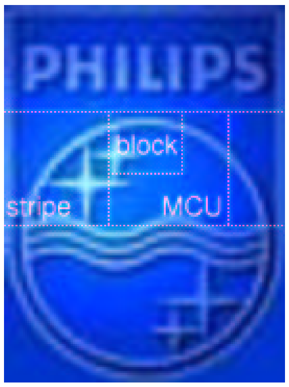


Figure 7.1: Hierarchical image view.

An image consists of stripes, see Figure 7.1. A stripe consists of MCU's. An MCU consists of (sub-sampled) chrominance and luminance blocks and a block consists of 8 by 8 values. Each color component is partitioned into these rectangular blocks of 8 by 8 values. If one data block is selected from each of the color components, then they form a minimum coded unit (MCU). That is the smallest group of interleaved data that completely describes a region of the image.

Figure 7.2 shows a block diagram of a JPEG decoder. The input of the JPEG decoder is a byte stream connected to the DMX block. The DMX block de-multiplexes the byte stream into the tables required for the variable-length decoding, tables required for the dequantization and the bytes that must be parsed by the decoder. The variable-length decoder (VLD) decodes the run length and Huffman encoded minimum coding units using the Huffman tables that were demultiplexed by the DMX block. The VLD outputs decoded pixel blocks, which are dequantized by the inverse quantization (IQ) block. The IQ block uses for that the quantization factors extracted from the input byte stream by the DMX block. The blocks then undergo inverse zigzag (IZZ), and two-dimensional inverse discrete cosine transformations (IDCT). The YUV2RGB block converts the blocks to stripes, applies vertical and horizontal scan rate conversion and color conversion from YUV to RGB.

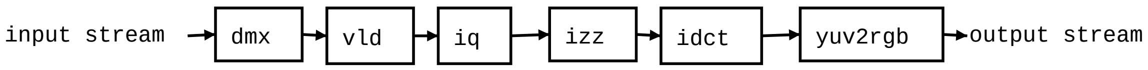


Figure 7.2: Block diagram of a JPEG decoder.

It is of no relevance to know the exact details of the different blocks of a JPEG decoder. The idea is that the concurrency measures indicate which compute nodes must be transformed and that only those nodes are considered. This should require only basic knowledge of how a JPEG decoder operates. For more details of JPEG, the book Image and Video Compression Standards [4] is recommended.

7.2.2 Design Exploration

A good starting point for the design exploration of the JPEG decoder would be a computational network with a compute node for each of the blocks shown in Figure 7.2. The JPEG decoder that is used as a starting point in the Philips case study implements a number of these blocks as compute nodes, but other blocks have been split in a collection of compute nodes. The computational network used in the Philips case study was chosen to be the starting point of the case study. The advantage is that we do not have to implement a JPEG decoder from scratch. Furthermore, the comparison of the end result of both case studies is fairer when the same code is taken as a starting point.
The computational network of this JPEG decoder is shown in Figure 7.3 and is referred to as the reference decoder or design 0. The figure shows that the IDCT, YUV2RGB and DMX block of the block diagram have been split in the computational network into several compute nodes. The computational network contains further a frontend and backend compute node. These compute nodes are connected to the input and output ports of the computational network. They read a JPEG image from file and write the image to an output file. The figure shows both the connections needed for the data-streams as the connections needed for control information. Compute nodes that are needed to copy the data output by one node to several connections, *forks*, are not shown as explicit nodes. These are simplified as black dots.

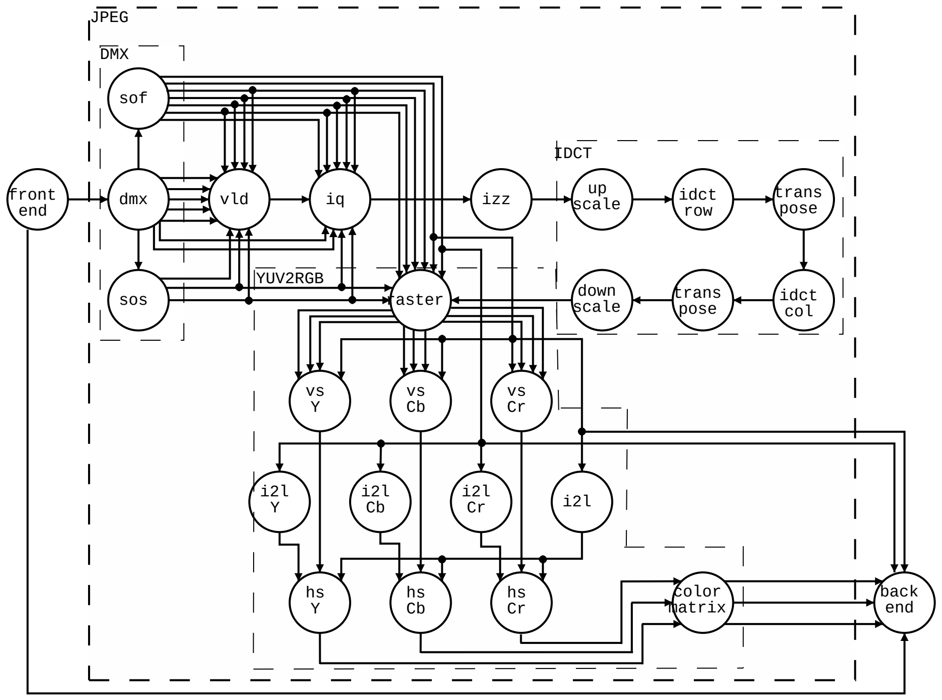


Figure 7.3: Reference JPEG decoder (Design 0).

The concurrency model is based on the analysis of simulation runs. The input given

Table 7.1: Characteristics of simulation data.

Image	Size (pixels)	#Color components
philips	50x67	3
safari	580x400	3
bbq	600x398	3
intro	640x480	3
shuttle	669x1004	3

to the computational network over these simulations should represent the average input pattern that can be expected by the computational network. To create an input that meets this requirement, we simulated all designs created in this case study with five different images. All images have three color components, but are different in size. The characteristics of the images are shown in Table 7.1. The main concurrency measures were calculated using the statistical analysis option of CAST.

The design exploration method presented in the previous chapter is used in this case study. This method describes a number of steps that must be performed in-order to get computational network with a balanced execution load. The steps have strict separations between the type of transformations performed in the different steps. In practice one might want to deviate from that, as is shown in the case study. The design exploration method is therefore used only as a coarse guideline.

Task Splitting

The first step in the design exploration method is the task splitting step as described in Section 6.3. The idea is to split the compute nodes that have the smallest restart measure. Table 7.2 shows the compute nodes with the lowest restart measure. The restart measures are scaled as described in Example 4.4 with the maximum restart measure found in this design. As this is the only design available at this moment of the design exploration, it is not possible to normalize using more designs. The compute nodes listed in this table are our first candidates in the task splitting step.

In the design 1 up to 4, we transform these compute nodes. The order in which the

Table 7.2: Restart measure for design 0.

Compute Node	
vld	4.0e-5
matrix	5.6e-4
idctcol	1.0e-4
idctrow	1.0e-4

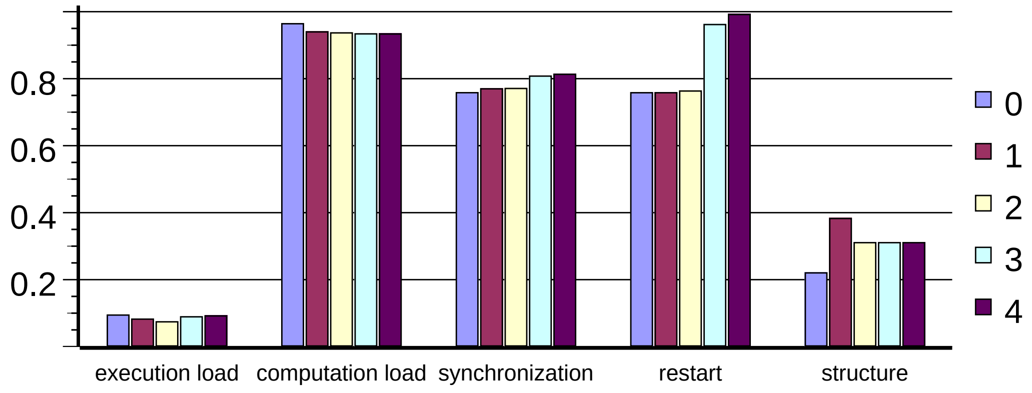


Figure 7.4: Concurrency measures for task splitting step.

nodes are transformed is chosen arbitrarily. We started with splitting the matrix node into a network (design 1). This node was chosen because it was the first one for which we found a way to split it. The matrix compute node is split in a network that computes the color conversion of the three color components in parallel, instead of doing it all in one node. The idctcol and idctrow compute nodes are the next compute nodes to be split into a computational network. These compute nodes realize the two 1-dimensional idct transformation required in the JPEG decoder. For this, the Loeffler algorithm is used [24]. This algorithm describes how to compute a 1-dimensional idct in four separate steps. Following the algorithm, it is possible to subdivide the two 1-dimensional idct compute nodes in a computational network. The computational networks contain a compute nodes for each of the four steps. This transformation result in design 2 of the JPEG decoder.

Figure 7.4 shows the values of the concurrency measures for design 0, 1 and 2 of the JPEG decoder, as well as for some other designs discussed further-on. The goal of the transformation from design 0 to 2 was to get a better restart measure. The figure shows that this has not been realized. The restart measure has not changed. This is not surprising as the vld node is the bottleneck. To get a better restart measure, we have to concentrate on the variable-length decoder. The designs 3 and 4 concentrate on the vld compute node, which implements the decoder. The problem is that the variable-length decoder must go sequentially through the byte stream that it reads from the input. This makes it very hard to split the node. There are however some improvements that can be made on the code and that lead to a better restart measure. The transformation, performed in the designs 3 and 4, do not require machine-dependent code changes. They only use some basic properties of the JPEG decoder, such as the fact that the decoder in most situations needs a single bit from a byte. Figure 7.4 shows that these transformations in dead lead to a better restart and synchronization measure. The changes made to the reference design in design 1 and 2 have not improved the restart measure, but they did improve the structure measure. The execution load stays almost constant during these transformations. As the vld node cannot be split in more nodes, we end the task splitting with design 4. This design is shown in Figure 7.5.

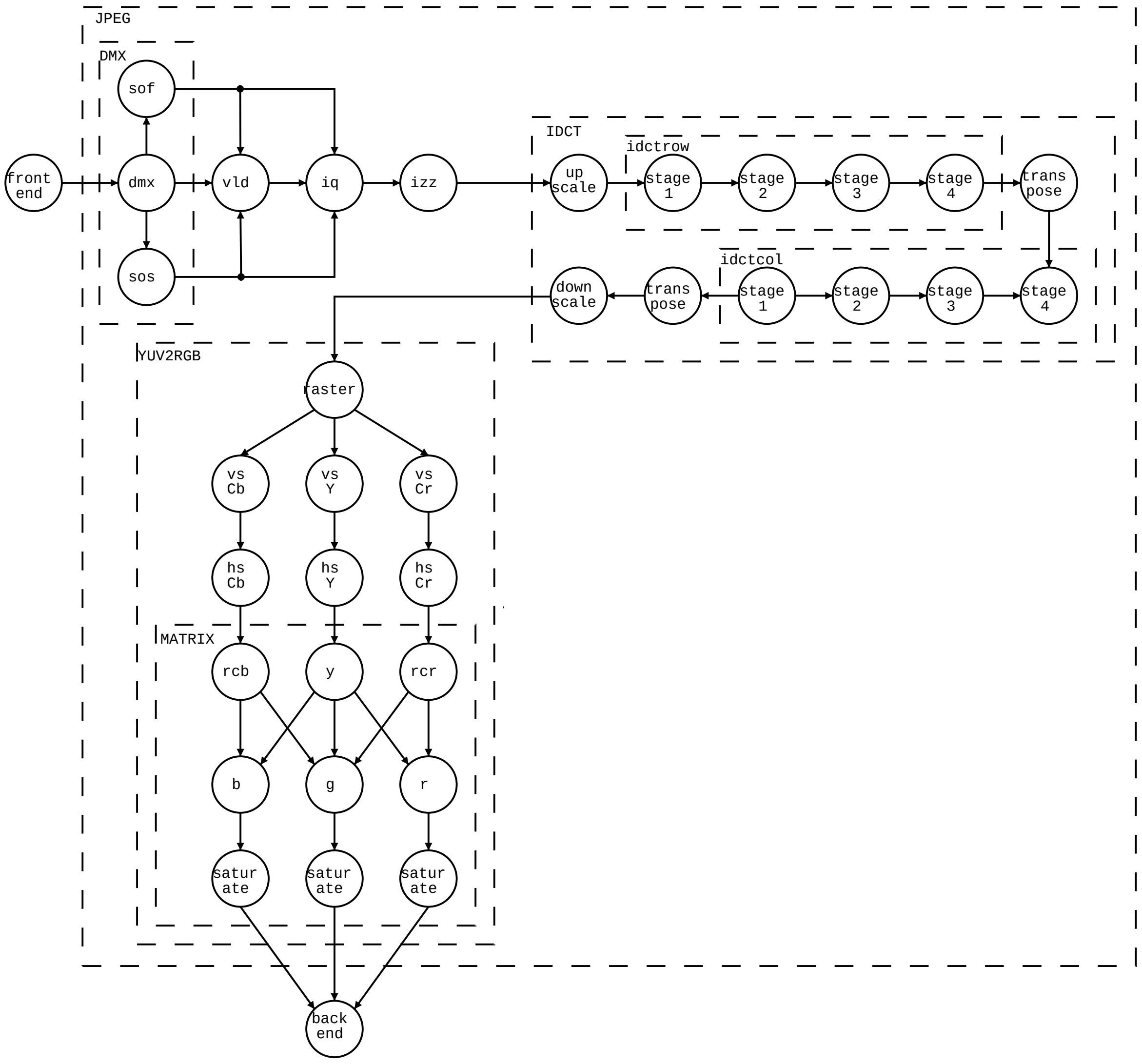


Figure 7.5: JPEG decoder (Design 4).

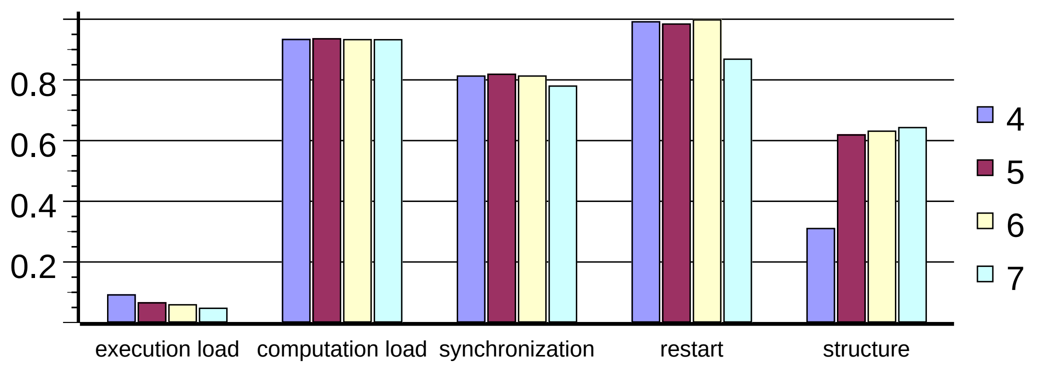


Figure 7.6: Concurrency measures for data splitting step.

Data Splitting

The next step in the design exploration method is data splitting. The compute nodes are considered in order of the number of computational paths that go through them. The vld, iq, izz and raster compute nodes and all compute nodes of the sub-network that implements the IDCT are part of 7 computational paths. This means that they are part of all computational paths in the computational network. These compute nodes must therefore be considered first in the data splitting step.

The vld compute node cannot be split. The izz and idct compute nodes are considered next. Copies of them can be created for each of the color components. This requires that a join and a split node is added to the computational network. The join node is integrated into the iq node. The split node is integrated into the raster compute node. The iq compute node must decide to which color component the transformed data belongs and send the data to the correct output port. The raster compute node must decide from which color component it needs data and read it from the correct input port. These transformations in the computational network lead to design 5. The values of the concurrency measure for this design are shown in Figure 7.6. This figure contains also the concurrency measures for design 4, which is the starting point of the data splitting step. The figure shows that the structure measure has improved by the applied transformation. The transformation meets the goal of the data splitting step.

The raster compute node has input ports for the three color components. These have connections with the output ports of the idct computational networks of the three color components. The raster compute node has also three output ports for the three color components. It seems therefore logical to consider the option to create three separate raster compute nodes. One for each color component. The transformation performed by the raster compute node allows this. The resulting JPEG decoder is design 6. The structure measure shows again some improvement compared to the previous design.

The next compute node considered is the iq compute node. The transformation carried out by this compute node makes it possible to create three instances of it. Each of the instances transforms the data for one color component. Again there is some improvement in the structure measure. But there is also a considerable decrease in the restart measure. The synchronization measure shows also a decrease. This is caused by the variable length decoder. To make three separate iq compute nodes, we had to insert a split node before these nodes. This split node is integrated in the vld compute node. This node is as a result slower than in the previous design. The decrease in the value of these two concurrency measures form an indication that the data splitting step of the design exploration method should not be continued. Strictly following the rules of the design exploration, we should choose design 6 and not design 7. However, we decide to continue with design 7 in the next step of the design exploration. The reason for this is that we have the idea that design 7 will produce better results in the task and data merging steps than design 6. This is a case in which the design exploration method is used only as a coarse guideline. The resulting computational network, design 7, is shown in Figure 7.7.

Communication Granularity

The communication granularity is the next step of the design exploration method. The granularity of the communication used in the final system will depend strongly on the costs of the communication. The reasons for this and implications have been discussed when the delay function for read and write events was introduced. In the first two steps of the design exploration method, we did not consider the costs of communication. This can no longer be avoided when we focus on the granularity of communication.
To do this, we must choose values for the a and b constant of the delay function for read and write events (see Section 5.3.2). At this point there is nothing known about the communication structure that is used in the target architecture. This makes it hard to estimate values for these constants. To overcome this problem, we assume that it is possible to use semaphores in the communication primitives of the target architecture. This implies that the b constant is 0. If the assumption is not valid, then it is most likely that the communication primitives used allow fast transfer of data over the physical communication medium, as a systems architect will try to provide efficient communication primitives. This results in a small b constant and with that in a small error in the estimated delay used in our analysis. This will probably not lead to decisions in the design exploration that have a large impact on the final result.
We still have to estimate a value for the a constant. For that we use the observation that calling a function that implements the communication primitives is more expensive than a normal memory operation. As an estimation of these costs we take a value of 30 logical clock values. The a constant is thus equal to 30. This estimation is based on the observation that a normal function call in software takes about 5 to 10 instructions. To implement the function another 20 instructions will most certainly be needed to send and receive the semaphores. To verify these numbers, we have performed a number of simulations with design 7 and different values for the constants in the delay function. These simulations were analyzed using the statistical analysis option of CAST. They showed that our assumption is valid.

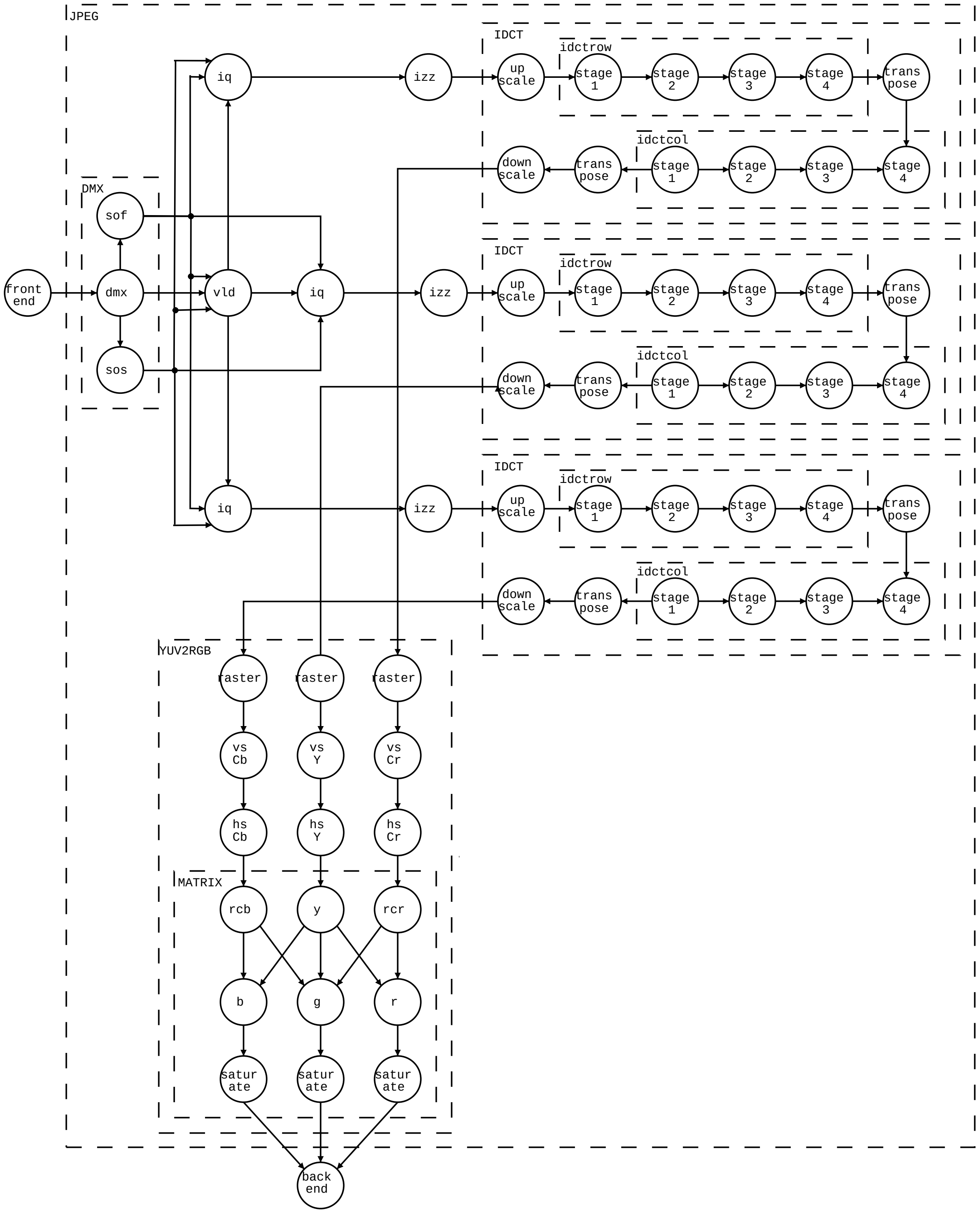


Figure 7.7: JPEG decoder (Design 7).

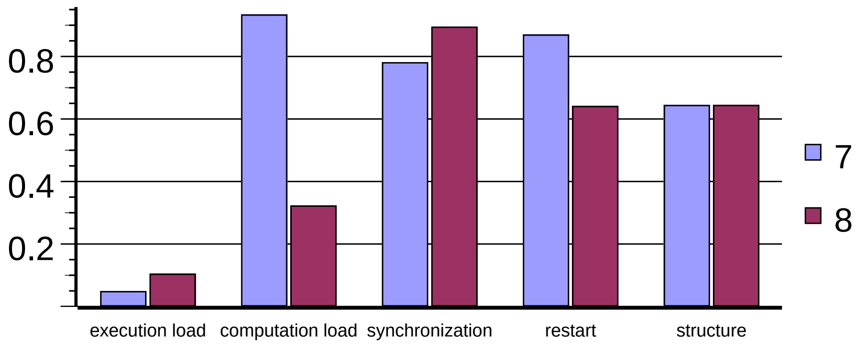


Figure 7.8: Cost of communication.

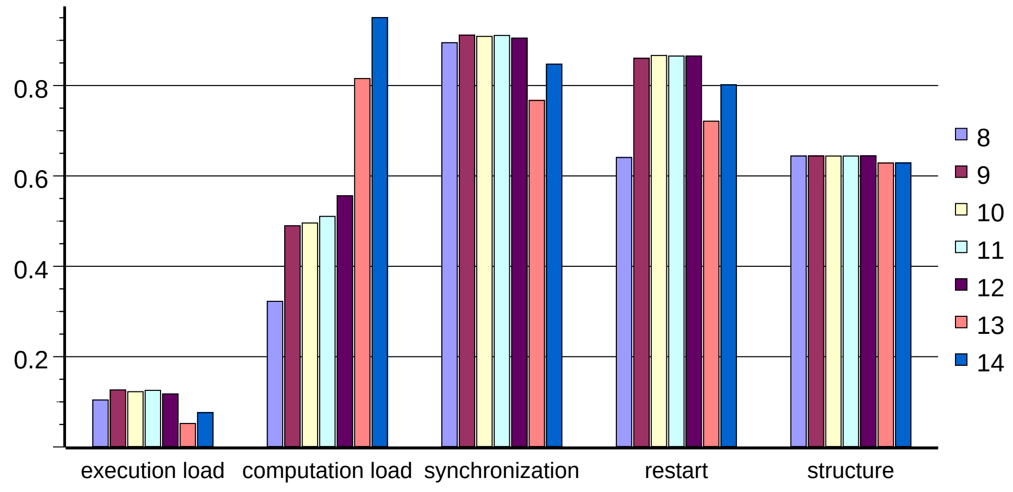


Figure 7.9: Concurrency measures for communication granularity step.

The impact of the costs of communication on the systems performance can be seen if we look at Figure 7.8. This figures shows the values for the concurrency measures of design 7 and 8. The only difference between these two designs is that design 7 assumes that a read or write event takes one logical clock value, while design 8 assumes that it takes 30 logical clock values. The computation load has dropped significantly in the latter case. This indicates that with the new costs of communication, the compute nodes are most of their time busy with communication and not with computation. This effect must be reversed when the granularity of communication is optimized.

To do this, we create a set of designs (designs 9 to 14) in which gradually more compute nodes communicate larger blocks of data in one time. The effect of these transformations on the concurrency measures is shown in Figure 7.9.

The figure shows that up to design 12 all concurrency measures except the computation load do not really change. Only the computation load goes up, as is intended with these transformations. Design 13 shows a sudden decrease in the restart and synchronization measure. This is caused by the newly created matrix compute node. This node replaces the computational network created in design 1. It removes the changes in the computa-

tional network introduced in the transformation from design 0 to design 1. The matrix compute node communicates on a pixel-by-pixel basis. The costs of communication make this so expensive, that the matrix node becomes the slowest node in the computational network. There are two solutions to overcome this. First, we could decide not to re-merge the computational network. In that case it is not possible to improve the granularity of communication for the nodes in this computational network. Second, we could try to improve the measures by increasing the granularity of communication in the matrix node. The design exploration method is at the moment in the communication granularity step. Therefore we choose the second solution. This approach is used in design 14. The matrix compute node communicates on a line-by-line basis. The concurrency measures show that this compensates to a large extent for the problems introduced in design 13 and further increased the computation load.
The transformation performed to get deswign 13 show again that the design exploration method is only used as a coarse guideline.

Two solutions

When the detailed measures of the concurrency model are studied, it becomes clear that the compute nodes on the computational paths for the two chrominance values have 25% of the execution load of the compute nodes on the computational path that processes the luminance values. This unbalanced execution load must be removed in the task and data merging steps. There are three solutions to do this. First, we could remove all data parallelism and then apply task merging to get a balanced execution load for the compute nodes (solution 1). Second, we keep all data parallelism and apply only task merging to get a balanced execution load (solution 2). Third, the two chrominance paths are combined in the data merging step and the execution load is balanced in the task merging step. The first two solutions are explored in this case study. The remainder of this section discusses how these solutions are derived. The third solution is not explored because of the limited time available for this project.

Solution 1: data merging

We remove in solution 1 the data-parallelism that is present in the computational network. This is realized in design 17. To arrive at this point, we made a number of intermediate designs (15 and 16). These designs demonstrate that the transformation does not lead to a bad solution. Design 15 does not have separated paths for the iq, izz and raster compute nodes and the idct computational network. In design 16, we combined the horizontal and vertical scan-rate conversion compute nodes. This gives a vhs compute node for each of the color components. These three vhs compute nodes are merged in design 17. This transformation removes all data-parallelism from the computational network. This can be seen from the values of the concurrency measures shown in Figure 7.11. The value for the structure measure is zero, meaning that there is data-parallelism in the structure. The computation load, synchronization and restart measure are almost constant over these design. This means that the concurrency properties measured by these measures do not really change. The execution load has doubled but is still relatively low. There is no data-parallelism left that can be removed. This ends the data merging step of the design exploration method.

Solution 1: task merging

The final step in the design exploration method is the task merging step. Compute nodes are merged in this step to get an even further balanced execution load. This process gives the designs 18 until 25. In each design a few compute nodes are selected that together have an execution load of less then 1. These compute nodes are merged into a new compute node.
We start with merging the four stages of the one dimensional idct computational network in one compute node (design 18). The upscale and transform compute nodes of the idct computational network are merged in design 19 into the idct compute nodes. Design 20 combines the dmx, sos and sof compute nodes. The resulting computational network has now a large resemblance with the block diagram of Figure 7.2.
In design 21, we combine the iq and izz compute node. The resulting compute node is combined with the idctrow compute node (design 22). The task merging step continues in design 23 with the merge of the dmx and vld compute nodes in a new jfif compute node. This transformation may seem strange, as the vld compute node is the longest executing node in the JPEG decoder. The task merging step is supposed to not merge this node with another node. However, the execution load of the dmx node is small. To raise the execution load of the computational network, we must merge the dmx node with another node. The dmx node is mainly busy with communication to the vld node, it seems therefore reasonable to merge these two nodes. Furthermore, this merge reduces the communication overhead. Figure 7.11 shows an increase of both the execution load and computation load for design 23 compared to design 22. In design 24 and 25, we merge the fork compute nodes in the other compute nodes. These fork compute nodes are responsible for copying data that they receive over their input port to all of their output ports. These nodes are mainly idle. If they are executing, the spend most of their time on communication and not on computation. The information communicated is typically control information (e.g., size of image or number of color components). These nodes are therefore characterized by a very low execution load. Design 25 is depicted in Figure 7.10.
The result of all of these transformations is shown in Figure 7.11. The figure shows that the computation load, synchronization and restart measure have not really changed during these transformations. The execution load has gained much compared to the starting point of the task merging step (design 17).

To get an overview of the result of this design exploration, we must look at Figure 7.12. It shows that the execution load of our final design is much better than that of the reference design. The computation load, synchronization and restart measure seems not

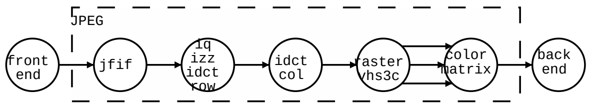


Figure 7.10: JPEG decoder (Design 25).

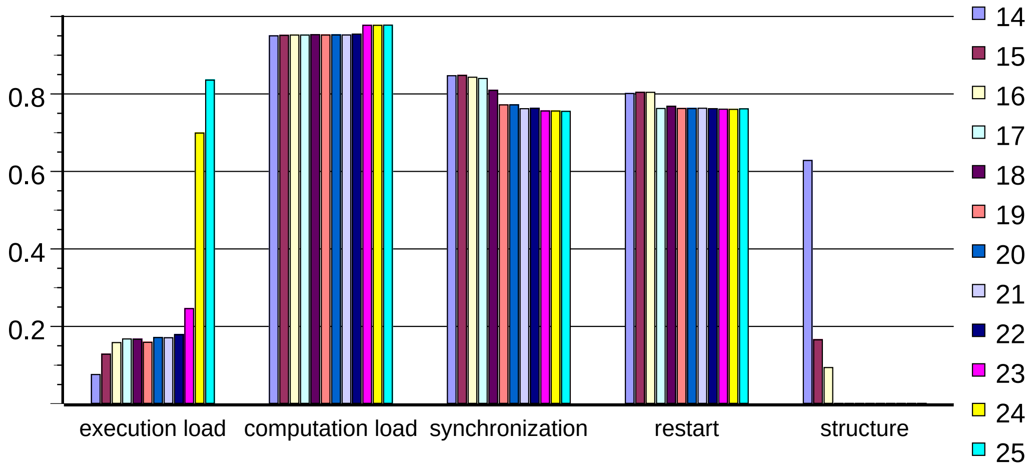


Figure 7.11: Concurrency measures for solution 1.

to have changed during the design exploration. Remember however that the costs of communication are different in design 0 and design 25. The drop in these measures when the costs of communication were introduced (design 8) has completely been compensated in design 25. The only measure that is really down is the structure measure. To raise the execution load, we had to give up on the data-parallelism in the structure. Overall, we can say that the computational network of design 25 implements a JPEG decoder that has better concurrency properties than the reference design. The compute nodes have a balanced workload load and good concurrency properties – most concurrency measures close to one.

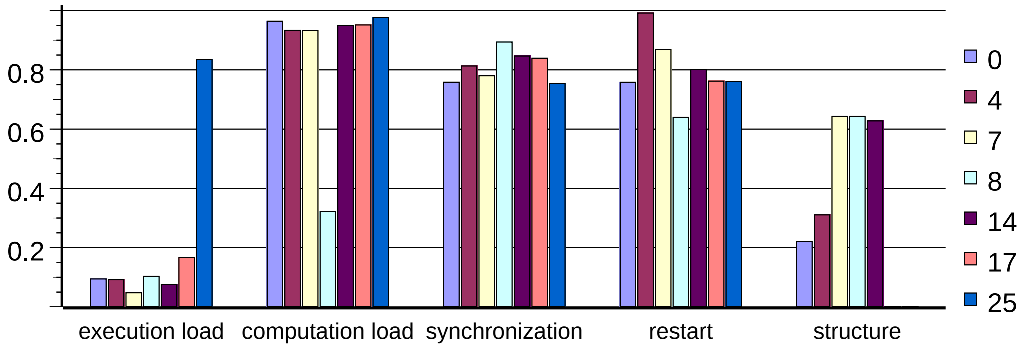


Figure 7.12: Concurrency measures for solution 1.

Solution 2: task merging

In the second solution, we do not change the data-parallelism that was found during the data splitting step. This solution skips therefore the data merging step of the design exploration method. It goes directly from the communication granularity step to the task merging step. Design 14 is the starting point for this. The solution is derived in designs 26 until 31. The same method as used in solution 1 is used here to arrive at the final design. Therefore, we do not discuss all of these transformations here. The concurrency measures for the different designs are shown in Figure 7.14. The structure measure shows that most of the data-parallelism is preserved, as its value does not change much from design 14 to design 31. The restart, synchronization and computation load do also not change very much. This means that these concurrency properties are not changed. The execution load goes up by almost a factor of 6. This cannot be further improved as the jfif node is the bottleneck. The computational network that implements the JPEG decoder in design 31 has a more balanced workload than in design 14. The final design, design 31, is shown in Figure 7.13.

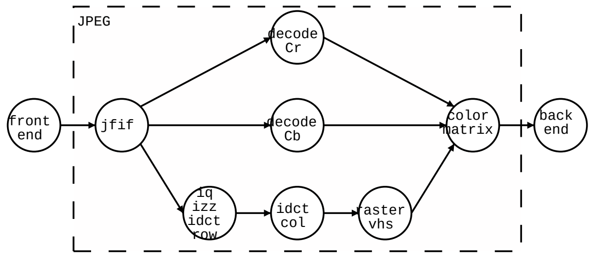


Figure 7.13: JPEG decoder (Design 31).

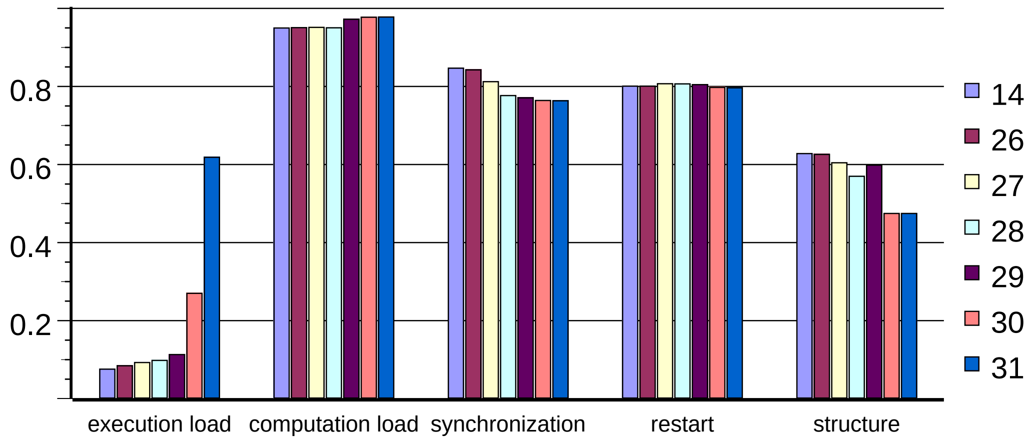


Figure 7.14: Concurrency measures for solution 2.

The complete overview of the design trajectory used to get from the reference design to solution 2 is shown in Figure 7.15. The figure shows that over each step in the design trajectory only one measure changes. In the end, all measures are optimized. The difference

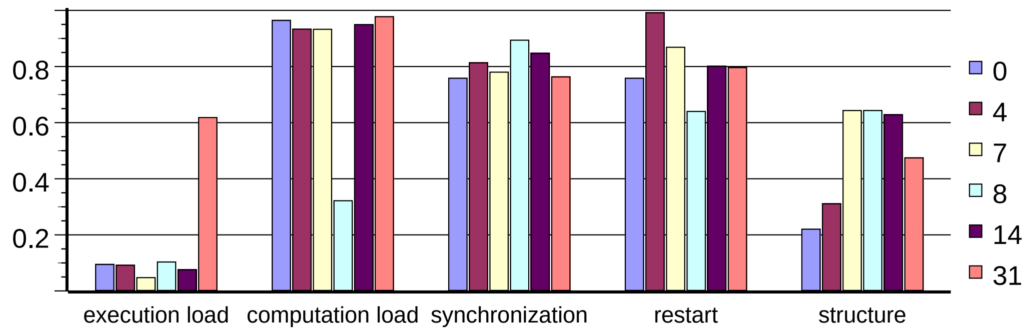


Figure 7.15: Concurrency measures for solution 2.

in the execution load between the reference design (design 0) and our final design (design 31) is clearly indicated in the figure. The computation load, synchronization and restart measure did not change when these designs are compared. It seems that the concurrency properties measured by them are similar for these two design. This is however not true as the cost of communication in design 0 and design 31 is different. These costs are 30 times higher for design 31 than in design 0. The design exploration method has helped us in finding transformations to the network that compensate for this. The structure measure has also improved during the design exploration. From these observations, we conclude that design 31 has better concurrency properties than the reference design. Design 31 is a computational network with a balanced workload and it does not give up on the other concurrency properties.

We now have two solutions that both implement the JPEG decoder in a computational network. To compare these, we have to revert to Figure 7.16. It shows that the computation load and synchronization of both solutions are similar. The second solution has a slightly higher restart measure than the first solution. This indicates that solution 2 needs less time between the input of two images that are decoded than solution 1 does, i.e., solution 2 has a higher througput than solution 1. This comes however with a price, as can be seen in the execution load. The execution load of this solution is not so good as that of solution 1. The choice between these design will depend on the architecture of the system on which it is mapped. Solution 2 might require more processors than solution 1, but it has a higher throughput and more data-parallelism in the structure. This data-parallelism can proof useful if there are in the physical communication medium options to have communications in parallel.

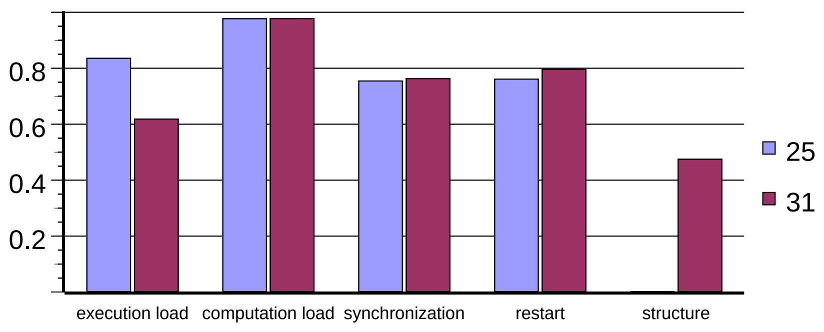


Figure 7.16: Comparison of solutions.

7.2.3 The Effect of the Compiler Used

The delay function for internal events assumed that the instruction set of the processors in the implemented system has no large impact on the concurrency. The only constraint is that the processors may not have an application-specific instruction set. CAST used in the case study a GCC i686 compiler. To verify the assumption, we must redo all experiments with a compiler for a different processor. For this, we assume that the target architecture consist entirely of MIPS processors. They have no optimized instruction set for a given application or application domain, but have a different instruction set than the i686 processor.

All designs created during the case study were analyzed again for their concurrency properties using CAST. The only difference with the analysis during the design case is that CAST no longer used the GCC i686 compiler, but a MIPS cross-compiler. The values of the concurrency measures of the different design steps are shown in Figure 7.17 for solution 1 and in Figure 7.18 for solution 2. There is only one difference compared to the results found during the design exploration. The execution load of solution 1 has gone down and is now similar to with the execution load of solution 2. This demonstrates that the instruction set has an impact on the result. However, solution 1, design 25, is still not a bad solution if concurrency is concerned. From this we conclude that even if we do not know the instruction set of the processors in the target architecture, we still get good results from the design exploration.

We now look in more detail to the values of the concurrency measures found for the various designs of solution 1 (see Figure 7.19). This figure shows that not design 25 but design 24 is optimal. This design has a higher execution load, synchronization and restart measure. This shows again that the architecture has an influence on the measures and the solution. This effect can however be found by simply re-simulating the created designs when more information on the target architecture becomes available.

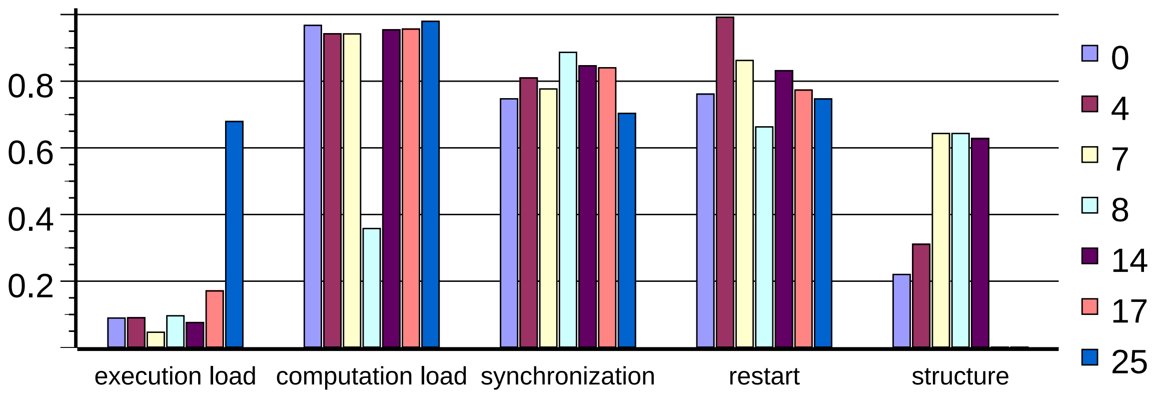


Figure 7.17: Concurrency measures for MIPS architecture (solution 1).

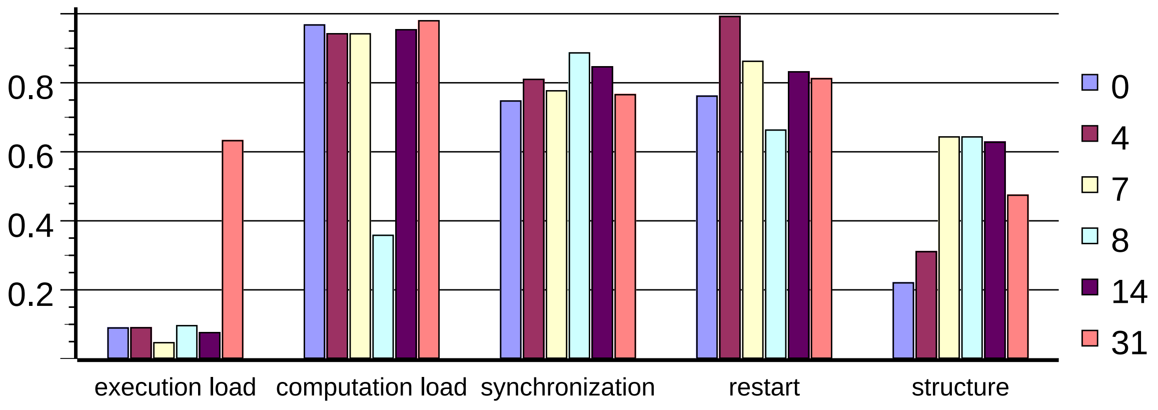


Figure 7.18: Concurrency measures for MIPS architecture (solution 2).

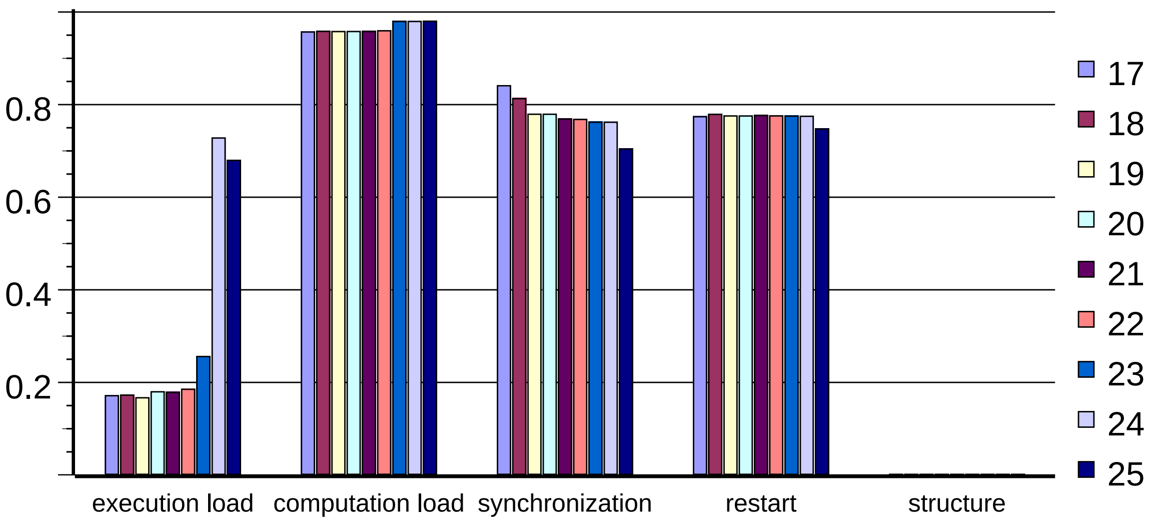


Figure 7.19: Concurrency measures for all design of solution 1 using MIPS architecture.

7.2.4 Mapping on a Multi-Processor Platform

In [18], a JPEG decoder is implemented on a single tile of the CAKE multi-processor architecture [34]. A tile consists of a heterogeneous set of processors and memories that communicate through a snooping interconnection network. Each processor has its own cache. The snooping protocol ensures that the caches have a coherent view on the single uniform shared memory space. In the article, a tile configuration is used with a homogeneous structure of MIPS processors and four memory banks to implement the memory space. All processors in the tile operate on a single queue of runable tasks. A small operating system, called run-time system, dynamically assigns tasks to processors. If one processor suspends a task, then another processor can resume this task. The YAPI library has been implemented in software on top of this tile-run-time system. In this library, each compute node is implemented as a separate task.
The article presents the implementation shown in Figure 7.20 as an optimal implementation of the JPEG decoder for this configuration of the CAKE architecture. This JPEG decoder implements also the functions for reading and writing to a file in the frontend and backend nodes. These operations have been separated in our design. This was done with the idea that they belong to the test environment and not to the design. To make a good comparison between our design and the design of Philips, we have to take those functions out in the Philips design. This gives a computational network with two extra compute nodes, one for reading and one for writing the data to a file.

To get a comparison on the performance of the designs found in the case study and the design of Philips, we performed simulations of all of these designs on a CAKE architecture. For this we used the same configuration as in [18]. We simulated the designs 25, 31 and the modified Philips design for different numbers of MIPS processors on a single tile. The results of these simulations are shown in Figure 7.21. The horizontal

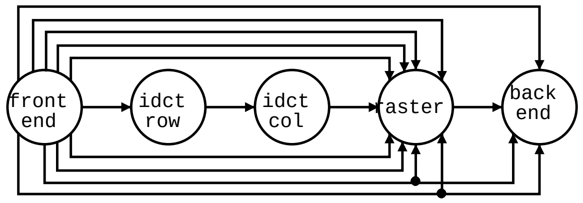


Figure 7.20: JPEG decoder Philips.

axes shows the number of MIPS processors used. The vertical axes shows the number of instructions needed to execute the JPEG decoder and the compute nodes that perform the input and output for it. The compute nodes for the input and output require in all cases the same number of instructions. Therefore we can compare the performance of the different designs by looking at the instruction counts.

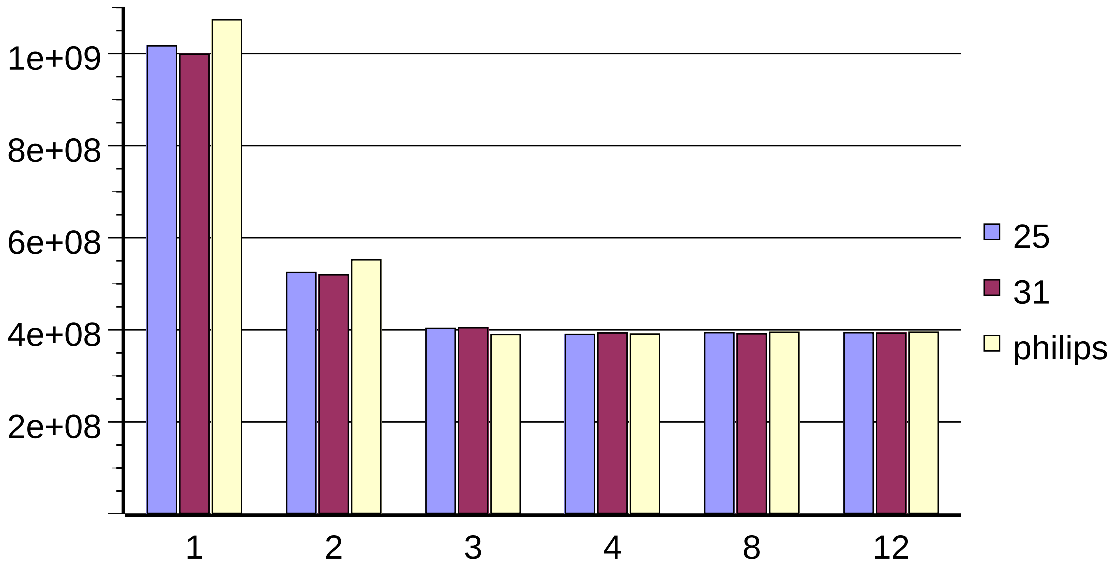


Figure 7.21: Mapping of solutions on CAKE.

The figure shows that the solutions derived in our case study have the same performance characteristics as the Philips design. They are slightly faster when one or two MIPS are used. It is interesting to note that when only 1 or 2 MIPS are used the design with the most compute nodes (design 31) has the best performance. This shows that the communication is implemented efficiently in the run-time system. The Philips design has the best performance when 3 MIPS are used. The difference in performance between the designs found in the case study and the Philips design is about 0.5 instructions per pixel of the decoded image. This difference is most likely caused by differences in coding style.

In the previous paragraph we made an analysis of all designs taking the MIPS compiler into account. This analysis showed that not design 25 but design 24 has better performance when MIPS processors are used in the systems architecture. With CAKE it is possible to check whether this conclusion is valid. Therefore we simulated design 24 on the same CAKE architecture using 3 MIPS processors. Figure 7.22 shows the number of instructions needed to decode the image on the CAKE architecture using 3 MIPS processors. It shows that design 24 has better performance than design 25. The conclusion made based on the CAST analysis is valid. The difference between the two design is however small compared to the difference indicated by the execution load measure for the two designs. An explanation for this might be the costs of communication. These are probably not estimated accurately enough in the case study. This influence of communication costs determines the difference in estimated performance using CAST and measured performance using CAKE. A different explanation might be that the scheduler of the run-time system finds a better schedule for executing the compute nodes than the one we create when we merge these nodes. A final conclusion on why this difference is smaller than estimated by CAST cannot be made.

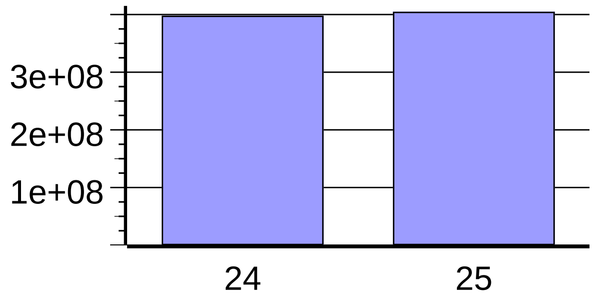


Figure 7.22: Mapping of designs 24 and 25 on CAKE.

We end this section with a remark on the simulation time required for simulations using the CAKE simulator and CAST. Both simulation used the same dual Pentium III 1GHz computer with 4GB internal memory. Only a single processor was used in both cases. A simulation using CAKE and the shuttle image took 45 minutes to complete, while a simulation with CAST and the same image took 4 minutes.
The CAKE environment simulates everything at a lower level of abstraction, making the result more reliable. But during this part of the design trajectory, this amount of detail is not needed. That makes CAST a more suitable solution, as it allows for fast design exploration. It also leafs open more implementation options. Code optimized for CAKE may not be optimal for another architecture.

7.3 Conclusion

The JPEG case study described in this chapter shows that the design exploration method and concurrency model provide a method for deriving a computational network in which all compute nodes have a balanced execution load and a communication behavior tailored to the system. The case study shows that the design exploration method and concurrency measures indicate which transformation must be performed. The concurrency measures provide feedback on their effect on the performance of the system. This helps the designer in deriving in a structured way a concurrent implementation of an application. This structured approach enables us to identify different implementations of the JPEG decoder. These implementations can be compared using the concurrency model.

Using the CAKE simulation environment, we could demonstrate that the derived design has a performance similar to a JPEG decoder made by an experienced designer without an explicit fine-tuning step adapting the application to the architecture. This demonstrates that simply following the design exploration method gives a result comparable to that found by an experienced designer.

The case study demonstrates that the instruction set has an influence on the result. This influence is however marginal if no application specific instruction set is targeted. The effect of the instruction set can be further canceled out by simple re-simulation of the created designs with the appropriate compiler.

Another important conclusion of the case study is that the costs of communication have a large impact on the systems performance. The biggest change in the concurrency measures occurs at the point where the costs of communication were changed. This shows that modeling these costs accurately is a very important step in the design trajectory.

Chapter 8

Conclusion and Recommendations

Concurrency will play an important role in next-generation embedded multi-media systems. These multi-media systems will often be multi-processor systems, which are inherently concurrent. To exploit the concurrency in these architectures, it must be made visible in the mapping of applications onto such architectures. This requires that it is made explicit in the specification of an application. This in turn requires a model of computation that can be used to specify a concurrent application and that allows formal reasoning about the concurrency in it. Existing models of computation do not meet with these requirements. To address this need, the research in this thesis was carried out with the following three goals:

- To develop a model of computation for parallel computations that is based on existing models of computation and allows formal reasoning about concurrency;
- To develop a concurrency model that allows formal reasoning about concurrency in an application;
- To propose a design exploration method that improves the concurrency properties of a parallel computation.

To state the outcome of the performed research, this chapter describes first the realized goals; then it gives some recommendations for future developments.

8.1 Realized goals

In general, we can state that the research described in Sections 3.2, 4.3, 5.2 and 5.4 contributes to fulfill both the first and second goal. Chapter 6 contributes to fulfill the third goal.

In Sections 3.2, we describe a model of computation for parallel computations, the computational-network model. The computational-network model allows the modeling of all kinds of different properties of systems in a very natural way. It introduces a natural way to explicitly specify concurrency in a system by introducing different compute nodes in a computational network. Section 5.2 provides a first implementation of this model. This implementation is based on YAPI, a Kahn process networks implementation developed at Philips Research. With the choice for YAPI, we immediately have a large set of applications at hand that are modeled using YAPI and that can be re-used in our work. The case study presented in Section 7.2 shows that the computational-network model is suitable for modeling real applications and specifying the concurrency in it.

In Section 4.3, we describe a concurrency model that consists of five main measures and is supported by a set of detailed measures. The main measures capture the different concurrency properties of the computational network. The detailed measures provide an insight in the concurrency properties of the components in the computational network. The examples presented in Section 4.4 and the JPEG case study show that the five concurrency measures are all meaningful. Each measure gives an insight in a different concurrency property. The measures allow formal reasoning about the concurrency in the specification as is shown in the case study. The results of the case study suggest also that the measures of the concurrency model are sufficient for obtaining good results.

In Section 5.4, we present the Concurrency Analysis and Simulation Tool (CAST). This program implements the analysis and simulation environment for computational networks. It takes a computational network and settings for the delay functions used in the time-stamping mechanism as an input. CAST executes the computational network and determines values for the concurrency measures. The statistical analysis option of CAST allows for the abstraction from a given input. This makes it possible to reason in a more abstract way about the concurrency in a computational network.

Chapter 6 describes a design exploration method. This method uses the concurrency model to create in five steps a computational network that has a balanced execution load and communication behavior tailored to the application. The case study described in Section 7.2 shows that this approach works. The design exploration method and concurrency measures indicate clearly which transformation must be performed and what its effect is on the performance of the system. It helps the designer in deriving in a structured way a concurrent implementation of an application.

The JPEG case study shows that the design exploration method is useful in finding in a structures way an specification that optimizes concurrency. The design exploration method helps in finding different specifications, solutions. These can be compared using the concurrency model as is shown in the case study. The case study shows further that it is possible to find a solution that is similar to a solution constructed by an experienced designer by using the design exploration method. The CAST analysis makes it possible to perform this optimization independent of the multi-processor architecture, while the experienced designer created an optimal implemenation for a specific architecture. CAST makes it possible to perform this optimzation also much faster than with a simulation model of the architecture.

The result from the JPEG decoder case study is very promising. It shows that the computational-network model, concurrency model and design exploration method help a system-designer in finding a system specification that has a good concurrent behavior. The concurrency model makes it possible to reason in a formal way about the concurrency in a system. This helps the system designer to derive in a structured way a concurrent system. To conclude, we can that the case study shows that the proposed computational-network model and concurrency model allow specification of and formal reasoning about concurrency in an application. These models make it possible to optimize the concurrency in a specification in an architecture independent way.

8.2 Recommendations

This section contains recommendations about future research into concurrency in computational networks. It gives recommendations for changes and extensions to the computational-network model, concurrency model and design exploration method as well as changes and extensions to the implementation of them.

The results of the JPEG case study are very promising, but more experiments are needed. These experiments must verify that the concurrency model captures all concurrency properties of a computational network and that the measures always provide a good insight in the concurrency. These experiments must include different applications and architectures to verify all assumptions made in the concurrency model.

The structure and synchronization measure cannot be computed compositionally with the definitions given in this thesis. This make it impossible to get an insight in the concurrency properties captured by these measures for the individual compute nodes and network components. This insight will be important when changes must be made to the computational network. Therefore it is recommended that a compositional computation for all 5 concurrency measures is created.

The value of the computation load dropped dramatically in the case study when the costs of communication were changed. This shows that the costs of communication have a large impact on the systems behavior. Therefore, it is important to model them accurately when performing a concurrency analysis. The costs of communication are modeled in the present concurrency model using a simple linear function. This linear function models only in a very generic way the physical communication medium. More research is needed to find a model that provides on an abstract level better information about the costs of communication.

The concurrency model takes only very little information about the architecture into account. It does not provide feedback on timing and energy in relation with the concurrency. The problem definition, Chapter 2, discussed that all of these aspects will become important when a multi-processor mapping trajectory is developed. It must therefore be researched how the concurrency model can be extended to take all of these things into account.

The computational-network model is not suited very well for capturing control type of applications or reactive behavior. Research on how the computational-network model can be extended to capture these things is recommended. This will give a generic framework for specifying applications that are mapped onto multi-processor systems.

The concurrency model uses the structure of the computational network in the structure measure. The structure plays also an implicit role in the other measures. The concurrency model does not link in a direct way the computational paths found in the structure and the other concurrency measures. However, this relation between the structure and the concurrent behavior of the system is important. Consider for instance the situation in which two computational paths join in a compute node. The events in this compute node will impose causality relations on the events of the compute nodes that are on these two computational paths. There is in this situation a clear link between the structure of the computational network and the event ordering. If this link is clear, one gets a better insight in which computational path(s) must be considered when optimizing the concurrency. It should therefore be investigated how to extend the concurrency model, so that it provides a better insight in the concurrency of the different computational paths.

The current implementation of the time-stamping mechanism in CAST uses text based files. This makes the time-stamping mechanism slow. It is recommended that a binary file format is used to speed-up the time-stamping.

A system designer that uses the design exploration method will get a set of designs, which are individually evaluated using CAST. The goal of the design exploration method is that the concurrency measures improve over this set of designs. To see this, the system designer must compare the concurrency measures of the different CAST runs. Comparing these values is not supported by CAST. It is however very important that these values can be compared in an intuitive way. It is therefore recommended that a user interface is developed that visualizes these measures and allows the system designer to compare different designs.
The user interface should also guide the system designer through the different steps of the design exploration method and suggest transformations in the network to the system designer. A number of these transformations can be performed in a (semi-) automatic fashion. It should be investigated which transformation can be performed under which conditions in a (semi-) automatic way. This research should lead to a tool that supports these transformations. Preferably, this tool is integrated with the user interface.

When a computational network is executed, a trace of all events in the computational network is generated. This trace is used in the concurrency model to calculate measures for the different concurrency properties of the system. Based on these result, a system designer selects a set of compute nodes that will be modified. One of the possible

modifications is to merge two (or more) compute nodes. The system designer must then manually code a new compute node that combines the two compute nodes. After that, the impact of this step can be evaluated with the concurrency model. For this evaluation, a new trace is generated and analyzed. This new trace will be very similar to the original trace. It will only no longer contain the communication between the merged compute nodes. The merging of the compute nodes has also imposed an ordering, a schedule, on the events that take place in the merged compute node.
It should be investigated whether it is possible to predict the ordering on the events in the merged compute nodes using a scheduling algorithm. In that way it would become possible to analyze the computational network with a number of compute node merged, without writing the code for it. As this scheduling is an estimation of the behavior of the merged compute nodes, this concept cannot be extended to merging a large set of compute nodes. The new compute node has then very likely a different behavior then predicted. However this method allows for automated design exploration around a given design. This will probably result in the need to code less design by hand and with that a faster design cycle.

The analysis of the computational network is now based on simulation models. This requires that compute intensive simulations are performed before the actual analysis can be performed. With systems getting bigger it may become very impractical, or even impossible, to use these simulation models. Another disadvantage of the simulation models is that they provide only insight on the behavior of the computational network for a given input. It is not possible to reason about the behavior for an abstract input of the network. These problems can be solved by using analytical models.

Bibliography

[1] *Alders, D., E.A. de Kock, P. van der Wolf and P.E.R. Lippens, Action synchronization. Technical Note 2001/087. Philips Research, Eindhoven, 2001.*

[2] *Basten, T. and J. Hoogerbrugge, Efficient execution of process networks. In: Communicating Process Architectures, Proc., Bristol, UK, September 2001. Amsterdam, The Netherlands: IOS Press, 2001. p. 1-14.*

[3] *Bergstra, J.A. and J.W. Klop, Algebra of communicating processes. In: CWI Symp., Proc., Amsterdam, The Netherlands, November 1983. Amsterdam, The Netherlands: Centre for Math. & Comput. Sci., 1983. p.89-138.*

[4] *Bhaskaran, V. and K. Konstantinides, Image and video compression standards. Amsterdam, The Netherlands: Kluwer Academic Publishers, 1995.*

[5] *Cadence Virtual Component Co-Design (VCC), http://www.cadence.com/products/vcc.html*

[6] *Chandrakasan, A.P., S. Sheng and R.W. Brodersen, Low-power CMOS design. In: Solid State Circuits, Volume 27, Issue 4. IEEE, April 1992. p. 472-484.*

[7] *Droste, M., Concurrency, automata and domains. In: Automata, Languages and Programming. Proc. 17th Int. Coll., Coventry, England, July 1990. Berlin, Germany: Springer-Verlag, 1990. p. 195-208.*

[8] *Engel, S.L., Mapping signal processing applications onto heterogeneous multiprocessor architectures. Master's Thesis, Eindhoven University of Technology, Faculty of Electrical Engineering, 2001.*

[9] *Fidge, C.J., Dynamic analysis of event ordering in message-passing systems. Pd.D. thesis, Australian National University, Australia, 1989.*

[10] *Harel, D., Statecharts: a visual formalism for complex system. In: Science of computer programming, Volume 8, Issue 3, June 1997. p. 231-274*

[11] *Janneck, J.W., Generalizing lookahead-behavioral prediction in distributed simulation. In: Parallel and Distributed Simulation 1998. Proc. 12th Workshop,*

Alta, Canada, May 1998. IEEE, 1998. p. 12-19.

[12] *International Telecommunication Union, Information technology - digital compression and coding of continuous-time still images. ITU Recommendation T.81, September 1992. ITU, 1992.*

[13] *Kahn G., The semantics of a simple language for parallel programming. In: Information Processing 74, Proc., Stockholm, Sweden, August 1974. Ed. by J.L. Rosenfeld. Amsterdam, The Netherlands: North-Holland Publishing Co., 1974. p. 471-475.*

[14] *Kahn G. and D.B. MacQueen, Coroutines and networks of parallel processes. In: Information Processing 77, Proc., Toronto, Canada, August 1977. Ed. by B. Gilchrist. Amsterdam, The Netherlands: North-Holland Publishing Co., 1977. p. 993-998.*

[15] *Karp, A.H., and H.P. Flatt, Measuring parallel processor performance. In: Communications of the ACM, Volume 33, Issue 5, May 1990. New York, USA: ACM Press, 1990. p. 539-543.*

[16] *Kock, E.A. de, et al., YAPI: application modeling for signal processing systems. In: Design Automation Conference, Proc. 37th Int. Symp., Los Angeles, USA, June 2000. IEEE, 2000. p. 402-405.*

[17] *Kock, E.A. de, and G. Essink, Y-chart application programmer's interface. Application programmer's guide version 1.0.1. Philips Research, Eindhoven, 2001.*

[18] *Kock, E.A. de, Multiprocessor mapping of process networks: a JPEG decoding case study. In: System Synthesis, Proc. 15th Int. Symp., Kyoto, Japan, October 2002. (to be published)*

[19] *Kung, H.T., and C.E. Leiserson, Systolic arrays for VLSI. In: Sparse matrix symposium, Proc., Philadelphia, USA, 1978. Philadelphia, USA: SIAM,, 1979. p. 256-282.*

[20] *Kung, H.T., Why systolic architectures? In: IEEE Computer, Volume 15, Issue 1. IEEE, 1982. p. 37-46.*

[21] *Kung, S.Y., VLSI array processors. London, UK: Prentice Hall.*

[22] *Lamport, L., Time, clocks, and the ordering of events in a distributed system. In: Communications of the ACM, Ed. by R.S. Gaines., Volume 21, Issue 7, 1978. p. 558-565.*

[23] *Lee, E.A. and T.M. Parks, Dataflow process networks. In: Proc. of the IEEE, Volume 83, Issue 5, May 1995. IEEE, 1995. p.773-801.*

[24] *Loeffler, A.L.C. and G. Moschytz. Practical fast 1d dct algorithms with 11 multiplications. In: Acoustics, Speech and Signal Processing, Proc. Int. conf. on, May 1989. p. 988-991.*

[25] *Luke, E.A. and I. Banicescu, J. Li, The optimal effectiveness metric for parallel application analysis. In: Information processing letters, Volume 66, Issue 5. June 1998. p. 223-229.*

[26] *Mattern, F. Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms. Ed. by M. Cosnard et al. Amsterdam, The Netherlands: Elsevier Science B.V., 1989. p. 215-226.*

[27] *Matzke, D., Will physical scalability sabotage performance gains? In: IEEE Transactions on Computing, Volume 30, Issue 9. IEEE, 1997. p. 37-39.*

[28] *Mazzeo, A. and N. Mazzocca, U. Villano, Efficiency measurements in heterogeneous distributed computing systems: from theory to practice. In: Concurrency: Practice and experience, Volume 10 Issue 4, May 1998. New York, USA: John Wiley and Sons. p. 285-313.*

[29] *Petri, C.A., Kommunikation mit automaten. Ph.D. thesis, Institut für instrumentelle Mathematik, Bonn, Germany, 1962.*

[30] *Pratt, V., Modeling concurrency with partial orders. In: International Journal of Parallel Programming, Volume 15, Issue 1, 1986. p. 33-71.*

[31] *Raynal, M. and M. Mizuno, M. Neilsen, Synchronization and concurrency measures for distributed computations. In: Distributed Computing Systems 1992. Proc. 12th Int. Conf., Yokohama, Japan, June 1992. IEEE, 1992. p. 700-707.*

[32] *Roig, C., A. Ripoll, M.A. Senar, F. Guirado and E. Luque, Improving static scheduling using inter-task concurrency measures. In: Parallel Processing Workshops 2001. Proc. Int. Conf., Valencia, Spain, September 2001. IEEE, 2001. p. 375-381.*

[33] *Skillicorn, D.B. and D. Talia, Models and languages for parallel computation. In: ACM Computing Surveys, Volume 30, Issue 2, June 1998. New York, USA: ACM Press. p. 123-169.*

[34] *Stravers, P. and J. Hoogerbrugge, Homogeneous multiprocessing and the future of silicon design paradigms. In: Int. Symp. on VLSI Technology, Systems and Applications 2001, Proc., Hsinchu, Japan, April 2001. IEEE, 2001. p.184-187.*

[35] *Strik, M.T.J., A.H. Timmer, J.L. van Meerbergen and G.J. van Roostelaar, Heterogeneous multiprocessor for the management of real-time video and graphics streams. IEEE journal of solid state circuits, Volume 35, Issue 11, November*

2000. IEEE, 2000. p. 1722-1731.

[36] *SystemC, http://www.systemc.org/*

[37] *Togneri, R., Parallel program analysis on workstation clusters: speedup profiling and latency hiding. In: Concurrency: Practice and experience. Volume 9, Issue 7, July 1997. New York, USA: John Wiley and Sons. p.721-751.*

[38] *Unified Modeling Language (UML), http://www.uml.org/*

[39] *Wallace, G.K., The JPEG still picture compression standard. IEEE Transactions on Consumer Electronics, Volume 38, Issue 1. IEEE, 1992. p. xviii-xxxiv.*

[40] *Wong, C. et.al., Task concurrency managment methodology summary. In: Design, Automation and Test in Europe, Proc., Munich, Germany, March 2001. Ed. by W. Nebel, A. Jerraya. IEEE, 2001. p. 813.*

Appendix A

Concurrency Analysis and Simulation Tool

A.1 Introduction

This appendix contains information about using the Concurrency Analysis and Simulation Tool. It describes the command line options and parameters used in the project and delay settings files. The transcript of a run of CAST is also given.

A.2 Command line options

Type `cast [options] [project]` to execute the Concurrency Analysis and Simulation Tool.

Options

- Options to control the CAST steps:

`-p`	Run parse stage
`-s`	Run simulation stage
`-s-compile`	Run compile stage of simulation
`-s-run`	Run simulation
`-a`	Run analysis stage
`-a-eventorder`	Create event ordering
`-a-simulation`	Determine concurrency measures for simulation
`-a-statistics`	Determine statistical concurrency measures

- Options to pass to the CAST parser:

`-p-file <file>`	File to be parsed

- Options to pass to the CAST simulator:

`-s-arg <arg>`	Arguments for simulator
`-s-name <name>`	Name of simulation

- Options to set parameters:

`--setparam <name> <value>`	Set a parameter

- Misc. options:

`--clean`	Remove all data produced by CAST
`-h`	Display the help message
`-v`	Display the version

Project
`project` is the name and location of the project file. If no file is specified, CAST assumes that the file `cast.xml` in the current directory must be used. The required content of this project file is described below.

A.3 Parameters

- Project (`<project> ... </project>`)

`<networkfile file=""/>`

`file`	Name of file containing structure information. (default: `cache/network.xml`)

`<castdir dir=""/>`

`dir`	Root directory of CAST

`<yapidir dir=""/>`

`dir`	Root directory of YAPI

`<gccoption option=""/>`

`option`	Option passed to GCC compiler. Type `g++ -h` to get a list of all options.

`<simulation arg=""/>`

`arg`	Command line argument passed to executable during simulation.

`<sourcefile file=""/>`

`file`	Name of source file containing code of YAPI process network.

`<param name="" value=""/>`

`name`	Name of the parameter.
`value`	Value of the parameter.

`<delayfile file=""/>`

`file`	Name of the file containing delay settings. (default inside project file)

- Delay settings (`<delaysettings> ... </delaysettings>`)

`<delay type="rw|connection"`	`d="" node|connection"default|name"/>`
`type`	Delay setting for read / write or connection.
`d`	Value of delay constant(s).
`node`	Hierarchical name of the node. (default applies to all nodes)
`connection`	Hierarchical name of the connection. (default applies to all connections)
`<primary node=""/>`	
`node`	Hierarchical name of the primary node.

A.4 Example

This paragraph contains the transcript of a CAST run with the producer-consumer example of YAPI. This example can also be found in the `examples/pc` directory of the CAST release.

Begin of Transcript

```
[sander@co3 pc]cast -p -s -a
Concurrency Analysis and Simulation Tool (v1.0)
Load project...
Parse...
File: consumer.cc
File: producer.cc
File: pc.cc
File: main.cc
Simulate (compiling)...
File: ./cxx/consumer.cc
File: ./cxx/producer.cc
File: ./cxx/pc.cc
File: ./cxx/main.cc
Simulate (linking)...
Simulate (running)...
Consumer started
Producer started
Producer pc.prod: 1000 values written
Consumer pc.cons: 1000 values read
Computation Workload:
---------------------------
|Process Instruction Count|
|pc.prod                  |
|pc.cons                  |
---------------------------
Communication Workload:
```

```
-----------------------------------------------
|        Wtokens Wcalls T/W Rtokens Rcalls T/R|
|pc.fifo    1001   1001   1    1001   1001   1|
-----------------------------------------------
Analyze (simulation)...
Creating event ordering.
Concurrency measures:

 measure        |     value
 ---------------+-----------
 execution ld   |  0.908309
 computation ld |  0.325676
 synchronization|  0.449134
 restart        |  2.04e-05
 structure      |         0
```

End of Transcript