TU/e technische universiteit eindhoven

Analyzing Concurrency in Streaming Applications

Sander Stuijk, Twan Basten

ES Reports
ISSN 1574-9517

ESR-2005-05
22 March 2005

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Analyzing Concurrency in Streaming Applications

S. Stuijk and T. Basten
Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.
{s.stuijk, a.a.basten}@tue.nl

March 23, 2005

Abstract

We present a concurrency model that allows reasoning about concurrency in executable specifications of streaming applications. It provides measures for five different concurrency properties. The aim of the model is to provide insight in concurrency bottlenecks in an application and to provide global direction when performing implementation-independent concurrency optimization. The model focuses on task-level concurrency. A concurrency optimization method and a prototype implementation of a supporting analysis tool have been developed. We use the model and tool to optimize the concurrency in a number of multimedia applications. The results show that the concurrency model allows target-architecture-independent concurrency optimization.

1 Introduction

The consumer-electronics market is characterized by rapid developments in embedded multimedia systems. In recent years, we have for instance seen successful market introductions of portable MP3 players, digital cameras and set-top boxes. The pace at which new products are introduced on the market is ever increasing. Consumer-product manufacturers try to cope with this trend by decreasing their time-to-market. On the other hand, the complexity of embedded multimedia systems is growing, as users have high expectation about the functionality and quality delivered by new products. To deal with these adverse trends, the electronic-design community expects that future electronic systems re-use platforms that integrate many IP-blocks. Software can be executed concurrently on the IP-blocks in these *multi-processor systems-on-chip*. Novel programming techniques are required to use these systems. These techniques must exploit the concurrency that is present in the hardware architecture and meet with the timing-, energy-, performance-, and cost-constraints. A coarse overview of the multi-processor system-on-chip programming trajectory is shown in Figure 1. The figure shows a subdivision of the programming problem into two subsequent steps (mapping and binding). The programming of the hardware level is done from an intermediate level, called the implementation level. This step binds one (or a few) compute tasks onto one processor. In this way, we can relay on traditional compiler technology and minimize the overhead of a run-time system. The step from the specification level to the implementation level is responsible for subdividing the (executable) specification in such a way that the resulting tasks can efficiently be programmed on the hardware platform. This so-called multi-processor mapping step must consider aspects like concurrency, energy and timing. To do this, it will need information about the underlying hardware platform. This information is gradually added during the mapping trajectory.

The programming trajectory covers a system-level design methodology from the early design stages till the actual system-on-chip solution. We focus on (data-intensive) streaming applications, as we are targeting multimedia applications. Many design flows for embedded multimedia systems are based on some kind of task graph [BWH+03, BHLM94, MK03, PvdWD+00, TC00].

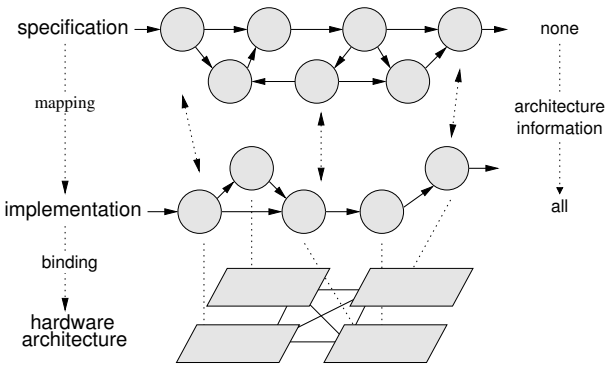


Figure 1: Multi-processor system-on-chip programming trajectory.

All such flows benefit from a good initial task graph as their input. In current design practice, the programming trajectory for such applications typically starts with an executable specification of the application written by an application designer. The specification is usually given as a sequential program that describes the logical functions used in the application and it is written in a language like C or C++. Target platforms usually allow for concurrent execution of the application. For this reason, part of the mapping flow deals with the extraction of the concurrency from the application. Concurrency has a large impact on the system, which means that the extraction should be performed early in the programming flow. Concurrency analysis is to some extent independent of the (precise) architecture targeted. In other words, some transformations performed to extract concurrency from the application are valid for a class of architectures. This enables efficient re-use of optimizations performed on the specification if changes are made to the hardware platform. In this paper, we propose a technique to analyze the concurrency that can be extracted independent of the exact target architecture. The result of the extraction is a task graph that makes data transformations and the underlying data streams explicit. It is our aim to provide a specification of the application which forms a good starting point for mapping it onto many different systems-on-chip platforms. In other words, we try to answer the question of how to come at a good initial task graph.

The next section gives an overview of related work on multi-processor programming and concurrency analysis and discusses the novelty of our approach. In Section 3, an abstract model for parallel (streaming) computations, called the computational-networks model, is presented. The model abstracts from the precise notion of a task graph that is often used in later stages of the mapping flow. The concurrency model is discussed in Section 4. A prototype concurrency-analysis tool implementing the concurrency measures is presented in Section 5. In Section 6, a supporting concurrency optimization method is presented. The concurrency model is used in a number of case studies to optimize the concurrency of a JPEG decoder, an H.263 video-conferencing decoder and a 3D recursive search algorithm. The results are presented in Section 7. Section 8 contains some discussion, and Section 9 concludes.

2 Related work

Multi-processor systems inherently contain concurrency. This concurrency must be exploited in the programming trajectory. This requires a model of computation which allows the specification of concurrency in an application and the application should fit easily in it. Furthermore, the description of the application should be at the correct abstraction level to perform the required analysis [KNRSV00]. A comprehensive survey of models for parallel computations used in different application domains and at different abstraction levels is given in [ST98]. The most interesting models for our application domain, streaming multimedia systems, are the dataflow

models. Examples of these are Kahn process networks [Kah74, KM77] and Synchronous dataflow [LP95]. In [LSV98], a framework is presented to compare the notions of concurrency, communication, and time between different dataflow models. Based on the desired notions, a designer can pick a model of computation that fits best with the application domain to describe the software and hardware behavior of a system component. The system components in the collection of components that form a system can be described in different models of computation. The Ptolemy framework [BHLM94] formalizes the interaction of different models of computation. As a result, a system composed out of system components in different models of computation can be analyzed. Our model of computation, presented in the next section, is an abstraction that generalizes all commonly used models, thus enabling system-level concurrency analysis.

Besides a model of computation that describes the concurrency, we need a concurrency model to reason about the concurrency in the application. Concurrency optimization is studied in the field of systems-on-chip design as part of the problem of multi-processor programming. Typically, the concurrency analysis is (an implicit) part of a design flow that maps an application onto a multi-processor system. The different design-flow approaches are best classified according to the used optimization criteria and abstraction level. The interested reader is referred to [Gri04] for an extensive overview and classification of related work in this area. Artemis [PvdWD$^+$00] is one of the projects mentioned in this paper. It is based on a Kahn process network description of the application and incorporates the ideas from SPADE [LvdWDV01], i.e., system-level co-simulation is performed by using symbolic instruction traces generated and interpreted at run-time through manually defined abstract performance models. In [EEP03] a technique to perform a multi-objective design-space exploration with Artemis is presented. Our approach provides an automatic way to derive performance numbers for a class of processor architectures simplifying the analysis. We also provide a design-space exploration method to extract concurrency from an application. Other hierarchical design-space exploration methods are Milan [MPND02], Mescal [MK03] and Metropolis [BWH$^+$03]. Milan combines tools for design-space pruning with simulations at different levels of abstraction. Simulators include trace-driven, task-level evaluation tools as well as cycle-accurate thrid-party simulators. Mescal provides a correct-by-construction mapping flow targeting heterogeneous, application-specific, programmable (multi-)processors. Applications can be specified in any combination of models of computation that is natural for the application. The Metropolis framework allows the description and refinement of a design at various levels of abstraction. Applications are modeled as a set of communicating processes. The performance numbers generated by the simulations tools of Metropolis are based on user-specified annotations. All three approaches focus on performing a design-space exploration. None of them aims explicitly at analyzing and extracting concurrency from an application, while this is the explicit goal of our work. Our work is complementary to these design-space exploration tools and it can be integrated into their flow.
A number of approaches exist in the field of system-on-chip design that explicitly aim at concurrency analysis and extraction. A good example of this is the Compaan tool [KRD00]. It automatically transforms nested loop programs into a process network which makes the concurrency explicit. The tool finds all possible concurrency contained in the application. The constraint that a program must be specified as a loop structure is absent in our work. Also, the most concurrent program is not always the best program, e.g., due to communication overhead. Our approach takes all these aspects into account. Another approach to concurrency analysis is found in [SRV$^+$03]. The authors present a technique to identify task-level concurrency independent of the target architecture. The approach is defined on JAVA program constructs and lacks a more formal underlying model. Our work defines such a model. One concurrency measure that considers the longest path in a task graph is defined. Our concurrency model contains a similar measure, but we show that more measures are needed to find all potential concurrency bottlenecks.

Analysis and extraction of concurrency from applications has also been studied extensively in the field of distributed computing. Most techniques analyze the data-dependencies between different parts of the application. These data-dependencies are expressed with a partial order [Pra86] or a message sequence chart [HL00]. Ravindran et al. describe in [RT93] the different sources of concurrency that can be identified by looking at the causality relations between events that occur in an application. A message sequence chart is used to express these causality relations and to define a concurrency measure. This measure considers the ratio between the number of orders in which messages can be communicated in the graph (all possible ways to interleave the messages while obeying the precedence relations) and the number of orders in which messages can be communicated in the graph if there are no precedence relations. Raynal [RMN92] developed another set of measures that quantify the degree of concurrency in a parallel computation. The model of computation used by Raynal is based on Lamport's logical clocks [Lam77]. It uses event diagrams to graphically depict the execution of the computation. One of the important goals of Raynal's paper is to provide analysis techniques independent of real time effects, such as system load and processor speed (i.e., independent of the hardware architecture). This differentiates it from most other concurrency optimization techniques in the field of distributed computing. Typically, concurrency optimization is performed for a given system architecture [MMV98]. A good example of this is the research on performance analysis and design optimization for systolic processors [JHS93, Kun98]. Commonly considered optimization criteria are the computation time, latency, throughput and the number of processors. These criteria form also the motivation for our work. Our concurrency model is most closely related to that of Raynal. The execution model is based on the same principles. Concurrency measures are defined that provide insight in the synchronization constraints between and workload of the different processes. There are two important differences between the approach of Raynal and our work. First, our concurrency model considers more aspects that influence the concurrency than Raynal - i.e. our model contains additional measures that identify concurrency bottlenecks not found by Raynal's measures. Second, the model of computation used by Raynal neglects the time needed to communicate data between tasks. This is not a valid assumption in the domain of streaming multimedia applications. In these applications, large amounts of data are communicated between tasks. Different mappings of the tasks onto the platform result in a different timing behavior of the application as these mappings may require different use of the on-chip interconnect. For this reason, communication time is taken into account in our concurrency model.

3 Model of Computation

The model of computation we introduce in this section captures the core of parallel (streaming) applications. We only specify those aspects that are necessary for concurrency analysis. In this way, it allows for many instantiations. The model is, for example, sufficiently abstract to comprise a number of data-flow models like Kahn process network [Kah74, KM77] and Synchronous dataflow [LP95] and a subclass of Petri nets, called marked graphs [CHEP71]. Our concurrency analysis can be applied to (executable) specifications in all these models.

3.1 Computational Networks

We assume that a parallel computation is organized as a collection of autonomous compute nodes that are connected to each other by point-to-point connections. Compute nodes exchange information through these connections. These connections are the only way of communication. A given node computes on data coming along its input connections to produce output on some or all of its output connections.

This informal definition of a parallel computation is the basis of the computational-network model. A compute node has a set of input ports and a set of output ports for connections to its environment. Input and output data is modeled using strings of data-elements, which is a good abstract model of data streams. The execution of a compute node implies the reading of input strings from its input ports and writing the appropriate output strings to its output ports. This is done following the *transformation* that describes the behavior of the node.

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

- I *is a set of input ports;*
- O *is a set (disjoint of* I*) of output ports;*
- t *is a transformation, for example, a function describing how a compute node computes a (tuple of) output strings using a (tuple of) input strings.*

Note that we are not interested in the exact form of a transformation and that we do not define the types of data received and sent over ports. For our purposes, these details are irrelevant, and including them in the definition of compute nodes would unnecessarily restrict and complicate matters. The only requirement is that transformations allow an operational implementation, as further explained in the next sub-section. The definition of a compute node enables us to define a *computational network*. It contains a set of compute nodes that are connected to each other using point-to-point connections that transfer data streams in order (fifo communication). We abstract from the exact capacity of connections. Some ports of compute nodes may remain unconnected. These ports allow connections to the environment.

Definition 3.2 (Computational network) *A computational network CN is a tuple* (N, C, I, O) *where*

- N *is a set of compute nodes;*
- C *is a set of connections;*
- *every connection in* C *connects an output port of a compute node to an input port of a compute node;*
- *every port of every compute node is connected to at most one connection;*
- I *is the set of input ports of the computational network, being defined as those input ports of the nodes in* N *not connected to a connection in* C*;*
- O *is the set of output ports of the computational network, being the unconnected output ports of the compute nodes in* N*.*

An example of a computational network is shown in Figure 2. The network contains five compute nodes. The input port of node a is unconnected and thus an input port of the network; d and e provide output ports to the environment.

The definition of a computational network does not allow hierarchy. However, it is straightforward to generalize the definition by allowing networks as nodes of another network. A hierarchical model would be useful if the accompanying concurrency model is compositional. However, this is not the case for our concurrency model. Based on our experiments, it seems, that compositionality is not crucial. We find it more important that the concurrency model captures all aspects of concurrency in an appropriate way and leave compositionality for future work.

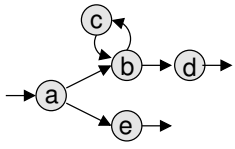


Figure 2: An example of a computational network.

3.2 Executions

In this sub-section, we give an abstract notion of executions of computational networks, which forms the basis for our concurrency model. A computational network consists of a set of compute nodes which together perform a computation. The nodes communicate with each other through connections. Each node performs a sequence of actions (e.g. C/C++ statements in an executable specification) which are modeled as a totally ordered sequence of events. These events are classified into the following three types:

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

2. **read event** Such an event models a read operation from an input port.

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

Lamport [Lam77] has shown that the events in such an event model form a partial order. This partial order is called the *causality relation* or *happened before relation* and is denoted by $\prec$. Lamport's *logical clocks* can be used to create an ordering that is consistent with causality for all events that occur during a computation. In our work, we use an adapted version of Lamport's clocks to obtain an abstract notion of time that we use in our concurrency model. Lamport's system of logical clocks assumes one logical clock per compute node. This logical clock assigns to every event a time-stamp that is the logical clock value at the moment the event occurred. Every event is performed within a period corresponding to a single logical clock value. The clock of a node is incremented once between two events. Furthermore, since communication imposes a causality relation that must be respected by the logical clocks, the clock of a reading node is updated using the time-stamp of the received event: the local clock is set to the maximum of the received time-stamp and the current clock value.
To reason accurately about timing aspects without referring to concrete implementations, we use Lamport's clocks but we associate a duration with the events that take place in the compute nodes and with the communication over the connections. Of course, these durations must be in some way reasonable for actual system implementations. They should be abstract but realistic. The duration for a connection can for instance be based on the amount of data communicated and the propagation delay of the on-chip interconnect. The latter is an architecture-dependent property that can easily be taken into account in the duration assignment. We introduce a *duration function* d that maps a set of events E of an execution plus the set of connections C of a computational network to the set of natural numbers, $\mathbb{N}$. Formally, $d : E \cup C \rightarrow \mathbb{N}$. Lamport's logical clocks can be modeled by assigning a duration of one to every event and a delay of zero to every connection.
Our time-stamping mechanism based on Lamport's logical clocks essentially is a time-stamping function t that maps the set of events E to the totally ordered set $\mathbb{N}$, formally, $t : E \rightarrow \mathbb{N}$. This mapping is such that $e \prec e'$ implies $t(e) < t(e')$. The timestamps can be computed via a set of counters, the local logical clocks. Each compute node in the computational network maintains a different counter, all initially set to 0. 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

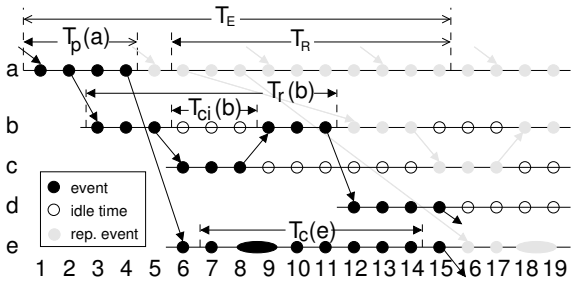


Figure 3: An event diagram.

time-stamps the event. Thus 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 n_i is the following:

1. When n_i executes an internal or 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 is the connection over which the event was received, the clock is advanced by setting $t_i := \max(t_i, y + d(c)) + d(e)$.

An execution of a computational network is the set of events that occur in all compute nodes when they transform a set of strings on the input ports of the computational network. The execution can be displayed graphically in an *event diagram*, such as the one shown in Figure 3. This figure shows in black an ordering of all events that take place during an execution in the computational network of Figure 2 for a given input, assuming the string on the input port of the network has a finite length. As we are targeting streaming applications, we have in practice often unbounded input strings. These unbounded strings can often be abstracted appropriately into an indefinite number of repetitions of the same finite input. This leads to an unbounded repetition of the same execution pattern. The gray nodes in Figure 3 represent the repetition of this execution pattern. In practice, one must make sure to get a representative execution pattern as a basis for concurrency analysis.
Let's consider the details of the execution in Figure 3. Node a starts with reading input from the environment. At the end, d and e produce values for the environment. The connection between nodes a and e has a duration of one logical clock value, all other connections have a duration of zero logical clock values. Node e executes one event that takes two logical clock values, all other events require one logical clock value. Note that this diagram is kept simple for illustrative purposes. The annotations in the diagram are explained below.
Our time-stamping mechanism can be used to analyze the ordering and abstract timing of events that take place in a computation. We define a number of measures, illustrated in Figure 3. Note that all the measures introduced here and also most measures of the next section are defined with respect to a single execution. We do not explicitly mention this execution in all formulas because that would compromise readability. Let $CN = (N, C, I, O)$ be a computational network.

Definition 3.3 (Processing time) *The processing time, $T_p(n)$, of a compute node $n \in N$ in which the set of events E_n occurs, is defined as follows:*

$$T_p(n) = \sum_{e \in E_n} d(e)$$

Definition 3.4 (Execution time) *The execution time, $T_E(CN)$, of CN is defined as the logical time needed for an execution. In other words, the execution time equals the largest value of the local logical clocks of all compute nodes at the end of the execution.*

Definition 3.5 (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 as follows:*

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

The term 'computation' is mainly used in this paper for the transformations performed on the strings of data in the network and not the communication of the strings of data. The combination of computation and communication is referred to with the term 'processing'; if idle time is taken into account as well the terms 'execution' and 'run-time' are used.

Definition 3.6 (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 write event occurred.*

We explained that we are aiming at streaming applications. The execution of a network can therefore be seen as a repetition of a single execution pattern. This repetition is shown with the gray nodes in the event diagram of Figure 3. Idle times at the beginning of nodes, such as b, c, d and e, in Figure 3, and the end of nodes, such as a, b, c, can be used for operations on other inputs. This motivates Definition 3.6, as well as the following definition.

Definition 3.7 (Run time) *The run-time $T_R(CN)$ of CN and $T_r(n)$ of node n is:*

$$T_r(n) = T_p(n) + T_{ci}(n);\ T_R(CN) = \max_{n \in N} T_r(n)$$

Definition 3.8 (Sequential execution time) *The sequential time of an execution, $T_{SE}(CN)$, is defined to be the sum of the processing times of all compute nodes in the computational network:*

$$T_{SE}(CN) = \sum_{n \in N} T_p(n)$$

The sequential execution time approximates the execution time of a sequential version of the computation. This approximation is in general not entirely accurate because communication in a parallel execution is in general replaced by memory accesses plus extra control statements in a sequential execution, but we have to make a trade off between accuracy and abstractness. The introduced error is acceptable as long as the measures defined above and in the remainder provide a good basis for concurrency analysis. Our experiments confirm that the accuracy is sufficient.

4 Concurrency Model

A computational network that realizes a computation has certain concurrency properties. The goal of the concurrency model is to quantify the presence of the different concurrency properties in a computational network. Concurrency 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 and by the communication within the network. It can also be influenced by the compute platform or the used (run-time) scheduler. This (run-time) scheduler assigns the nodes to processors on which they execute. We are interested in the concurrency properties that are determined by the computational network itself and not its implementation environment, because we want to have a computational network that has good concurrency properties for many environments in which it may operate. Only in a later design phase, we propose to consider the environment and fine tune the concurrency in the computational network to this environment.

However, this phase is not considered in this paper. To leave out the effects caused by the implementation environment, we assume that a compute node can execute as soon as the required data becomes available and furthermore that nodes do not have to wait for data on the input ports of the computational network. Using logical time, these assumptions can easily be realized.

An important goal of the concurrency measures is that they provide a global direction when optimizing the concurrency. To realize this, all measures are normalized to the range $[0, 1]$, in which a value of 1 means that the measured concurrency property is optimal and a value close to 0 means that it is very bad. The measures should besides the global direction also provide enough detail to find the concurrency bottlenecks. For this reason, a detailed measure per node is defined. We next introduce all five concurrency measures and motivate their applicability. We end the section with a short discussion on how the different measures can be used by a system designer, and why all five measures are necessary.

Computation load. In a parallel execution, we want to minimize the overhead of communicating data between nodes. The nodes should spend as much time as possible on computation and not on communication. The time spent by a node on the computation is expressed in the computation time. The time that a node spends on both the computation and the communication is expressed in the processing time. The ratio between computation time and processing time should be as high as possible for every node, as computation, i.e., data transformation, is the main goal of every computational network. These observations lead to the first concurrency measure, the *computation load*.

Definition 4.1 (Computation load) *The computation load of computational network CN and a compute node $n \in N$ are defined as follows:*

$$CompLd(CN) = \frac{\sum_{n \in N} CompLd(n)}{|N|};\ CompLd(n) = \frac{T_c(n)}{T_p(n)}$$

The computation load of the network serves as the global measure; the computation loads of the nodes serve as the detailed measure. The nodes with low computation loads may point to concurrency bottlenecks.

Example 4.1 We calculate the computation load of the computational network of Figure 2 with the event diagram of Figure 3. Let *ECN* denote the network. The computation time and processing time of the different nodes are found using Definitions 3.5 and 3.3 and the event diagram of the computation.

$$CompLd(ECN) = \frac{1}{5} \cdot \left(\frac{1}{4} + \frac{2}{6} + \frac{1}{3} + \frac{2}{4} + \frac{8}{10} \right) = \frac{133}{300} \approx 0.44$$

Thus 44% of the total processing time is spent on meaningful computation. Node a has the lowest computation load, namely 1/4. To improve this, the node should be assigned a larger computation task, or it can be merged with another node.

Processing load. A compute node is during an execution either busy, performing events, or it is idle. It can be idle because it is waiting for data or it has finished its processing but other nodes have not finished executing. To get a balanced workload over nodes, we must balance the processing and run-times of the different nodes. This is important to optimize streaming behavior. To get a notion of the workload balance, we consider the ratio between the processing time and the run-time. The second concurrency measure, *processing load*, looks at this aspect.

Definition 4.2 (Processing load) *The processing load of computational network CN and of a compute node $n \in N$ are defined as follows:*

$$ProcLd(CN) = \frac{\sum_{n \in N} T_p(n)}{|N| \cdot T_R(CN)};\ ProcLd(n) = \frac{T_p(n)}{T_r(n)}$$

For individual nodes, the processing load computes the ratio of the processing time and run-time. In other words, it calculates the ratio between the time that a node is busy and the time that a node is either busy or waiting before it can continue processing. For the network, we assume (ideal) streaming behavior and we only consider the event diagram of a (small) part of the actual execution, typically the events caused by one or a few inputs to the network. Each node can start operating on a next input to the network with a rate that is determined by the node with the longest run-time. Therefore, the processing load measure of a network must not consider the ratio of the processing time and run-time per node, but compare the processing time of the nodes to the run-time of the network. The bottlenecks in obtaining a better processing load are the nodes with the lowest processing load and the node with the longest run-time.

Example 4.2 We continue with our running example. To calculate the processing load of computational network *ECN*, we need the maximum run-time of the nodes in the network. Node e requires 10 logical clock values from the logical clock value at which it starts processing. The other nodes require fewer logical clock values. The processing load is then equal to:

$$ProcLd(ECN) = \frac{4 + 6 + 3 + 4 + 10}{5 \cdot 10} = \frac{27}{50} \approx 0.54$$

The processing load for the individual nodes is 1 for nodes a, c, d and e, and $\frac{6}{9} = \frac{2}{3}$ for node b. The processing load of the network indicates that the nodes in the network are on average 54% of their time busy with computation or communication and 46% of their time idle. Potential points for improvement are nodes b (lowest processing load) and e (longest run-time). The fact that almost all nodes have a processing load of 1 whereas the overall processing load is only 0.54 indicates that the workload balance over the nodes is bad and that e is the most serious bottleneck, which brings us immediately to the next measure.

Restart interval. The compute node with the longest run-time is determining the rate at which new computations can be started in the computational network. This node plays an important role in the throughput of the computational network. The throughput is an important property when a system designer is designing a streaming application. To get a notion of it, we introduce the *restart* measure through Definition 4.3. In general, the closer the restart measure comes to one, the higher the throughput realized by the computational network. However, note that the best restart does not guarantee the best network. Generally, good values for the restart can be obtained through very fine-grained compute nodes. However, this gives communication overhead (and possibly scheduling overhead). The restart measure should therefore be balanced with other measures. Restart is an abstract notion of throughput; it is not equal to it.

Definition 4.3 (Restart) *The restart of computational network CN and a compute node $n \in N$ are defined as follows:*

$$Restart(CN) = \frac{1}{T_R(CN)};\ Restart(n) = \frac{1}{T_r(n)}$$

The restart measure partly overlaps with the processing load, as they both point to the node with the longest run time. However, the restart measure is more fine-grained as it may point to a set of nodes which have a long run-time. The processing load points only to the node with the longest run, ignoring other nodes with a long run-time that are also potential throughput bottlenecks.

Example 4.3 The restart value for our example network *ECN* is 1/10 with node e being the bottleneck node with the lowest restart value. One issue needs explanation. Consider two networks CN_1 and CN_2 that realize the same computation. The maximum run-time of the nodes in CN_1 is 1000 and 100 in CN_2. The restart for CN_1 is 0.001 and for CN_2 0.01. Looking at the absolute values, it is difficult to see that CN_2 is much faster than CN_1. This relative difference in restart interval must be made visible when comparing different solutions for the same application. This can be done by normalizing the values of the restart measure over a set of designs with the largest value. The disadvantage of this approach is that the 1 value for the best network (CN_1 in the example) may suggest that the restart value is optimal whereas this is obviously not always the case. Nevertheless, we choose this solution in our design optimization method discussed later in this paper in order to make relative differences visible.

Synchronization. A parallel computation will in most cases be faster than a sequential implementation of that computation. This is often in the literature referred to as speed-up [MMV98]. The realized speed-up depends on the synchronization that is required between the different nodes in the network, the introduced communication overhead, and how well the computation is balanced over the different nodes. The second and third aspect are covered by the computation load and processing load respectively. The influence of synchronization is not yet fully captured in the measures so far, although a poor synchronization does affect the processing load. 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. Synchronization determines in this way the time that a computation will take in a computational network (see Definition 3.4).
Our concurrency measure, *synchronization*, is related to the speed-up. 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 our objective that a value of 1 for a measure indicates a good solution from the concurrency point of view. Synchronization measures for nodes are not really meaningful. Due to communication overhead, individual nodes are usually slower in a parallel execution than in a sequential execution. For diagnosing synchronization bottlenecks, we can use event diagrams instead. In Figure 3, for example, the synchronization pattern between the nodes b and c causes idle time that may be removable.

Definition 4.4 (Synchronization) *The synchronization of CN is:*

$$Sync(CN) = 1 - \frac{T_E(CN)}{T_{SE}(CN)}$$

Assuming that an execution performs at least one event, the range of values for this measure is in $(-\infty, 1)$ where a value close to 1 indicates a good solution and 0 a solution that is as fast as the sequential execution. Negative values indicate that the execution is slower than a sequential execution. The fact that negative values are possible is not really a problem with respect to our goal that values should be in the range $[0, 1]$. Negative values will be rare and can easily be removed by taking the maximum with 0; it is more important that the optimum value is close to 1.

Example 4.4 To compute the value for the synchronization measure of computational network *ECN* of Example 4.1, we need the execution time of the computational network and the sequential execution time. Figure 3 shows that the execution time equals 15 logical clock values. The sequential execution time is found using Definition 3.8 and is equal to 27. The synchronization is then:

$$Sync(ECN) = 1 - \frac{15}{27} = \frac{12}{27} \approx 0.44$$

We can conclude that the parallel execution performs the computation approximately 44% faster than a sequential implementation of the computation.

Structure. The previous measures consider the event diagram of an execution of a computational network with a given input. The structure of the network plays only an implicit role. The structure itself can already provide insight in the synchronization constraints and potential bottlenecks in the network. It reveals the chains of compute nodes that belong to the different parts of the computation taking place in the network. In other words, it reveals the different data-streams that are processed in the network. If many different data-streams go through one node, then this node may be a synchronization bottleneck for those data-streams. A measure is needed to quantify this concurrency property. Many parallel data-streams can be a sign of good utilization of data parallelism.
A data path through a network is a sequence of nodes from a network input to a network output. In the presence of cycles (feedback loops), there are infinitely many such paths. Therefore, we restrict our paths to go through at most one feedback loop. This prevents grouping of paths with different feedback loops, which are in fact different data-streams, in one path. A path p_1 is called a *sub-path* of p_2 if the nodes on path p_1 are a subset of the nodes on path p_2. Path p_2 is in that case called a *super-path* of p_1. The paths that are present in a computational network can be grouped into *computational paths*.

Definition 4.5 (Computational path) *A computational path is defined as the tuple (u, u', C) with u and u' respectively an input node of the network and an output node of the network and C a set of paths. For every path $\langle v_0, v_1, v_2, \ldots, v_k \rangle \in C$, it holds that $v_0 = u$ and $v_k = u'$. For the set of paths C, the following must hold:*

1. *For each $p_1, p_2 \in C$, 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);*

2. *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.*

Note that requirement 1 in the above definition excludes the possibility that two feedback loops are part of one computational path; requirement 2 implies among others that it is not allowed to skip feedback loops.
The computational paths in a computational network represent the different data flows that go through the network. Exploiting parallelism implies that it is tried to maximize the number of different data flows. They must share as little compute nodes as possible to avoid synchronization bottlenecks. This observation leads to the definition of the *structure* measure. The measure is zero if all computational paths go through all nodes, which implies that there is no structural parallelism in the structure. This is for example the case for a pipeline structure. A value close to one indicates that the structure of the computational network is very parallel. The bottleneck for the structure are thus the nodes through which the most computational paths go.

Definition 4.6 (Structure) *The structure of computational network CN and a compute node $n \in N$ are defined as follows:*

$$Struct(CN) = \frac{\sum_{n \in N} Struct(n)}{|N|}$$

$$Struct(n) = 1 - \frac{|comp.\ paths\ through\ n|}{|comp.\ paths\ in\ CN|}$$

Example 4.5 We continue with our running example. The network ECN contains three paths: $p_1 = \langle a, b, d \rangle$, $p_2 = \langle a, b, c, b, d \rangle$, $p_3 = \langle a, e \rangle$. These paths can be grouped in the computational paths $cp_1 = (a, d, \{p_1, p_2\})$ and $cp_2 = (a, e, \{p_3\})$. Through node a go two computational paths,

whereas all other nodes belong to one computational path; thus node a is a potential bottleneck. The structure for the overall network is then:

$$Struct(ECN) = 1 - \frac{2+1+1+1+1}{2} \cdot \frac{1}{5} = 1 - \frac{6}{2 \cdot 5} = 0.4$$

So, nodes perform on average transformations to 60% of the data-streams in the computational network.

The system designer can use the concurrency model to get feedback about the relevant properties of the specified system. We will now briefly discuss on how a system designer can use the various measures introduced in this section. A system designer will in general be concerned about the latency and throughput of a system. The latency is measured in the computational-network model in the execution time. The latency relative to a sequential solution is captured in the synchronization measure. To minimize the latency, one should try to optimize this measure. 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. In a multi-processor context, a system designer has to divide the application over multiple processors. The objective is to balance the workload and to minimize the overhead of communicating data between the different processors. These properties are expressed in the concurrency model with respectively the processing load and the computation load. Applications may contain data-parallelism that can be exploited in the system. The structure measure points to places in the application where potentially a gain in data-parallelism can be made.

As the above discussion illustrates, the different concurrency measures analyze different concurrency properties. Omitting one measure may result in a non-optimal solution. Consider, for example, the situation in which we would ignore the computation load. Assume now that the nodes in the network are split into a set of nodes in which each node contains only a single internal event and the required read and write events. The solution would have a very good restart measure and synchronization measure. Depending on the data-dependencies in the application it will also have a good structure measure. Typically most internal event have almost the same duration - i.e. the processing load of the network will also be good. So, these measures all indicate that this is a good solution. However, this solution is not good as there is an enormous communication overhead. This overhead is indicated by the computation load. Similar cases can be constructed for all other four measures. None of the five measures can be omitted from the model without introducing the risk that a concurrency optimization process ends in some local optimum, focusing too much on one or a few concurrency aspects.

5 Prototype implementation

5.1 Computational-network model

The computational-network model introduced in Section 3 captures the core of parallel (streaming) applications. It abstracts from a number of different models of computation known from literature. Among them are Kahn process networks and Dataflow graphs. For our implementation of the computational-network model, we use an existing implementation in C++ for Kahn process networks, namely YAPI [dKES+00]. This allows the reuse of code that is written for YAPI. The restriction is that we currently can only simulate computational networks described in YAPI. The coupling with YAPI is however very loose because the analysis tools to determine concurrency measures and the graphical user-interface are independent of YAPI. This implies that any framework that allows modeling of our computational-network model can be used with our analysis tools.

5.2 Time-Stamping mechanism

Section 3.2 defines the execution of a computational network using a time-stamping mechanism based on Lamport's logical clocks. To implement this time-stamping mechanism, we need a duration function that associates a duration with the communication of data over a connection and with each event that occurs in the compute nodes. The duration function is implemented in the following parts:

Internal events. The time-stamping mechanism should allow reasoning about causality and some timing aspects on a relatively high level of abstraction without referring to implementations/physical time. Therefore, in our implementation, we define the duration of internal events (C++ statements) as the number of instructions needed to execute these events on a processor using a standard compiler. We assume that the influence of a specific instruction set does not have too much influence on the results. Our first experiments show that the proposed notion of time is both accurate and abstract enough to perform optimizations independent of the exact processor chosen from a class of processors. As an alternative, we allow to do statistical analysis over a number of compilers/instruction sets taking for example the average number of instructions as the duration.

Read/write events. The duration function for read/write events must assign a duration to each read or write event that represents in some way the time required to communicate the data to the connection. In other words, it represents the actual time needed to call the read or write function and to transfer the data to the connection interface. This can be seen as a linear function $ax+b$ with x the number of data elements communicated. The constant b approximates the time needed to call the communication primitives; the constant a approximates the time needed to read/write one data element.

Connections. The duration function for connections is implemented as a linear function $cy+d$ with y the size of the data elements going through a connection. The constant d approximates the access time of the communication medium; the constant c approximates the time needed to transport one data element. This linear function models all relevant aspects of communicating data over a connection. One may only say that sharing of communication resources is not taken into account. However, this does not need to be taken into account if we have reserved connections, e.g., independent virtual connections multiplexed over one physical connection. Alternatively, at the targeted abstraction level, one can take the average penalty introduced by sharing into account in the constants of the linear function.

5.3 CAST

The previous section has introduced a concurrency model that allows analysis of five different concurrency properties of a computational network. The concurrency model uses the structure of the computational network and the event diagrams that can be obtained by executing the network. To construct an event diagram, a list of all events that occur in all nodes during an execution with a given input is needed. The definition of a compute node leaves the possibility open that the behavior of a node is data-dependent. This implies that there need not to be a single, unique event diagram for a computational network. For Kahn process networks, an event diagram is unique for a combination of a computational network and input. So, to construct an event diagram, the computational network must be executed (i.e. simulated) with an input. Simulating the network does not contradict with our desire for abstraction. To allow abstraction from a single input, we can use multiple simulations and perform statistical analysis on them.

The software tool CAST can be used to compute the concurrency measures of a computational

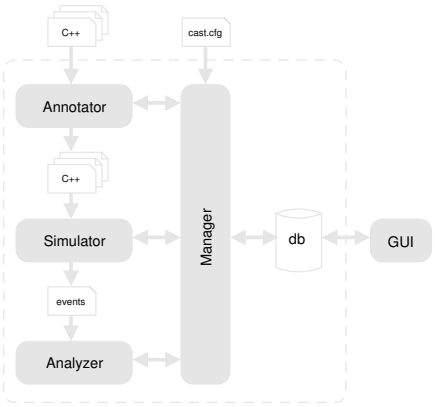


Figure 4: Overview of CAST.

network. It can create an event diagram for a network and given input and analyze this. The tool can also perform statistical analysis on the results of multiple simulations. The overview of CAST is shown in Figure 4. The core of the program consists of three components (i.e., annotator, simulator and analyzer). Furthermore, there is a manager which acts as an API to the database used to store all analysis results and there is a graphical user-interface to present the analysis results to the system designer. All components are described below in some detail.

Manager. The manager component provides the annotator, simulator and analyzer with a simple, uniform interface to the information contained in the database. This database contains the settings of the program, the name all C++ files used and all analysis results.

Annotator. The annotator is responsible for annotating the original source code of the computational network with functions for logging the individual events to a file when the network is simulated. The annotator also adds source code for tracing the events and extraction of the network structure. This source code forms the CAST run-time environment. Finally, we use the annotator to map each internal event (C++ statement) onto a duration as described in Section 5.2. Details are specified in a CAST configuration file.

Simulator. The simulator takes the annotated C++ source code from the annotator component and compiles it into an executable. The actual simulation of the computational network involves now running the executable with the correct input. During this simulation, the CAST run-time environment creates a trace of all events that occur in all compute nodes. The read and write events are always logged individually. Internal events that occur in one compute node between two consecutive read or write events can either be logged individually or grouped together in one or a few internal events. The grouping allows for abstraction of the exact sequence of internal events that occur in a compute node. It can for instance be used to abstract from the exact internal events that occur when a function call is made, while maintaining the accurate timing information. Furthermore, it can lead to a considerable reduction in the size of the trace and therefore to a speed-up of the simulation. Grouping options can be specified in the configuration file. An event diagram can be constructed from the event trace created during the simulation. The CAST run-time environment extracts also the network graph. The network graph can of course be extracted from the C++ source code by a static analysis, but it is more convenient to extract the graph during a simulation.

Analyzer. The calculation of all concurrency measures is performed in the analyzer. It uses the event trace generated during the simulation step and a description of the network structure, the network graph. Using these two and the settings for the duration function, it maps the

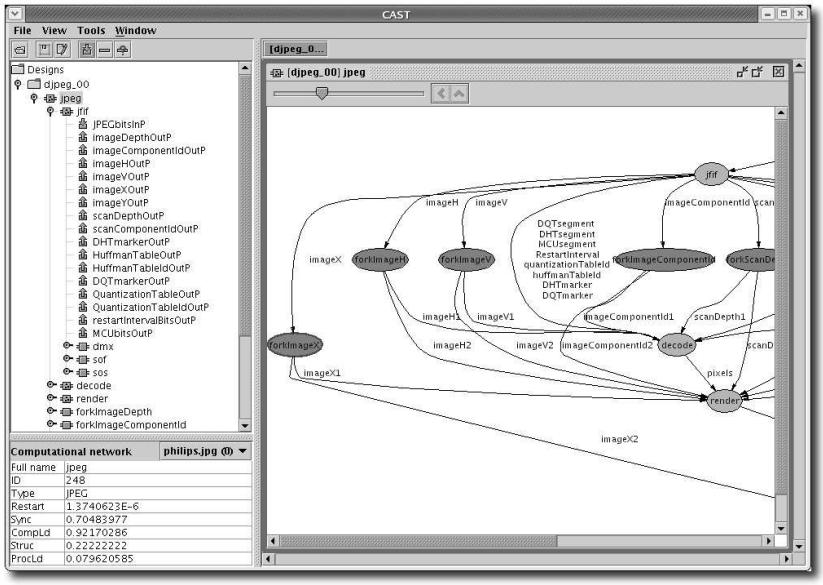
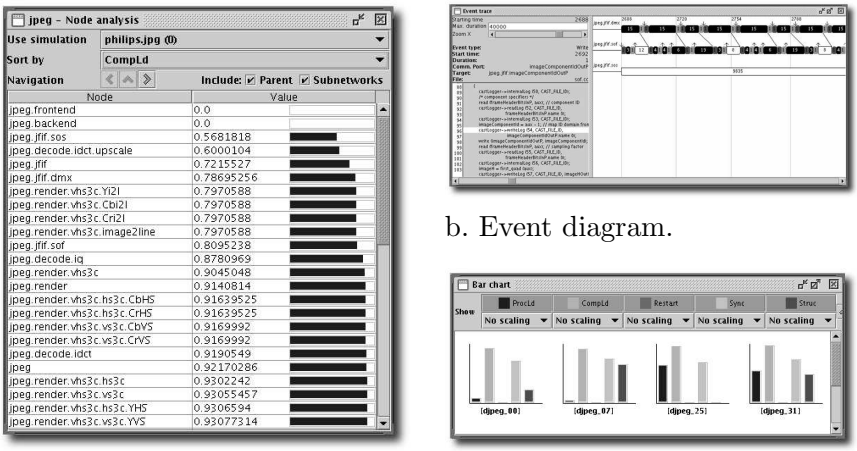


Figure 5: The main window of CAST showing a JPEG decoder.

events onto the appropriate duration and then orders these events according to the causality relations - i.e. it creates an event diagram. After that, it has enough information to compute the values of the different concurrency measures. Note that it is not needed to store the whole event diagram in the memory to compute the concurrency measures. During the construction of the event diagram, the tool must maintain a set of counters that register the times defined in Definitions 3.3 through 3.8. This makes it possible to analyze realistic applications without requiring excessive amounts of memory. For instance, storing the event diagram of an H.263 decoder which decodes a movie of 5 minutes can easily require 8GB of memory space, while storing the counters (independent of the length of the movie) requires less than 10KB.
For typical applications (e.g., JPEG or MPEG encoders or decoders), it is practically impossible to analyze the network for all possible inputs. Therefore, CAST contains a statistical analysis module. With this module, not shown explicitly in Fig. 4, it is possible to select analysis results of different simulations and compute the average, variance, minimum and maximum for each individual compute node and computational network. In this way, we can minimize effects from a specific simulation input, but also effects of a specific instruction set or setting for the duration functions.

Graphical user-interface. CAST contains a visualization software package that helps the designer in understanding the concurrent behavior of the network and identifying potential concurrency bottlenecks by providing a direct method of feedback on the network analysis results. Figure 5 shows a screen-shot of CAST (operating on a JPEG decoder). To identify concurrency bottlenecks, CAST maps the values of the concurrency measures onto the node sizes or node colors. This gives the designer a very clear insight in where the actual concurrency bottlenecks are. The designer can also analyze the concurrency measures of the nodes using bar charts (see Figure 6.a). Furthermore, direct access is provided to parts of the event diagram (see Figure 6.b). The displaying of event diagrams is organized in such a way that even fairly large designs (e.g., more than 10 million events per node) can easily be interpreted by the designer. The events in the event diagram are related to C++ statements in the source code. The graphical user-interface shows this relation by highlighting the C++ statement related to an event when the mouse moves over the event in the event diagram. If desired, the designer can directly open his design environment and edit the source code. The user-interface provides also the option to compare the concurrency measures of different designs in a design-space exploration. This helps



a. Analyze details of nodes.

b. Event diagram.

c. Compare different solutions.

Figure 6: Screen-shots of CAST.

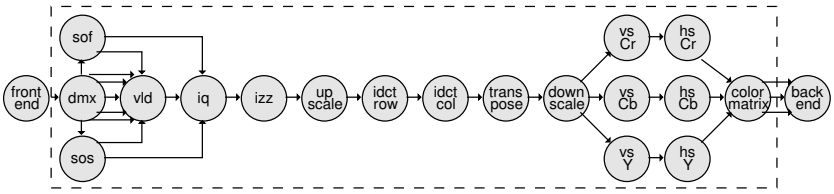


Figure 7: A JPEG decoder (Design 0).

the designer to compare the impact of different design decisions (see Figure 6.c).

6 Design-space exploration

A prototype concurrency-analysis tool should not only implement the computational-network model and the concurrency measures, but it should also provide support for design-space exploration, i.e. , it should support the designer in finding a computational network with optimal (good) concurrency properties. This section presents a generally applicable design exploration method consisting of four steps which is used in conjunction with the concurrency model to realize this. The different steps of the method are explained by deriving in a structured way an implementation of the JPEG decoder [ITU92] that has a balanced workload and good communication behavior. The basic idea of the design exploration method is to first extract all the available concurrency in an application and then design a network optimally exploiting this concurrency.

Starting point. An experienced designer of Philips research optimized the JPEG decoder for a given multi-processor architecture [dK02]. We started with the same computational network

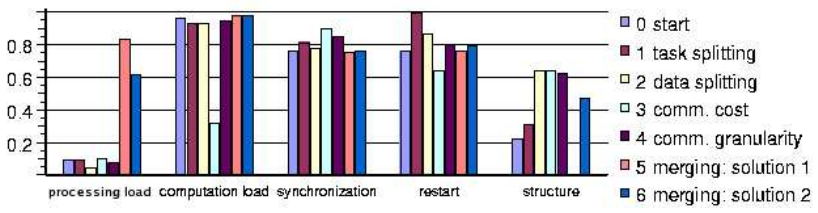


Figure 8: Concurrency measures for the JPEG decoder.

as this designer used. Using the same starting point gives us a fair comparison between the end results. The computational network of this JPEG decoder is shown in Figure 7 and is referred to as design 0. The frontend and backend nodes model the environment of the network we want to optimize. These nodes are not taken into account in the analysis. The details of JPEG are not relevant for the remainder. Concurrency measures for this design are calculated using the statistical option of CAST and a set of five different images. Figure 8 shows the results for design 0 of the JPEG decoder, as well as for some other designs discussed further-on.

Task splitting. The design exploration method starts with task splitting. The goal of this step is to extract the available task-parallelism from the application by splitting compute nodes as far as possible. The task splitting step must optimize the restart measure. The candidates for improvement in this step are the compute nodes with the lowest restart. We selected and modified the four nodes with the lowest restart in design 0. The resulting design is labeled design 1. Figure 8 shows the result of the transformation on the concurrency measures. It shows that the restart has indeed improved. Observe that the absolute restart values computed during the exploration are all very low. Figure 8 shows normalized values as explained in Example 4.3.

Data splitting. The data-splittings step aims at extracting coarse grained data-parallelism in the application. The amount of data-parallelism in the network is visible in the structure measure; this step should improve this measure. The nodes considered in the transformation are the bottleneck nodes of the structure measure. Figure 7 shows that the data is processed in three parallel streams (i.e. three color components) between the *downscale* and *color matrix* nodes. It is possible to create these parallel streams already after the *vld* node. This will increase the data-parallelism. The *vld* node has to process the bitstream sequentially. Hence, it contains no data-parallelism. In the JPEG decoder of design 1, we created separate computational paths for the three color components (design 2). The concurrency measures (see Figure 8) show that the goal of this step is realized.

Communication granularity. While extracting concurrency from an application, cost of communication is not relevant. However, the costs of communication will play an important role in the granularity of communication used in the final implementation. We observe that calling a function that implements the communication primitives is more expensive than a normal memory operation that is part of a normal internal event. To respect this observation, we assign a constant delay of 30 logical clock values to each read/write event. Note that the exact costs are not important; they must only respect the above observation. The statistical analysis function of CAST has been used to verify the concurrent behavior for different cost functions, showing that a delay of 30 is reasonable. Other values, possibly depending on the size of events, do not really affect the relative values of the measures.

The impact of the costs of communication on system performance can be seen in Figure 8. Design 3 is the same network as design 2 but taking into account costs of communication. The computation load has dropped significantly; the nodes are most of their time busy with communication. This effect must be reversed by optimizing the granularity of communication. This results in design 4. The nodes in the JPEG decoder communicate no longer single pixel values, but blocks of pixels at once. Figure 8 shows that a good granularity of communication can almost completely compensate for the introduced costs of communication.

Merging. In the merging step, we combine nodes to remove some of the available parallelism to obtain a more balanced workload, i.e., a processing load close to one. To optimize processing loads of individual nodes, we also have to remove synchronization bottlenecks (using event diagrams). There are three solutions possible to realize a balanced workload for the JPEG

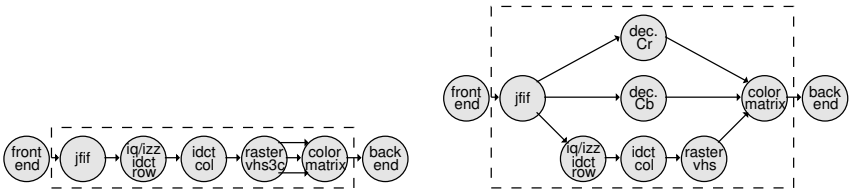


Figure 9: JPEG decoder (Design 5 and Design 6).

decoder. They differ in the amount of data-parallelism that is preserved in the final solution. First, we can remove all data-parallelism and then remove some of the task-parallelism (solution 1). This results in the computational network shown on the left in Figure 9 (design 5). Another solution is to preserve all data-parallelism and remove only the functional task parallelism, design 6, shown on the right in Figure 9. The third solution would be to remove some of the data-parallelism, but not all. This third solution is not further explored in this case study; we focus on the two extreme solutions. Figure 8 shows that both designs 5 and 6 meet the goal of the merging step, namely a high processing load. The structure measure shows that design 6 still contains data-parallelism, while design 5 does not. Both designs have similar values for synchronization and computation load. Synchronization does not play an important role in this case study because of the regular communication patterns. Figure 8 shows that all seven designs give a similar result when compared to a purely sequential version. However, this does not mean that they are all equally good. The synchronization measure can be interpreted as the speed-up which can be achieved if maximal parallelism can be exploited, for example by using one processor per compute node. So, designs 5 and 6 are equally fast when compared to the sequential solution, but 5 may need more resources. The latter is also visible in the lower processing load for design 6. Note that the precise resource usage and throughput in an implementation depend on mapping and scheduling decisions. As we will see in the next section, designs 5 and 6 perform equally well when dynamically scheduled on a homogeneous multiprocessor. The nodes in design 5 have less idle-time during the execution than the nodes in design 6. Design 6 has however a higher restart measure. This implies that it may have a higher throughput than design 5 in an actual implementation.

7 Case Studies

In this section, we present three case studies in which CAST and the design-space exploration method are used to derive in a structured way a computational network with optimal (good) concurrency properties.

7.1 JPEG decoder

The actual design-space exploration of the JPEG decoder was performed in the previous section. The concurrency measures of the resulting solutions, shown in Figure 8, indicate that the designs 5 and 6 are good solutions. To verify this, we mapped these solutions and a reference solution on a multi-processor platform.

As already mentioned, in [dK02], a JPEG decoder was implemented on a the CAKE architecture [SH01]. More specifically, it was mapped onto a single tile of the CAKE multi-processor architecture. A tile consists of a set of processors and memories that communicate through a snooping interconnection network. All processors in the tile operate on a single queue of runable tasks. A small operating system, called the tile run-time system, dynamically assigns tasks to processors. The YAPI library has been implemented in software on top of this tile-run-time system. As in [dK02], our experiment uses a single tile with a homogeneous structure of MIPS

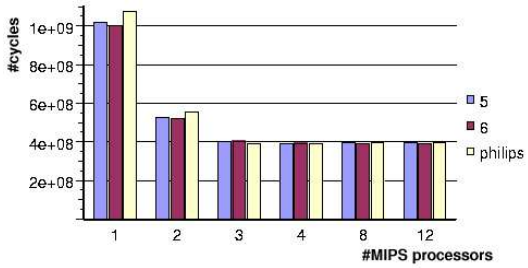


Figure 10: Mapping on CAKE.

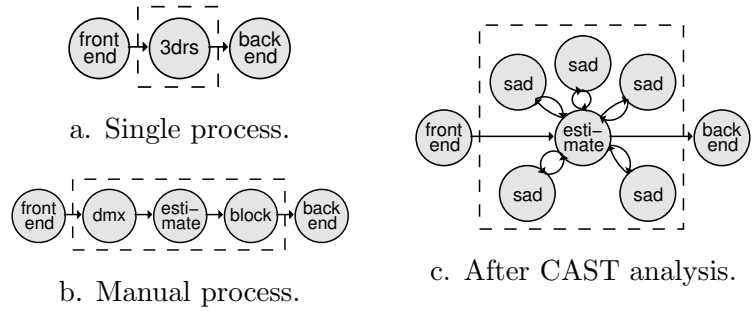
processors and four memory banks to implement the memory space.

To compare the performance of the designs 5 and 6 found in our case study and the Philips case study of [dK02], we simulated these designs for different numbers of MIPS processors. The results of these simulations are shown in Figure 10. The figure shows that the solutions derived in our case study have the same performance characteristics as the Philips design, which is a good result considering that our analysis and design-space exploration has been done independent of the CAKE architecture. Our designs are slightly faster when one or two MIPS are used. The Philips design has the best performance when 3 MIPS are used. However, the difference is not really significant. Design 5 and 6 have similar performance for three or more processors.

7.2 3D Recursive Search

This section presents a second case study that demonstrates the effectiveness of the concurrency model; it also shows the need for a higher level of abstraction than cycle-accurate simulations. The case study implements a parallel version of a sub-pixel accurate motion estimator using a 3D recursive search algorithm (3DRS). The case study was performed by a student during a master's project. The student started with an implementation of the 3DRS algorithm written in C. The first step involved separating the actual algorithm from the code that is needed to simulate the environment (e.g., read and write files to disk). This resulted in the computational network shown in Figure 11.a. The whole 3DRS algorithm is implemented as sequential code in a single compute node. The student then parallelized the algorithm by hand. This resulted in the computational network shown in Figure 11.b, which was considered optimal by the student. Analysis of the concurrency properties using CAST showed that the motion estimator, node *estimate*, is a bottleneck. A study of this node revealed that the motion estimator has to compute five sums-of-absolute differences (SADs) on the same data-set. These computations can be performed in parallel. The CAST analysis showed also that the *dmx* and *block* nodes should be integrated with the *estimate* node after extraction of the SADs. These changes were implemented by the student and led to the design shown in Figure 11.c. The concurrency measures for the three different solutions are shown in Figure 12. The measures indicate that the largest gain is achieved in the change from the manual solution to the solution found using CAST.

To evaluate the different designs we wanted to map them on a single tile of the CAKE multiprocessor architecture and measure the achieved speed-up of the various parallel implementations. This requires that after mapping the code onto the architecture, we simulate the whole system with a movie of which the motion is predicted. For this, we used a movie that contains a sequence of six frames. Simulating one design takes 4 hours on a single 1GHz Pentium III processor with 4GB of internal memory. The counters used in the cycle-accurate simulator of the CAKE architecture turned out to be too small to hold the actual cycle-count. This made it impossible to obtain, in this way, performance measures for our designs.



a. Single process.

b. Manual process.

c. After CAST analysis.

Figure 11: 3D recursive search network.

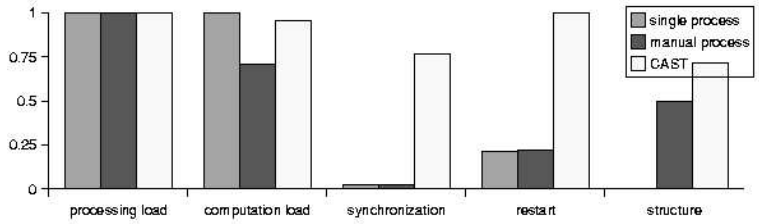


Figure 12: Concurrency measures for the 3DRS implementations.

To get a notion of the speed-up, we can also use the output produced by CAST. CAST computes the execution time of the computational network (see Definition 3.4). This time is equal to the time needed to execute the computational network on a multi-processor system if each node is mapped onto a different processor. In other words, we can use this time as an estimate of how long the computational network will run on a system that contains as many processors as there are nodes. Since the times computed by CAST are based on instruction counts, they are quite accurate. This gives us an estimate of the execution time of the three solutions when they are mapped onto respectively one, three and six processors. We can also calculate the time that the system will need when it is executed on a single processor; this time is equal to the sequential execution time (see Definition 3.8). This gives us an estimate of the execution time of the three solutions when they are mapped onto a single processor. An estimate of the required execution time for a system that contains more than one processor, but less than the number of nodes in the network can also be made. This is required for the design shown in Figure 11.b when it is mapped onto two processors and for the design shown in Figure 11.c when it is mapped onto two, three, four or five processors. For the design shown in Figure 11.b, it holds that the execution of the node *estimate* takes more time than the time needed to execution the *dmx* and *block* nodes. So, it seems logical to map *estimate* on one processor and the other two nodes to the second processor. The execution time of *estimate* can then be used as an estimate of the execution time (after a brief initialization phase) when this design is mapped onto two processors. For the design shown in Figure 11.c, it holds that as many SAD nodes can run in parallel as there are processors in the system. Part of the computation of the motion estimator node can in principle run in parallel with these nodes. But for our estimation, to be on the safe side, we assume that it can never run in parallel with the SAD nodes. The execution time of the system is then equal to the run-time of the motion estimator node and the run-time of a single SAD node multiplied by the number of SAD nodes that must run in series.

To obtain all these time measures, we simulated the designs with our movie and analyzed the execution using CAST. This required approximately 5 minutes for each design. The resulting performance numbers normalized with respect to the execution time for the sequential design (Figure 11.a) are shown in Figure 13. The results show that both parallel implementations

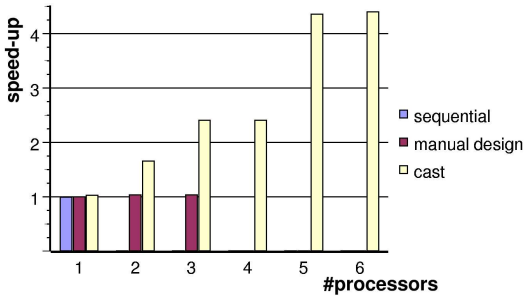


Figure 13: Speed-up of 3DRS implementations.

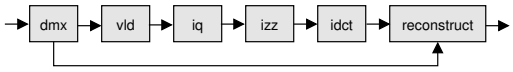


Figure 14: H.263 decoder.

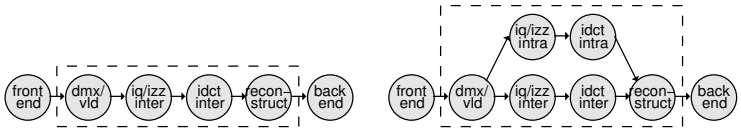
have a speed-up when they are executed on a multi-processor system. They show also that the design found using our concurrency model and CAST has a considerably higher speed-up than the design found by the student. One thing that needs explanation is the execution time of the three designs on a single processor system. One would expect that the single node solution would have the best performance, but our measurements do not show that. This outcome can be explained through the abstract model of communication delay used in our system. The cost of communication in the parallel implementations is more or less the same as the cost of the control code in the sequential implementation. It is hard to say which effect is the most dominant, but the effects are so small that they do not affect the conclusions concerning the speed-ups.

This case study shows that our concurrency model helps in finding and extracting task-level concurrency from an application. The case study shows also that CAST is useful in getting fast and still accurate performance estimates at a relatively high-level of abstraction.

7.3 H.263 Decoder

H.263 is a standard video-conferencing codec optimized for low data rates and relatively low motion [ITU98]. The codec was used as a starting point for the development of the MPEG-II codec which is optimized for higher data-rates. The structure of an H.263 decoder is shown in Figure 14. The H.263 decoder supports three types of frames: I-frames, P-frames and PB-frames. To decode a PB-type of frame, the reconstruct uses the previous and next decoded frame and the already decoded blocks of the current frame. For a P-type of frame, the reconstruct uses the previous decoded frame and the already decoded blocks. For an I-frame, only the already decoded blocks are used. Thus, an I-frame has no dependencies with data from other frames. Note that the fact that I frames are independent makes it possible to process them in parallel with a P- or PB-frame. Further details of the H.263 decoder are not relevant.

A bachelor student, with no background in video coding, was asked to extract the available concurrency from a given sequential C specification of the H.263 decoder. The student had to complete this assignment within approximately 200 hours. First, the student split the decoder into the tasks shown in the block diagram of Figure 14. Using CAST and its graphical user-interface, the student analyzed this computational network to identify potential concurrency. This led to a number of transformations on the network, which were analyzed again using CAST. This process was repeated until two final solutions were found. The first solution implements the decoder as a pipeline of nodes (see Figure 15.a). The second solution exploits the option that I-frames can be processed independent of P- and PB-frames (see Figure 15.b). The concurrency



a. Solution 1. b. Solution 2.

Figure 15: H.263 decoder

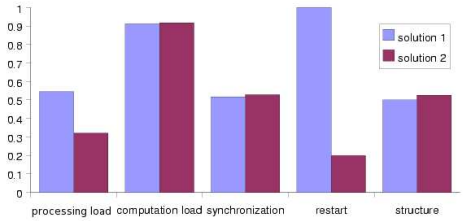


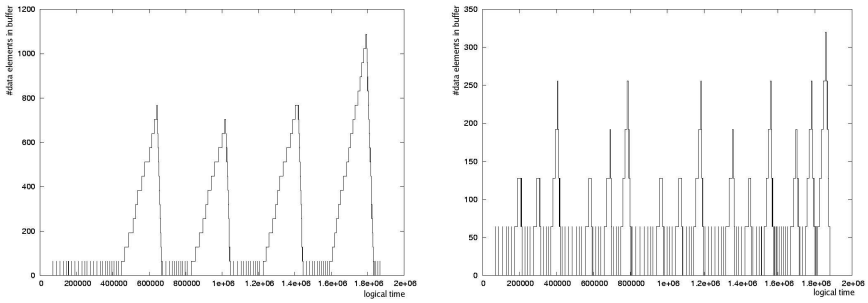
Figure 16: Concurrency measures for the H.263 decoder.

measures for both solutions are shown in Figure 16. The synchronization measure has a value of approximately 0.5 for both solutions indicating that they are approximately twice as fast as a sequential solution. The processing load suggests that solution 2 may need more resources to achieve this. The solutions also differ with respect to potentially achievable throughput, measured by the restart measure. The reason for this is that the merging of the two data streams in the reconstruct node of the second solution adds additional complexity to this bottleneck, making it only slower. Overall, the measures show that solution 1 outperforms solution 2.
Unfortunately, it is impossible to benchmark these solutions using the CAKE simulator as it has problems with the required simulation length. It is also not possible to calculate the speed-up in a similar manner as done in the previous case study. The reason for this is that it is practically impossible to find out when which nodes can execute in parallel. However, the case study does show that a student with no background in the application domain is able to quickly identify different sources of concurrency using our concurrency model and CAST.

8 Discussion

The computational-network model and the concurrency measures neglect most architecture properties. In this section, we discuss options to take several architecture properties into account in the concurrency analysis. The elaboration of this architecture-dependent phase of concurrency analysis is left for future work.

Heterogeneous architecture. The assignment of a duration to an internal event (see Section 5.2) assumes implicitly that a homogeneous platform is used as it uses the same compiler for all nodes to relate their internal events to a duration. In practice, the used multi-processor system may be a heterogeneous system. We see two different solutions to take this heterogeneity into account in our concurrency measures. The first solution assumes that a mapping of nodes to processor types is made. In that case a different compiler for the different node mappings can be used. So, for each node the mapping of an internal event onto a duration is based on the compiler that comes with the processor to which this node is mapped. The second solution would be to use scaling factors for the durations of the internal events of the different nodes. For example, assume that a node a is mapped onto a processor which is twice as fast as the processor to which a node b is mapped. Then the duration of each internal event in b should be multiplied with two to take this difference in speed into account. The second approach can also be used to model the effect of hardware accelerators, i.e., when a node is not mapped onto



a. b.

Figure 17: Buffer size requirements for a connection in the JPEG decoder.

a programmable processor but directly implemented in hardware.

Buffer sizes. The computational-network model assumes implicitly that the connections used between the nodes have infinite size. This implies that the execution of a compute node can never be blocked on a full connection. In practice, a connection will be assigned a buffer of finite size to store data as the amount of available memory in a system is limited. As a result, a node which produces data may have to wait until there is enough space in the connection. In other words, the producing node must wait until the consuming node has read enough data elements from the connection. This dependency between the producing and consuming node will affect the event diagram and thus the concurrency measures. If a separate buffer with fixed size is assigned to each connection, then we can analyze this impact in the following way. During the construction of the event diagram the used buffer space of each connection is counted over time. If insufficient space is available, a node stalls the execution of a write event (i.e. it inserts idle time) till there is enough space on the connection to write the data elements. The resulting event diagram can then be analyzed in the normal way and the impact of the buffer size on the concurrency measures becomes visible. Note that this allows a fast exploration of the effect of buffer sizes on the concurrency properties of a design, without the need for re-executing the application.
It may also be interesting to study the number of data elements that are stored over time in the buffer and to take this information into account in the concurrency optimization. For example, Figure 17.a shows the number of data elements stored in a connection between two of the compute nodes in a JPEG decoder over time. Typically only 64 or 128 data elements are stored in the connection at the same time. However, there are a few points at which many more data elements need to be stored. Furthermore it is clear that the node which produces the data elements does this at a more or less constant rate. So, the increase in used buffer space must be caused by the consuming node. Analysis of the source code of this node revealed that each time after it had received a certain amount of data a transformation was applied on it. While this transformation was executing, the producing node continued filling the connection. This transformation, however, could be performed after a small amount of data was received - i.e. the transformation could be distributed more evenly over the execution time of the producing node. Applying this transformation to the source code of the producing node led to the buffer size requirements shown in Figure 17.b The required buffer size is reduced with a factor three while the concurrency properties of the design are preserved.

9 Conclusion

In this paper, we presented a concurrency model with a supporting design-space exploration method and an analysis tool that allow reasoning about concurrency in streaming applications at the executable-specification level. The model consists of a set of five global measures that

provide guidance when optimizing the concurrency and a set of detailed measures that provide insight in concurrency bottlenecks. The presented examples and case studies show that these measures are meaningful, do not (fully) overlap, allow reasoning about concurrency and are sufficient for obtaining good results. Our method still gives results with analysis times in the order of minutes when cycle-accurate simulations are no longer feasible due to long simulation times and extremely high cycle counts. The JPEG decoder case study furthermore shows that the concurrency model and accompanying design-exploration method allow target-architecture-independent concurrency optimization; when the end result is implemented on a homogeneous system of MIPS processors, the performance is similar to an optimized design implemented on that architecture by an experienced designer. The 3D recursive search case study illustrates that our concurrency model can be useful in getting fast and accurate performance estimates at a relatively high-level of abstraction. The H.263 decoder case studies shows that also non-experienced designers are able to quickly identify different sources of concurrency using our concurrency model and CAST.
Future work includes more experiments with different applications and architectures to verify the assumptions made in the concurrency model and to fine-tune the model. We also plan to study the issue of compositionality and to extend the concurrency model to take architecture information into account for the architecture-dependent step of the design process. Costs of communication may have a large impact on system performance, meaning they must be estimated accurately. We want to study the modeling of these costs in more detail to get a model that provides abstract but accurate information about these costs.

Acknowledgment. We want to thank Erwin de Kock for developing YAPI and giving access to all results of his JPEG case study, Paul Stravers and Jan Hoogerbrugge for providing the CAKE multiprocessor-architecture simulator and Jef van Meerbergen, Henk Corporaal and Johan Lukkien for their comments and suggestions, Jan Ypma for his work on the graphical user interface, Andre Carmo for his experiments with the 3DRS application and Rik Kneepkens for his work on the H.263 decoder.

References

[BHLM94] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4(2):155–182, April 1994.

[BWH+03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavango, C. Paserone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[CHEP71] F. Commoner, A.W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523, October 1971.

[dK02] E.A. de Kock. Multiprocessor mapping of process networks: A JPEG decoding case study. In *ISSS'02, 15th System Synthesis Symposium, Proc.*, pages 68–73. ACM, 2002.

[dKES+00] E.A. de Kock, G. Essink, W.J.M. Smits, R. van der Wolf, J.-Y. Brunei, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference, Proc.*, pages 402–405. IEEE, 2000.

[EEP03] C. Erbas, S. C. Erbas, and A. D. Pimentel. A multiobjective optimization model for exploring multiprocessor mappings of process networks. In *CODES/ISSS'03,*

Int. conf. on hardware/software codesign and system synthesis, Proc., pages 182–187. ACM, October 2003.

[Gri04] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal, Elsevier*, 38(2):131–183, December 2004.

[HL00] L. Hélouët and P. Le Maigat. Decomposition of message sequence charts. In *SAM'00, 2nd Workshop on SDL and MSC, Proc.*, pages 46–60. Irisa, 2000.

[ITU92] ITU. Information technology - digital compression and coding of continuous-time still images. ITU Recommendation T.81, September 1992.

[ITU98] ITU. Video coding for low bit rate communication. ITU Recommendation H.263, February 1998.

[JHS93] K. T. Johnson, A. R. Hurson, and B. Shirazi. General-purpose systolic arrays. *IEEE Computer*, 26(11):20–31, November 1993.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proc.*, pages 471–475. Amsterdam, The Netherlands: North-Holland, 1974.

[KM77] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing '77, Proc.*, pages 993–998. Amsterdam, The Netherlands: North-Holland, August 1977.

[KNRSV00] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[KRD00] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *Int. Conf. on Hardware Software Codesign, Proc.*, pages 13–17. ACM, 2000.

[Kun98] S.Y. Kung. *VLSI array processors*. London, UK: Prentice Hall, 1998.

[Lam77] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1977.

[LP95] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[LSV98] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[LvdWDV01] P. Lieverse, P. van der Wolf, E. Deprettere, and K.A. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI Signal Processing*, 29(3):197–206, November 2001.

[MK03] A. Mihal and K. Keutzer. *Networks on Chip, Mapping concurrent applications onto architectural platforms*, chapter 3, pages 39–59. Dordrecht, The Netherlands: Kluwer Academic Publishers, January 2003.

[MMV98] A. Mazzeo, N. Mazzocca, and U. Villano. Efficiency measurements in heterogeneous distributed computing systems: From theory to practice. *Concurrency: Practice and Experience*, 10(4):285–313, May 1998.

[MPND02] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multigranular simulation. In *Conf. on languages, compilers and tools for embedded systems, Proc.*, pages 18–27. ACM, 2002.

[Pra86] V. Pratt. Modelling concurrency with partial orders. *International Journal of Computer and Information Sciences*, 15(1):33–71, 1986.

[PvdWD+00] A.D. Pimentel, P. van der Wolf, E.F. Deprettere, L.O. Hertzeberger, J.T.J. Eijndhoven, and S. Vassiliadis. The artemis architecture workbench. In *Progress Workshop Embedded Systems, Proc.*, pages 53–62. Utrecht, The Netherlands: STW Technology Foundation, 2000.

[RMN92] M. Raynal, M. Mizuno, and M. Neilsen. Synchronization and concurrency measures for distributed computations. In *12th International Conference on Distributed Computing Systems, Proc.*, pages 700–707. IEEE, June 1992.

[RT93] K. Ravindran and A. Thenmozhi. Extraction of logical concurrency in distributed applications. In *ICDCS'93, International Conference on Distributed Computing Systems, Proc.*, pages 66–73. IEEE, May 1993.

[SH01] P. Stravers and J. Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *International Symposium on VLSI Technology, Systems and Applications, Proc.*, pages 184–187. IEEE, 2001.

[SRV+03] R. Stahl, L. Rijnders, D. Verkest, S. Vernalde, R. Lauwereins, and F. Catthoor. Performance analysis for identification of (sub)task-level parallelism in java. In *Int. Workshop on Software and Compilers for Embedded Systems, Proc.*, pages 313–328. Springer, October 2003.

[ST98] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[TC00] F. Thoen and F. Catthoor. *Modeling, verification and exploration of task-level concurrency in real-time embedded systems*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2000.