Analyzing Concurrency in Computational Networks

(Extended Abstract)

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

Abstract

We present a concurrency model that allows reasoning about concurrency in executable specifications. The model mainly focuses on data-flow and streaming applications and at task-level concurrency. The aim of the model is to provide insight in concurrency bottlenecks in an application and to provide support for performing implementation-independent concurrency optimization.

1. Introduction

Multi-processor systems are rapidly becoming a standard solution for implementing embedded multi-media systems. They provide relatively high compute power at a low energy cost. To exploit the concurrency inherently present in multi-processor systems, the parallelism available in an application mapped onto such a system must be made visible in the mapping and programming trajectory. This abstract briefly presents a concurrency model that allows architecture-independent task-level concurrency optimization in executable specifications. The main focus is on streaming applications. The concurrency optimization leads to a specification that forms a good starting point for mapping the application onto a multi-processor system. The final implementation requires an architecture-dependent step, which is not covered in this abstract. The optimization criteria are inspired by those used in performance analysis but they are targeted towards streaming and concurrency. The novelty is that we perform target-architecture-independent optimization at the executable-specification (source-code) level. The result of this optimization is a specification that can easily be optimized for many different implementation platforms. In other words, our techniques help in making re-usable specifications. For details, see [3].

2. Model of Computation

Our model of computation, the computational-network model, assumes that an application is organized as a hierarchical collection of autonomous compute nodes that are connected to each other by means of point-to-point connections corresponding to data streams. A given node computes on data it receives along its inputs to produce output on some or all of its outputs. The actions performed by a node are modeled as a totally ordered sequence of events, and the actions of a network as a partial order of events. To reason accurately about timing aspects without referring to concrete implementations, we use an adapted version of Lamport's logical clocks [2], associating a delay with events and with communication. Figure 1 shows a computational network with a partial event diagram. The computational-network model is used to model applications for image, video and graphics processing (e.g., an MPEG decoder or a still-texture decoder). It captures the core of parallel (streaming) applications, and nothing more. Well known examples of computational-network models are Kahn process networks and the synchronous data-flow model.

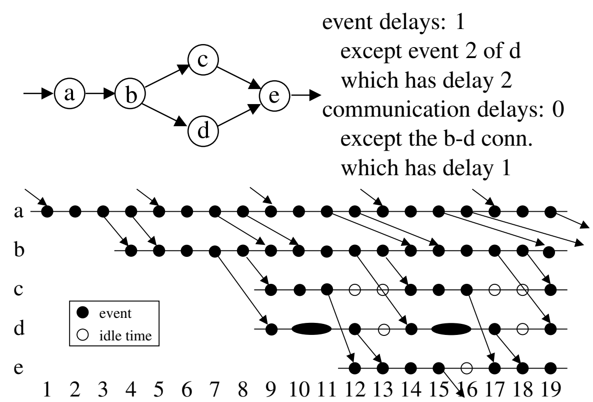

Figure 1. A network with a partial event diagram.

3. Concurrency Model

Our concurrency model aims at performing a target-architecture-independent concurrency optimization. Its concurrency measures abstract from the environment in which a computational network operates, and are calculated from the computational-network structure and an event diagram of an execution. The measures are used in conjunction with a design method that consists of four steps. Each step tries to optimize one different aspect of task-level concurrency; it optimizes one of the concurrency measures, while the other measures are used to balance the overall concurrency optimization. The measures are computed for the network and for the individual nodes in the network. The measures for the compute nodes provide insight into the concurrency bottlenecks. The measures for the network can provide global guidance when optimizing concurrency.

Task-splitting. The compute node with the longest run-time is determining the rate at which new computations can be started in the network. In other words, this node determines the throughput of the network. The throughput is an important property when a system designer is designing a streaming application. The *restart* measure provides an abstract notion of it. To optimize the restart, the slowest compute node must be split in a set of compute nodes with better values for the restart. 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.

Data-splitting. The structure of a network reveals the chains of compute nodes that belong to 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. The more data-streams can be distinguised in a network, the more data-parallelism is present. However, if many different data-streams go through one node, then this node may be a synchronization bottleneck for those data-streams. The *structure* measure is used to quantify this concurrency property. The (task-level) data-parallelism that is present in the specification should be made explicit to optimize this concurrency property.

Communication granularity. 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, as computation, i.e., data transformation, is the main goal of every computational network. The ratio between time spent on computation and time spent on both computation and communication is expressed in the *computation load*. This ratio can be calculated for the network as a whole and for individual nodes and should be as high as possible. The granularity of communication must balance time spent on communication and time that nodes have to wait for input data.

Merging. During an execution, a compute node is either busy, performing events, or it is idle. It can be idle because it is waiting for data or because it has finished its execution while other nodes have not yet finished. To get a balanced workload over nodes, we must balance the execution times (computation plus communication time) and run-times (execution time plus idle time) of the different nodes. This is important to optimize streaming behavior. To get a notion of the workload balance, the *execution load* considers the ratio between the execution time and the run-time. Nodes that have a low execution load must be merged with each other to get a better overall execution load for the network.

A parallel computation will in most cases be faster than a sequential implementation of that computation. This is often referred to as speed-up. The realized speed-up for a computational network depends on the synchronization that is required between the different nodes in the network, the introduced communication overhead, and the balance of the computation over the different nodes. The second and third aspect are covered by the computation load and execution load respectively. The influence of synchronization is not yet fully captured in these measures, although a poor synchronization does affect the execution load. Synchronization is important when considering concurrency, because synchronization is limiting the execution of compute nodes and with that the number of nodes that can run in parallel. Synchronization constraints may impose the restriction that two nodes can only execute in sequence. This concurrency property is captured in the final measure of our model, the *synchronization* measure. The design method does not contain a special step in which this property is optimized. It must be taken into account in all steps.

4. Results

The concurrency model has been tested on, among others, a JPEG decoder. We used the tool CAST [4] that implements our model and that operates on a C++ library for specifying Kahn process networks. The performance of our optimized JPEG decoder, when mapped onto a homogeneous multi-processor platform, turned out to be similar to the performance of a JPEG decoder manually optimized for this platform [1]. The results illustrate that the concurrency model enables architecture-independent analysis of task-level concurrency in streaming applications at the executable specification level.

References

[1] E. A. de Kock. Multiprocessor mapping of process networks: A jpeg decoding case study. In *15th System Synthesis Symp., Proc.*, pages 68–73. ACM, 2002.

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

[3] S. Stuijk. Concurrency in computational networks. Master's thesis, TU Eindhoven, 2002. `http://www.ics.ele.tue.nl/~sander`.

[4] S. Stuijk, J. Ypma, and T. Basten. CAST - a task-level concurrency analysis tool. In *ASCI 2003, 9th Annual Conf. of the Advanced School for Computing and Imaging, Proc.*, 2003.