WWW.SILEXICA.COM

ARTICLE

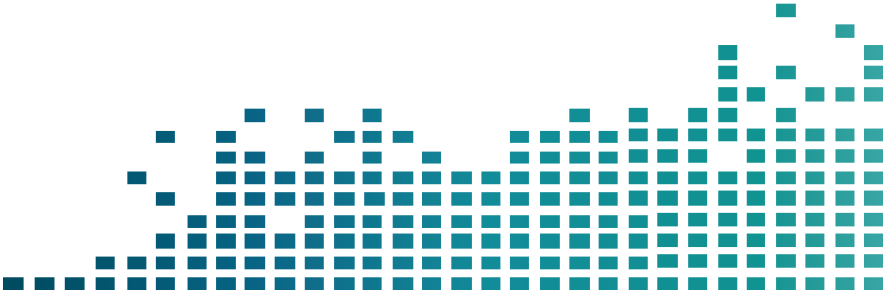
Kahn Process Networks

SILEXICA

multicore meets simplicity

Contents

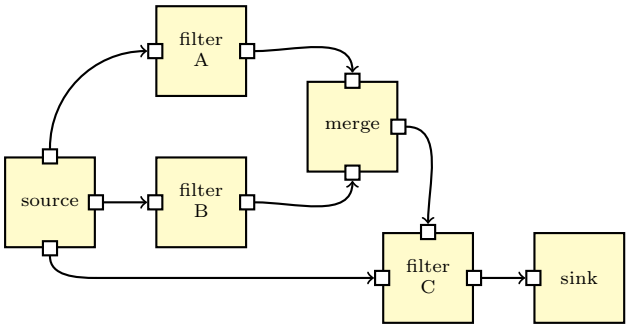
SILEXICA



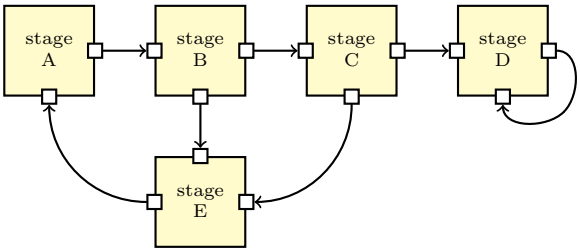
1 Introduction

Modern personal computing devices feature multiple cores. This is not only true for desktops, laptops, tablets and smartphones, but also for small embedded devices like the Raspberry Pi. In order to exploit the computational power of those platforms, application programmers are forced to write their code in a parallel way. Most often, they use the threading approach. This means multiple parts of the code execute at the same time and have access to same set of shared data. However, parallel programming using threads is complicated and hard to debug. When a thread modifies data that might be read by other threads, the programer has to insert proper synchronization. Almost every programmer who has used threads has experienced data races and/or non-deterministic behavior. In such situations, the application might shows a bug or crashes, but once loaded into the debugger, it completes without problems. The reason for such effects is that the timing of thread execution can affect the computational behavior inside the threads. This means that if the timing changes, e.g. due to the OS scheduling the threads in a slightly different way, the threads might behave slightly differently - or in the worst case do something completely different.

The approach of Kahn Process Networks (KPNs) is a parallel programming method that also allows to harness the potential of multi-core systems. It has been invented by Gilles Kahn in 1974. (His original paper can be accessed at: http://www1.cs.columbia.edu/~sedwards/papers/kahn1974semantics.pdf) This programming model avoids the issues of data-races and non-determinism. It ensures that the behavior of the parallel threads of execution — called processes in case of KPN — is never influenced by side-effects like timing. However, the benefits of deterministic and race-free parallel execution come at a small price of a couple of restrictions regarding the communication between the processes. Most important, the KPN processes (not to be confused with OS processes) are not allowed to share data structures. Instead, they communicate among each other by sending data over unidirectional communication channels. Thus, a KPN application can be depicted in form of a directed graph. The nodes represent the processes and the arrows the FIFO channels (see Figure 1.1a for an example). The KPN graphs might be also contain cycles in case the application includes feedback loops (see Figure 1.1b). On the positive side, the determinism of KPNs allows automatic tools to reason about the behavior of the processes. This enables deep analysis of their properties and thus generation of efficient code or hardware from the high-level KPN specification. Examples for automatic code generation based on KPNs are SCADE (http://www.esterel-technologies.com/products/scade-suite/) and SLX (https://www.silexica.com/products/products-overview). Accelize (https://www.accelize.com/technology) uses KPNs to generate efficient FPGA (Field Programmable Gate Array) hardware implementations from KPN models.



(a) An example of a KPN graph.



(b) An example of a cyclic KPN graph.

2 Blocking Reads and Deadlocks

The control flow and the computations on local data inside the KPN processes are not restricted. For example, a process is allowed to call functions, loop over local data and access its input and output channels at any point in the code. Those communication channels work in a first-in, first-out (FIFO) manner. This means that the first data item written into a channel by the sender process is received first by the receiver process. The determinism of the KPN model is enabled by one additional restriction regarding reading from channels: it is not allowed to check if there is data available in an incoming channel without consuming it. If a process decides to read from a channel, execution of this process is blocked if there is no data available. Execution continues only when a data item becomes available in the channel. This restriction is required for deterministic execution of the whole application. The blocking read behavior may lead to deadlocks. A deadlock is a situation in which all KPN processes are waiting for input on an empty channel. In this situation, no process will ever continue to get of the blocking because no process is able to write data to any channel. However, such application bugs are usually easy to find and fix because they appear deterministically. Thus, they can be debugged in a straight-forward way.

The theoretical model of KPN always allows to write a data item to a channel at any time without blocking. This means that the channels provide an infinite amount of buffer space. A real implementation has to deviate from this behavior and limit the amount of space available to a channel. When a channel is completely filled and a process attempts to write to it, execution of this process will be blocked in a similar manner as for reading from an empty channel. Given fixed channel sizes, the application behavior still stays deterministic. However, there might be additional deadlocks due to some processes being blocked on write operations. Those deadlocks are called artificial in contrast to the structural deadlocks in which all processes are blocked on read.

3 Analysis and Code Generation

Determining finite channel sizes for an actual implementation of a KPN application so that no artificial deadlocks occur is a hard problem. It is proven that it cannot be solved by an automatic tool in the general case. However, real KPN applications can usually be analyzed for typical inputs in order to determine suitable channel sizes. For example, the SLX Mapper (https://www.silexica.com/products/slx-mapper) is able to analyze KPN applications for a given set of representative input data. It will compute channel sizes that enable deadlock-free execution. Further it can trade-off memory consumption on the target platform to execution speed. This means it will make the channels larger than the minimum size if this leads to a significant increase in execution speed. The determinism of KPN applications enables an analysis of all KPN processes on all processor cores of the platform. Powerful heuristics inside the tool can determine a mapping of the KPN processes to the processor cores with a high execution speed, low memory footprint or a good tradeoff between the two metrics. The SLX Generator (https://www.silexica.com/products/slx-generator) can be used to generate efficient platform-specific target source code from the KPN application and the mapping information.

4 Summary

The KPN programming model imposes a few restrictions on parallel programming regarding communication between the threads of execution. However, those restrictions results in deterministic execution and avoid data races completely. Further, the determinism enables automatic tools to analyze the application and generate efficient target code.

If you got interested in the KPN programming model, stay tuned to this blog. There will be a follow-up post about a generator of synthetic KPN applications soon.