



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Semester Thesis
at the Department of Information Technology
and Electrical Engineering

Efficiently programming the Intel SCC

SS 2011

Simon Wright

Advisors: Dr. Iuliana Bacivarov
Dr. Hoeseok Yang
Professor: Prof. Dr. Lothar Thiele

Zurich
31st August 2011

Abstract

The Intel Single-Chip Cloud Computer (SCC) is a 48-core microprocessor. The distributed operation layer (DOL) is a framework that allows mapping of applications to multiprocessor platforms. The DOL was extended to allow mapping-dependant code generation for the SCC and a networked Linux cluster. The application is given as Kahn process network (KPN), which consists of processes communicating via FIFO channels. It can be arbitrarily mapped to and subsequently executed on both target platforms. The main contribution is the inter-process communication between processes mapped to different cores (SCC) or workstations (Linux cluster). To this end, a remote FIFO model is presented that separates a remote FIFO channel in to two endpoints. It is shown that deadlocks can occur if multiple such channels share the same physical channel. A solution, using synchronization between the sender and the receiver to control the data flow, is proposed. The model is implemented for the two target architectures. Finally, the performance of the implementation is evaluated with an exemplary application.

Acknowledgements

I would like to thank my advisors Iuliana Bacivarov and Hoeseok Yang for supporting me during this semester thesis. They were always patient and helpful when I ran into problems and helped me to overcome them.

I also want to thank Prof. Dr. Lothar Thiele for allowing me to do this semester thesis at the Computer Engineering group.

Furthermore, I would like to thank Devendra Rai, for his advice and for helping me with the SCC, and Lars Schor, for his help with the DOL and for providing me with the template for this report.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	DOL Framework	2
1.2.1	Application	3
1.2.2	Architecture	5
1.2.3	Mapping	6
1.3	Contributions	6
1.4	Related Work	7
1.5	Outline	9
2	The SCC Architecture	11
2.1	Architecture	11
2.2	Programming the SCC	13
2.2.1	Writing applications	13
2.2.2	Executing applications	14
2.3	Extending DOL to support the SCC	14
3	Remote FIFO Model	17
3.1	Endpoint concept	17
3.2	Channel management	18
3.2.1	Connection per KPN channel	19
3.2.2	Connection per physical channel	19
3.3	Deadlock-free implementation	20
3.3.1	Deadlock situation	20
3.3.2	Flow control to avoid deadlocks	21
4	Implementation	27
4.1	Local and remote FIFO channels	27
4.1.1	Local FIFO implementation	27
4.1.2	Remote FIFO implementation	28
4.2	Remote Channel Management	29

4.2.1	The Remote Channel Connector	29
4.2.2	The Remote Channel Manager	31
4.3	Integration into the DOL	33
4.3.1	Code generation procedure	33
4.3.2	Code generation for an example application	35
5	Evaluation	39
5.1	Test setup	39
5.2	Results and discussion	41
5.3	Future work	42
5.3.1	Improving performance	42
5.3.2	Extension to DAL	42
6	Conclusion	45
A	Toolchain guide	47
B	Acronyms	51
C	Presentation Slides	53

List of Figures

1.1	DOL design flow implementing the Y-chart approach.	3
1.2	Example process network.	4
1.3	Example architecture.	6
1.4	Possible mapping of the example PN on to the example architecture.	7
2.1	High-level SCC block diagram.	12
2.2	Simplified block diagram of a SCC tile.	13
3.1	FIFO channel when two processes are mapped to the same core.	18
3.2	FIFO channel when two processes are mapped to different cores.	18
3.3	TCP/IP channels when establishing a connection per edge in the KPN.	19
3.4	TCP/IP channel when establishing a connection per physical channel.	20
3.5	Implementation leading to a deadlock in certain situations.	22
3.5	Implementation leading to a deadlock in certain situations.	23
3.6	Part 4 of figure 3.5 without data being consumed from the sending endpoint.	24
3.7	Deadlock-free implementation.	25
4.1	Illustration of RCC functionality.	31
4.2	Illustration of RCM functionality.	32
4.3	Simple example PN.	35
5.1	KPN of the application used for the evaluation.	40
5.2	Mapping the application to the SCC.	40
5.3	Evaluation results.	41

1

Introduction

1.1 Motivation

The requirements of modern real-time multimedia and signal-processing applications on embedded systems can no longer be met by traditional single processor architectures which have reached their physical limits. Therefore, such systems are increasingly developed as multiprocessor system-on-chip (MPSoC) designs. Such systems are typically highly integrated, offer high computation power while maintaining a moderate power consumption. However, programming such architectures, to efficiently make use of the given resources, remains a challenge.

Therefore, extensive research is currently done to develop design flows and tools which ease the programming of such platforms and which help designing an ideal MPSoC for certain applications. The design complexity is reduced by raising the level of abstraction and by providing tools to automate the design flow. A few examples are given in the following. The *Daedalus* framework [2] provides an environment to program and prototype multimedia MPSoC architectures. It assumed that the MPSoC is constructed from a set of IP components. Given a sequential application, the framework automatically converts it to a parallel form. This is then used to find a number of promising MPSoC's made from the previously mentioned components. These architectures are automatically synthesized to a FPGA and can subsequently be used for prototyping. A similar approach is taken by the *MAMPS* project [4] and the *Koski* framework [10]. *CoFluent Studio* is a commercial software

for modelling and simulating multiprocessor systems (among other electronic systems) using SystemC. It offers behavioural and performance estimation without requiring embedded software application code or a precise platform description.

All these frameworks/tools have in common that they are based on the so called *Y-chart approach* [7]. It will be discussed in more detail in connection with the DOL framework in section 1.2.

The Intel *Single-Chip Cloud Computer* (SCC) [13] is an example of an MP-SoC. The SCC is an experimental microprocessor embedding 48 cores with an x86 instruction set on a single piece of silicon. Other examples of MP-SoC's include the processors made by *Tilera* [6] (with up to 100 cores!) or the *Cell Broadband Engine* [1] from IBM, Toshiba and Sony (one 64-bit Power Architecture processor core plus eight synergistic processor cores).

1.2 DOL Framework

The *distributed operation layer* (DOL) [21] is a framework developed at the TIK laboratory at the department of electrical engineering at ETH Zurich. The DOL allows programmers to easily make use of MPSoC systems without having a detailed knowledge of their actual architecture. If the programmer writes the application according to certain guidelines (see section 1.2.1), the DOL can automatically generate the code to execute the application on the chosen target architecture. The focus of DOL lies on stream-oriented applications, which are typically found in multimedia or signal processing. The DOL framework implements the previously mentioned *Y-chart approach*, which is illustrated in figure 1.1.

The key idea of this design flow is to have a separate specification for the application (consisting of the application code and the application specification), which is provided by the application programmer, and the architecture (consisting of the architecture specification and a architecture dependant library) which is provided by the DOL programmer. The mapping combines the two specifications and determines "where" (binding the application to certain architecture elements) and "when" (scheduling of the application) the application is executed on the target architecture. Based on these inputs, the DOL generates code which can be compiled, resulting in one or multiple executables. These can be executed on the chosen architecture. The necessary inputs are described in more detail in the following sections.

The goal of this thesis was to extend the DOL to support code generation for the SCC architecture. The DOL framework also provides further functionality such as performance analysis and design space exploration (e.g. automatically finding an optimal mapping, given the application and archi-

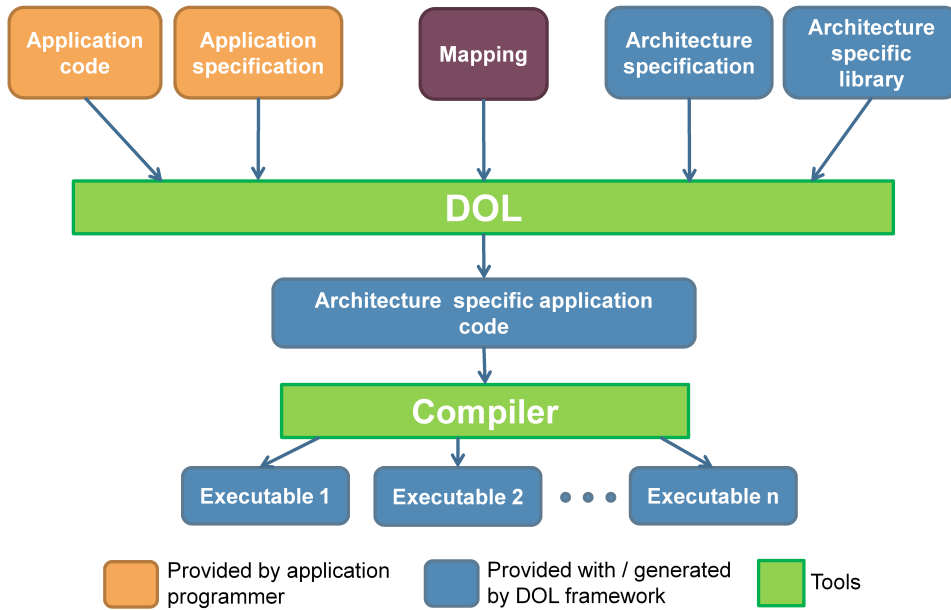


Figure 1.1: DOL design flow implementing the Y-chart approach.

ecture specification). These aspects were not part of this thesis (for more details see [12]).

1.2.1 Application

In order to map an application to a target architecture, using the DOL, the application must be written in a specific manner and implement/make use of the DOL API. DOL applications have to be implemented using *Kahn process networks* (KPN) as model of computation [17]. In such networks, a group of sequential processes communicate via unbounded FIFO channels. In general, processes may read data from several input channels, process the read data, and write the results to several output channels. There are also processes that act purely as data source or data sink. These have no input or output channels respectively. An example of such a process network is shown in figure 1.2. The application specification consists of two parts as shown in figure 1.1.

Application code

It is up to the programmer to implement her or his application in the form of a KPN. The structure of such a process is defined by the DOL API and

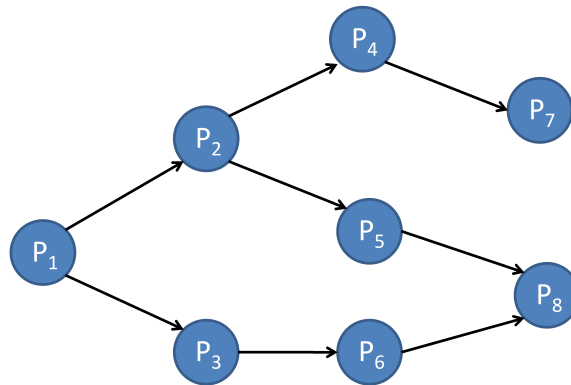


Figure 1.2: Example process network.

is shown below. Each process in the network must implement two DOL API functions which are explained in the following.

```

1 init () {
2     ...
3 }
4
5 fire () {
6     ...
7     DOL_read();
8     // do some computation
9     DOL_write();
10    ...
11    if (...) {
12        DOL_detach();
13    }
14 }

```

Listing 1.1: Typical structure of a DOL process.

- `DOL_init()`: This function initializes the process when the application is started. It is only called once per execution of the KPN.
- `DOL_fire()`: The fire function contains the actual functionality of the process. Typically, data is read from one or more input channels using the `DOL_read()` function. The read data items are then processed and the results are written to one or multiple output channels using the `DOL_write()` function. These functions do not have to be implemented by the application programmer. Their implementation depends on the target architecture (see section 1.2.2). The fire function is called repeatedly by the scheduler that controls the execution of the process network until the `DOL_detach()` function is called.

- `DOL_detach()`: The detach function tells the scheduler that the process calling the function no longer has to be executed. The process is responsible for detaching itself, i.e. this function is called from within `DOL_fire()` and the process has to know when to detach itself (e.g. after a certain number of data items have been processed).

Further details on how to program DOL applications using C/C++ as programming language and on the DOL API can be found in [18].

Application specification

The application specification describes the topology of the process network. It contains the information how the individual processes are connected with each other and therefore also which processes exchange data. Additionally, the size of the individual FIFO channels are defined here. The topology is described in XML according to DOL guidelines.

1.2.2 Architecture

The "architecture input" of the DOL design flow consists of two parts which are provided by the DOL programmer.

Application specific library

The application specific library implements the `DOL_read()` and `DOL_write()` functions used by the application programmer to write the code of the processes in the process network. These functions make use of the hard- and software resources offered by the target architecture. For example, for a multi-core system, a call to `DOL_write()` could be implemented as a write to a shared memory location and a call to `DOL_read()` as a read from a shared memory location.

Furthermore, the library is also responsible for the scheduling of shared resources. For example, if multiple processes are running on a single-core system and one process was assigned a higher priority by the application programmer, the architecture specific library is responsible for the higher priority process being always executed as soon as it is possible.

Architecture specification

Similar to the application specification, the architecture specification describes the hardware resources offered by the target platform. The archi-

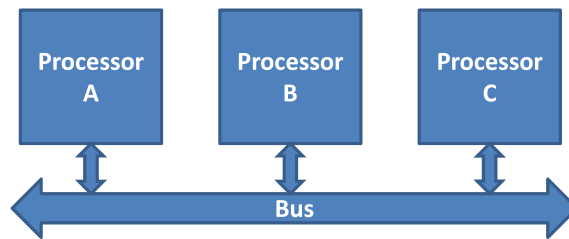


Figure 1.3: Example architecture.

architecture is described in terms of processors, memories, hardware (communication) channels and a few other elements. For example, consider the architecture shown in figure 1.3. The architecture specification could include different types of processors (e.g. processor A and B are x86 processors and processor C a DSP) and the bus is described as hardware channel. The architecture is described in XML according to DOL guidelines.

1.2.3 Mapping

The mapping combines the "application input" with the "architecture input" from figure 1.1. Elements of the application specification are mapped to elements of the architecture specification. DOL supports both manual and automatic mapping. In the case of a manual mapping, the mapping specification has to be provided by the application programmer. If automatic mapping is supported for the target architecture, it is also possible to let the DOL find an optimized mapping according to some design constraints such as achieving minimum power consumption or minimum execution time.

As an example, consider figure 1.4 which shows a possible mapping of the previously shown example process network to the example architecture. Processes are assigned to the processors on which they will be executed. The communication channels between processes running on distinct processors are mapped to the bus. Therefore, the data exchange between those processes will take place via the communication bus.

1.3 Contributions

The main contribution of this thesis was extending the DOL to support the execution of Kahn process networks on the SCC as well as on a networked Linux cluster (see section 2.3). This was done in two steps.

1. Implementation of the architecture specific library. The main goal was

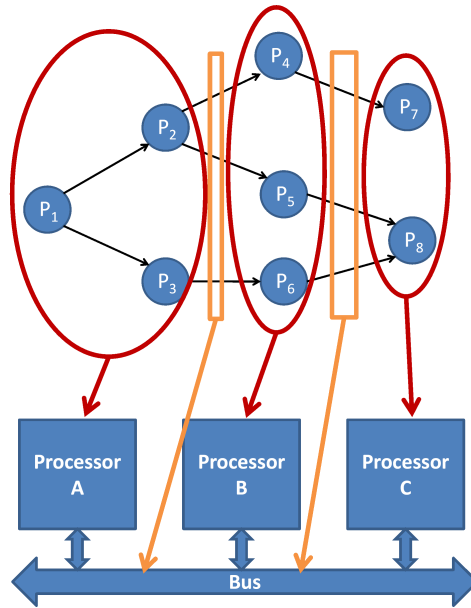


Figure 1.4: Possible mapping of the example PN on to the example architecture.

to have a scalable implementation with respect to the number of cores as the SCC offers many. It was shown that a flow control mechanism is necessary for executing distributed process networks without the possibility of deadlocks occurring (see section 3.3).

2. Extension of the DOL back-end by implementing a code generator for the two architectures. This allows the automatic code generation given the inputs described in section 1.2.

Finally, the implementation and the code generation was tested with a exemplary application.

1.4 Related Work

The main difficulty of this thesis was to implement the communication between processes running on different cores on the SCC. Communication between SCC cores is an ongoing research topic and there exist multiple approaches.

1. **RCCE**: RCCE is a small library provided by Intel. It allows communication via message passing on the SCC. It makes use of its specialized

message passing hardware. RCCE logically partitions the total message passing buffer (MPB) memory (see section 2.1) of $24 \times 16\text{KB}$ into 8KB buffers (one for each core). Two different interfaces are provided. The "basic" interface offers high level functions to send and receive messages between cores or to synchronize programs with barriers and fences. The "gory" interface offers lower level functions for better control over the MPB's such as directly reading and writing from or to a MPB. Additionally, RCCE also includes an API for various power management functions. However, RCCE only offers *synchronous* or *blocking* communication. This means that a process, that wants to send some data to another process, will have to wait for a matching receive call by the other process before it can return from its send call. The idle time, while a sending process is waiting for the receiving process, could possibly be used more efficiently. More information can be found in [20].

2. **iRCCE**: iRCCE is an extension of RCCE by a group at RWTH Aachen University. The main improvement is the implementation of *asynchronous* or *non-blocking* send and receive functions. Assume a core tries to send or receive a message, but the function cannot be completed at the instance of the call. Instead of waiting for the other core (as would be the case with RCCE), the function returns immediately, allowing the calling process to do some computation and trying to complete the communication at a later point in time. iRCCE implements a queueing mechanism to handle multiple outstanding communication requests to preserve the order of subsequent send and/or receive calls. Additionally, iRCCE also improves the performance of some other RCCE functions. More information can be found in [9].

3. **RCKMPI**: RCKMPI is an MPI implementation for the SCC by Intel. It makes use of the specialized message passing hardware resources offered by the SCC to provide low latency and high bandwidth communication. The advantage of RCKMPI over RCCE/iRCCE is that MPI applications are highly portable because MPI implementations exist for many platforms. Conversely, applications that were written using MPI can easily be executed on the SCC without any additional effort to port the application. More information can be found in [22].

Extending the DOL to support a certain architecture has been addressed in several previous theses. An example is the work by Lars Schor using the Cell Broadband Engine (used in the Playstation 3) as target architecture [19].

1.5 Outline

Chapter 2 provides an overview of the SCC architecture and how it can be programmed. Chapter 3 proposes a remote FIFO model, showing some theoretical aspects for the implementation of a remote FIFO channel. How these concepts were implemented is shown in chapter 4. Finally, an evaluation of the implementation and possible future work is given in chapter 5.

2

The SCC Architecture

The Intel *Single-chip Cloud Computer* (SCC) is a fully integrated many-core microprocessor. The SCC was developed as part of Intel's *Tera-scale Computing Research Program*. The program is concerned with increasing the performance and capabilities of current computers. The SCC should help to investigate many-core CPU's, their architectures and how to program them. Several research partners from academia and industry have been granted access to the SCC to do advanced software research. The Computer Engineering and Networks Laboratory of the Department of Electrical Engineering and Information Technology at ETH Zurich is one of those research partners.

This chapter gives an overview of the SCC architecture and what hardware resources it offers. Furthermore, the implication of the provided hardware on the programming of the SCC is discussed. An overview on the SCC architecture including how to program it using RCCE (see section 1.4) and an introduction to some SCC specific tools can be found in [15]. More detailed architectural information can be found in [14].

2.1 Architecture

The high-level block diagram of the SCC is shown in figure 2.1. The SCC die contains 24 tiles. The architecture of such a tile will be discussed shortly. The tiles are arranged in a 6×4 array and connected by a network-on-chip with mesh topology. The traffic flow between the tiles is handled by routers (one per tile). There are four on-chip memory controllers which

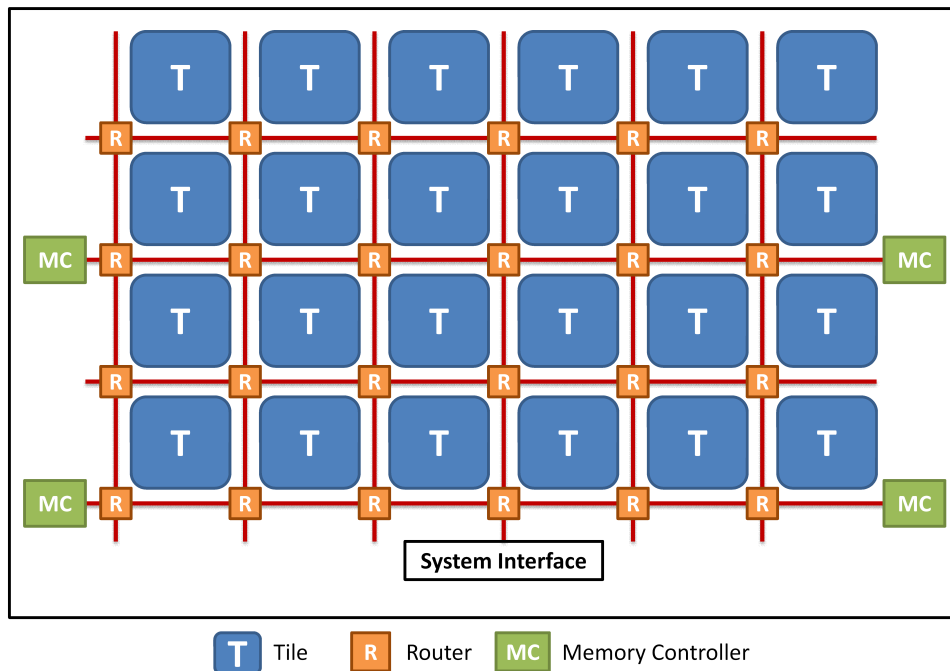


Figure 2.1: High-level SCC block diagram.

can access up to 64GB of external DDR3 memory. The external memory can be used as memory private to each core and as memory shared by all cores. The exact partitioning of the external memory can be configured. The entire microprocessor is controlled by a board management microcontroller (not shown in the figure) which initializes and shuts down critical system functions. It is normally connected to an external PC acting as management console (MCPC) via the system interface. Via the MCPC, programs can be loaded onto SCC cores and SCC configuration registers can be modified.

A simplified block diagram of one such tile is shown in figure 2.2. Its most important elements are described in the following.

- **P54C:** The P54C is a x86 instruction set core based on an older Pentium design. There are two such cores per tile. Each core has its own 16KB instruction and 16KB data L1 caches.
- **L2 cache:** Each core has its own 256KB L2 cache. A core's private off-chip DRAM is cached through the core's L2 and L1 caches according to the rules of the P54C processor. There is no cache coherence among the cores.
- **Mesh Interface Unit (MIU):** The cores on a tile access the network-

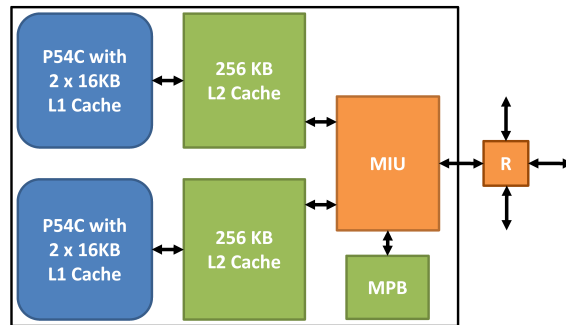


Figure 2.2: Simplified block diagram of a SCC tile.

on-chip via the MIU which is connected to the tile’s router. The MIU sends and receives data to and from the mesh. The MIU can be seen as an address translator. A core uses 32-bit addresses (*core address*), resulting in a 4GB memory space per core. However, the SCC platform can have up to 64GB of memory which it addresses with the *system address*. The MIU uses a lookup table (LUT) to translate the core address into a system address. There is one LUT per core. Each LUT entry can point to any memory location in the system (e.g. external memory or MPB). The LUT can be configured via the MCPC.

- **Message Passing Buffer (MPB):** In addition to the traditional cache structures, each tile has a 16KB MPB SRAM. Each core of the SCC can access the MPB of every tile. The idea of the MPB is to offer fast on-chip shared memory for efficient message passing between the cores.

2.2 Programming the SCC

2.2.1 Writing applications

The hardware offered by the SCC supports a message passing programming model. The total available MPB memory is typically divided into 8KB per core where each core can only write to its own 8KB but read from all other core’s memory segments. This allows efficient message passing between the cores via shared memory. It is possible for the programmer to explicitly access the MPB memory by configuring the memory map of a core via its LUT and configuration registers. To ease the programmer’s job, these low level details are abstracted into some message passing library such as RCCE (see section 1.4). The programmer can simply use the high level functions offered by such a library and be assured that the specialized hardware is

used internally.

2.2.2 Executing applications

Currently, the SCC supports two platforms.

1. Linux platform: Each core runs its own Linux operating system. Applications can be executed on core with operating system support. In this mode, the SCC behaves exactly the same as a networked cluster of Linux workstations.
2. Baremetal platform: Applications run directly on the cores without operating system support.

To execute applications on either of the two platforms, the SCC is accessed via the MCPC. From the MCPC, the SCC can be configured, applications can be compiled and loaded to one or several cores and finally be executed. The application's I/O is through the MCPC. It is also possible to use SSH to access each core individually from the MCPC and directly execute applications on a core.

An introduction to writing applications for the SCC and using the software offered by the MCPC can be found in [16].

2.3 Extending DOL to support the SCC

Due to the similarities between the SCC (Linux platform) and a networked Linux cluster, it was decided to extend the DOL to support both the SCC and a regular Linux cluster. This implied that the architecture specific library had to be written so it works for both architectures.

There were several reasons for this decision. Firstly, due to the similarities, supporting both platforms does not require substantially more effort. Secondly, because the SCC is a research microprocessor, it is not widely available and therefore not always accessible by the programmer. The Linux cluster on the other hand is one of the most popular multiprocessor architectures in the world. Before testing an application on the SCC, the programmer could test his or her application on a Linux cluster. If it runs on this platform, it should also work on the SCC. Furthermore, there are other MPSoC architectures that make use of Linux. The Linux cluster implementation could also be useful to prototype applications for these platforms. Finally, being able to easily program applications to run on a networked Linux cluster could be of interest itself.

The major part of extending the DOL for the two platforms, is the implementation of the remote inter-process communication. This is discussed in-depth in the following chapter.

3

Remote FIFO Model

This chapter describes some of the theoretical considerations that are important for the implementation of remote FIFO channels. Section 3.1 describes how a remote FIFO channel is modelled and section 3.2 shows how multiple remote FIFO channels can be managed. Finally, section 3.3 shows that some form of flow control is necessary to avoid deadlocks and a solution is presented. Note that when mentioning a "(SCC) core" in the following, it could also say "Linux workstation".

3.1 Endpoint concept

Consider the situation in figure 3.1. Two processes P_1 and P_2 are mapped to the same SCC core. P_1 sends data to P_2 . By definition of a KPN, processes exchange data over FIFO channels. As the processes run on the same core, the FIFO channel can easily be implemented with globally allocated memory because the two processes share the same address space when they are executed in separate threads.

Now consider the situation in figure 3.2. The processes P_1 and P_2 are mapped to two different SCC cores. P_1 still sends data to P_2 . The FIFO channel can no longer be implemented as described above. Instead of having one single FIFO channel which is accessed by both processes, there is a "sending endpoint" on the writing process's core and a "receiving endpoint" on the reading process's core. The writing/reading process writes/reads to the sending/receiving endpoint as it would to a local FIFO channel. Functionality

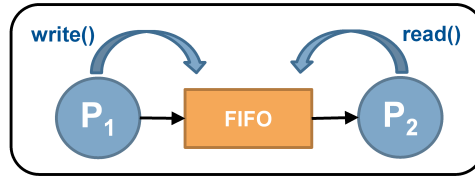


Figure 3.1: FIFO channel when two processes are mapped to the same core.

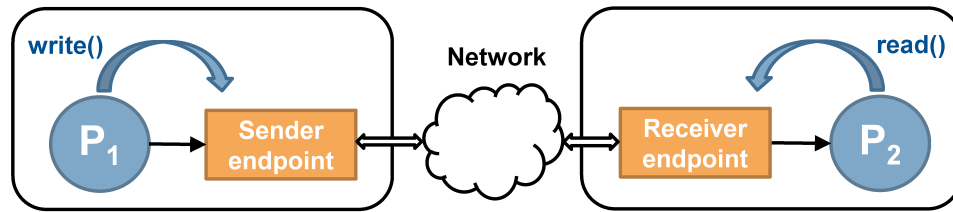


Figure 3.2: FIFO channel when two processes are mapped to different cores.

is required to transfer the data from the sending endpoints of one core to the corresponding receiving endpoints on other cores. This is discussed in section 3.2.

3.2 Channel management

To transfer data from one core to another, it must be sent over the network. For sending and receiving data, the *sockets* API is used. An introduction can be found in [11]. The sockets API documentation can be found in the UNIX man pages. As discussed in section 2.2, the SCC (when used as Linux platform) is equivalent to a networked Linux cluster. Therefore, every core on the SCC has its own IP address. The sockets API was chosen because it is widespread, and compatible with both architectures targeted in this thesis, i.e. the SCC and a cluster of Linux workstations.

A connection can be established between a socket pair (one each on the writing/reading process's core). Stream sockets are used, which results in TCP/IP connections. A TCP/IP connection guarantees that the data is not lost in the network and that it is received in the same order as it was sent. Sections 3.2.1 and 3.2.2 discuss how such connections should be established between cores.

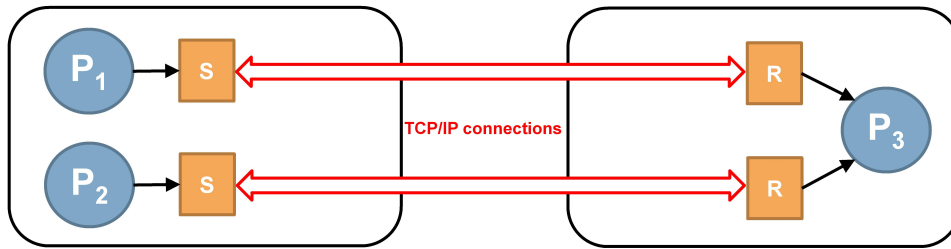


Figure 3.3: TCP/IP channels when establishing a connection per edge in the KPN.

3.2.1 Connection per KPN channel

One possible approach is shown in figure 3.3. Assume two processes P_1 and P_2 , mapped to one core, send data to a process P_3 , mapped to a different core. This corresponds to two edges in the KPN. For each pair of communicating processes that are mapped to different cores, a TCP/IP connection is established. One socket pair is required per connection.

Although the data of both connections goes through the same hardware channel (from one core to the other over the network), two "software channels" are used. This results in unnecessary overhead compared to using only one TCP/IP channel (one socket pair). Firstly, more system calls are necessary because e.g. the core running the writing processes has to call `send()` on two sockets instead of one. Secondly, before the data is actually injected into the network, it is packetized by adding header information (such as TCP and IP headers). If two separate TCP/IP channels are used, this overhead has to be sent twice when P_1 and P_2 send data to P_3 . If the data were to be sent over one TCP/IP connection, the header information only had to be sent once if the data from both processes is sent at the same time.

3.2.2 Connection per physical channel

By using only one TCP/IP channel per pair of cores that run processes that communicate with each other (i.e. per physical channel), the communication overhead (less calls to socket functions) and the network traffic can be reduced in certain situations.

This approach is shown in figure 3.4. The core running the reading process (P_3) receives data from the two processes P_1 and P_2 over one TCP/IP connection. The receiving side must be able to tell to which receiving endpoint of P_3 the received data has to be written because the data could be from P_1 or P_2 . This is achieved by multiplexing the data on the sending

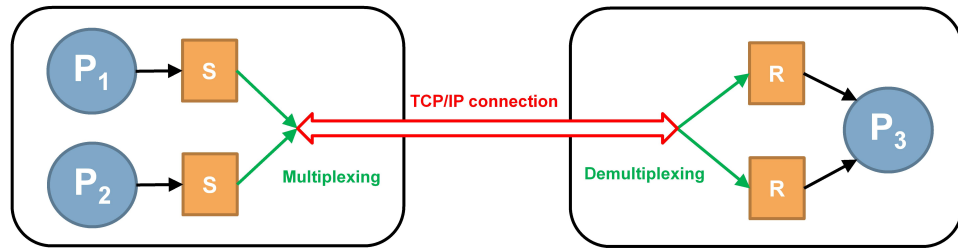


Figure 3.4: TCP/IP channel when establishing a connection per physical channel.

side. Multiplexing is done by adding some header information to the data from each sending endpoint. The header must have the following content.

1. Information from which sending endpoint the data originates or to which receiving endpoint is should be written. Every edge in the KPN of the application is assigned a unique number. This number can be used to represent this information because only one sending and receiving endpoint belong to one channel (KPN edge).
2. Information on the length of the data belonging to this header. This is necessary so that the receiver can tell where the payload ends and the next header starts if multiple of these packets were received.

The receiving side can then demultiplex the received data by reading the header information and writing the payload data to the proper receiving endpoints.

3.3 Deadlock-free implementation

In this section, it is shown that some form of flow control is necessary to guarantee a deadlock-free execution when multiple logical channels share the same physical channel. Section 3.3.1 shows how such a deadlock can occur. Section 3.3.2 shows the proposed solution.

3.3.1 Deadlock situation

Assume data could be transferred from sending endpoints as soon as a process has written to one. If this was the case, a deadlock can occur as shown in figure 3.5.

Processes P_1 and P_2 are mapped to core A and process P_3 to core B. Both processes on core A send data to the process on core B. In order that P_3 can process data, it requires both a data item from P_1 and P_2 . To send data items from both sending endpoints on A to B, it must first be transferred from the sending endpoints to an output buffer where also the header information is added. From the output buffer, it is sent over the TCP/IP connection to core B, where it is received to an input buffer. The header information is decoded and the data is distributed to the corresponding receiving endpoints. With the previous assumptions, the situation described in the following leads to a deadlock.

1. P_2 generates a data item (red) and writes it to its sending endpoint.
2. The data item is multiplexed to the output buffer. By reading it from the sending endpoint, the data is consumed from there.
3. The data is sent over the network from the output buffer of A to the input buffer of B.
4. By demultiplexing, the data item is transferred from the input buffer to one of the receiving endpoints of P_3 . P_3 can not consume the data because its other receiving endpoint is empty.
5. Assume P_2 generates another data item (pink). This could happen in a general KPC application, because e.g. P_2 produces data much quicker than P_1 .
6. Steps 2 and 3 are repeated with the second data item. However, the receiving endpoint to which the data should be written is already occupied. To avoid losing the data, it must be kept in the input buffer. The input buffer is now full. This means that no data can be transferred from core A to core B. Data can only be transferred if P_3 consumes that data in the occupied sending endpoint. This will only occur if data from P_1 is transferred from A to B. Hence, we have a deadlock.

Changing the sizes of the sending/receiving endpoints and/or the input/output buffers can not solve this problem. Bear in mind that this problem is not specific to the SCC platform or DOL applications. It can occur everywhere where two (or more) senders send data to one receiver over the same physical channel and where the receiver requires data from both senders in order to process data.

3.3.2 Flow control to avoid deadlocks

The solution to the problem described in section 3.3.1, is to control the data flow between a sending endpoint and its receiving endpoint. In general, this

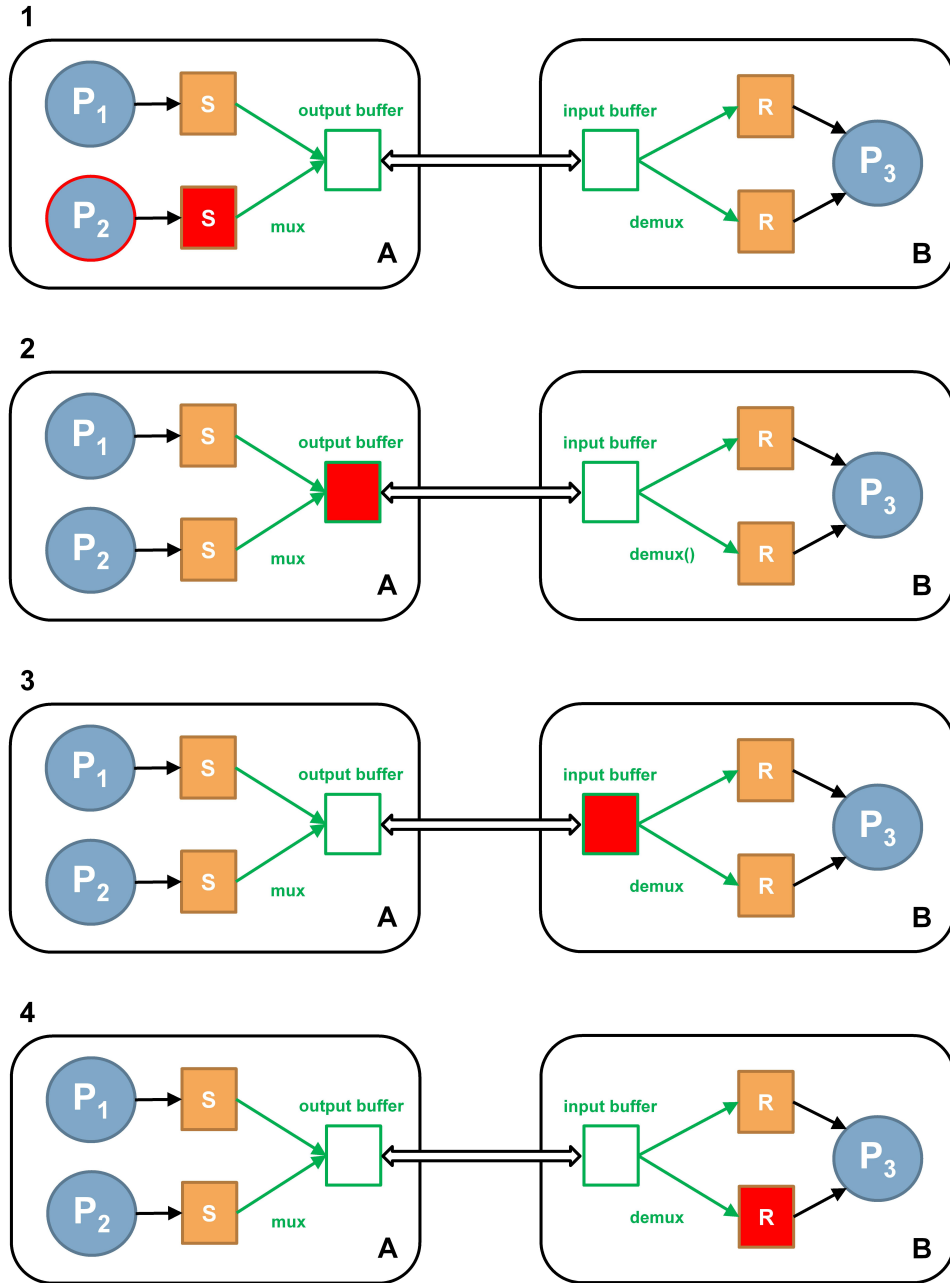


Figure 3.5: Implementation leading to a deadlock in certain situations.

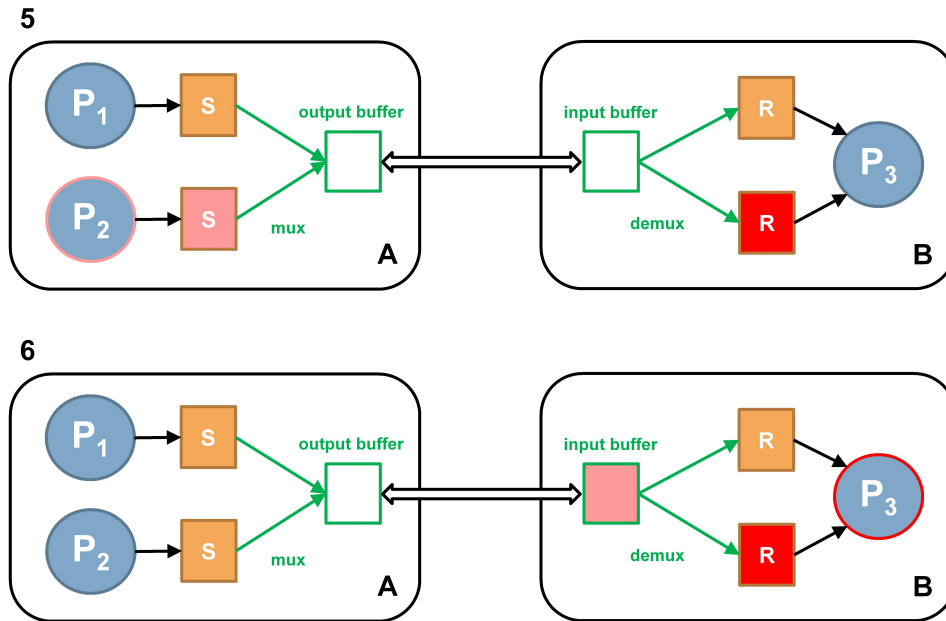


Figure 3.5: Implementation leading to a deadlock in certain situations.

means that a sender should only send as much data as it knows the receiver can handle. As a consequence, the sender must in some way be informed by the receiver on how much more data it can accommodate.

Assume that data is not consumed from the sending endpoint when writing it to the output buffer. Part 4 of figure 3.5 would then look like shown in figure 3.6. It shows that if the receiving endpoint is full, its corresponding sending endpoint will also be full. In figure 3.6 this means that P_2 can not generate data because it can not write to its sending endpoint. Hence, the input buffer on core B can never be full and it can always receive data from P_1 independently of how long it takes P_1 to generate data.

This approach requires a mechanism to free the space in sending endpoints when appropriate. It is safe for a process to write to a sending endpoint when there is free space in its corresponding receiving endpoint. This avoids having a full input buffer. As long as there is data in a sending endpoint, it looks for the writing process as if its reading process had not consumed the data yet. So the sending endpoint needs to be updated to reflect the state of its receiving endpoint when the process reading from the receiving endpoint has consumed data from it. This synchronization between sending and receiving endpoints is shown in figure 3.7.

1. Both processes P_1 and P_2 have produced a data item. Because P_3 has

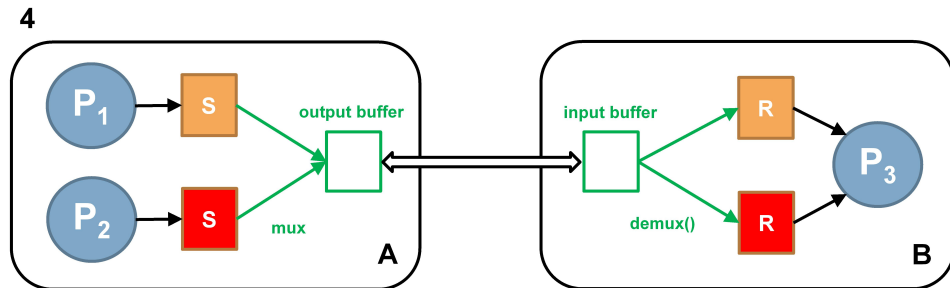


Figure 3.6: Part 4 of figure 3.5 without data being consumed from the sending endpoint.

not consumed this data from its receiving endpoints yet, it also remains in the sending endpoints.

2. P_3 reads the data from its receiving endpoints thereby consuming the data.
3. Core B notifies core A how much data was consumed from its receiving endpoints. Core A clears the same amount of data from its corresponding sending endpoints. P_1 and P_2 can now write data to their sending endpoints again.

Again, this solution works for the more general case described at the end of section 3.3. It ensures that each sender always knows the minimum amount of data it can safely send to the receiver without "blocking" the physical channel for the other sender.

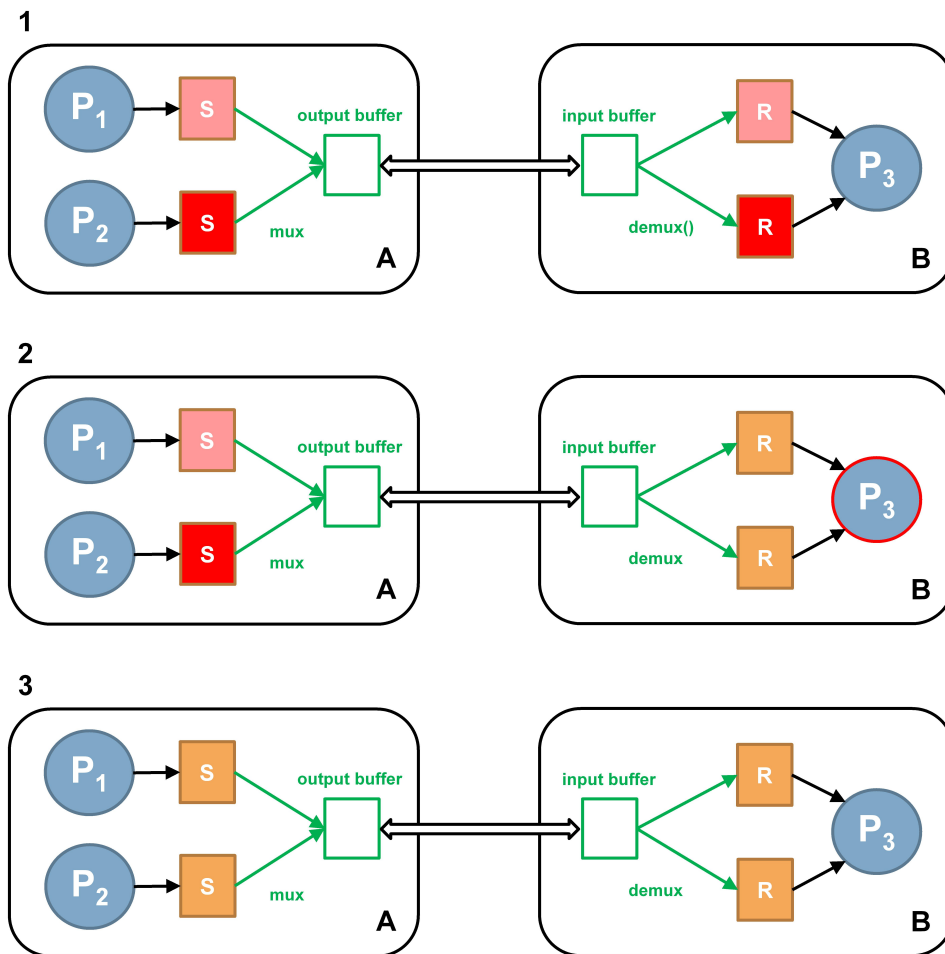


Figure 3.7: Deadlock-free implementation.

4

Implementation

This chapter describes the implementation of the architecture specific library (see figure 1.1). This includes the implementation of the concepts described in chapter 3. Furthermore, it is shown how the DOL was extended to support the new architectures.

4.1 Local and remote FIFO channels

This section describes the implementation of local (used if two processes are mapped to the same core) and remote (used if two processes are mapped to different cores) FIFO channels.

4.1.1 Local FIFO implementation

Before this thesis, the DOL already supported the execution of process networks on a single core of a workstation running Linux (or a different POSIX compliant operating system). This DOL extension (also termed *visitor*) is called *PipeAndFilter* (PaF).

In the PaF implementation, each process of the process network is executed in its own thread. For handling threads, the *POSIX threads* (or *pthread*) API is used. It allows the manipulation and synchronization of and between threads. An introduction can be found at [8]. Because the PaF visitor allows the execution of process networks on a Linux workstation, it is also possible

to use it to generate applications to be run on a single SCC core (as discussed in section 2.2).

Figure 3.1 illustrates the functionality of the PaF visitor. As stated above, each process of the process network (in this example the PN consists of two processes P_1 and P_2 connected by a channel from P_1 to P_2) runs as an own thread on the same core. Because all processes share the same address space, data can be exchanged via globally allocated memory. The FIFO channels of the PN are therefore implemented as shared memory FIFO buffers. The shared memory is protected with *pthread*s *mutex* objects to avoid that two processes access the shared memory at the same time to avoid data corruption.

This shared memory FIFO buffer is implemented as circular FIFO buffer in a class called *Fifo*. It offers a simple read/write interface as shown in listing 4.1.1. It is similar to the interface specified by the DOL API (see [18]). Furthermore, there are several auxiliary functions, e.g. to check how many bytes are free in the buffer. For each channel in the PN, one *Fifo* object is instantiated. Two processes that are connected by an edge in the PN both have a pointer to the same *Fifo* object. The writing process uses the *Fifo*'s write interface and the reading process uses its read interface.

```

1 virtual unsigned read(void* destination , unsigned len);
2 virtual unsigned write(const void* source , unsigned len);

```

Listing 4.1: Fifo read/write interface.

As described in section 1.2.1, the application programmer uses the DOL API to read and write from channels in the PN. What actually happens when these functions are called, depends on the architecture specific library of the DOL visitor in question. For the PaF visitor, a call to *DOL_write()* results in the corresponding process calling the *write()* function of the *Fifo* object shared with an other process, thereby writing to a shared memory location. Conversely, calling *DOL_read()* leads to the corresponding process calling the *read()* function of the *Fifo* object shared with an other process, thereby reading from a shared memory location. Reading from a *Fifo* object consumes the read data from the buffer.

The PaF *Fifo* is reused in the SCC visitor for channels between processes that are mapped to the same core.

4.1.2 Remote FIFO implementation

As discussed in section 3.1, a "sending endpoint" is generated on the writing process's core and a "receiving endpoint" is generated on the reading process's core if the two processes are mapped to different cores.

These endpoints were implemented in the classes *RemoteSendingFifo* (RSF) and *RemoteReceivingFifo* (RRF). Both the RSF and RRF classes are derived from the *Fifo* class. This ensures that a process can use the same interface when reading/writing from/to a local or a remote channel. A part of their interfaces is shown in listings 4.2 and 4.3.

```
1 unsigned read(void* destination , unsigned len);
2 void clear(unsigned len);
```

Listing 4.2: Part of the RSF interface.

```
1 unsigned read(void* destination , unsigned len);
2 unsigned sync();
```

Listing 4.3: Part of the RRF interface.

The RSF *read()* function has the same semantics as the *Fifo*'s but is modified internally. It implements a read from the RSF's buffer without consuming the read data. As shown in section 3.3.2 this is necessary to avoid deadlocks. Additionally, there is the *clear()* function which frees the specified number of bytes in the RSF's buffer. It is used after data was consumed from a RRF belonging to a RSF.

The RRF has a *sync()* function which returns the number of bytes that were consumed from the RRF since the last call to *sync()*. It is required to tell a RSF belonging to a RRF that data was consumed. The RRF also has a *read()* function with the same form as the *Fifo*'s. Internally, it is modified to count the number of consumed bytes since the last call to *sync()*. This counter is reset and returned when calling *sync()*.

4.2 Remote Channel Management

This section describes the implementation of the channel management. The concepts from sections 3.2 and 3.3 are implemented. As mentioned there, internet sockets using TCP/IP channels are used to establish remote connections.

4.2.1 The Remote Channel Connector

To establish a TCP/IP connection between two cores, a socket needs to be set up on each of the two cores. A socket is managed by the *RemoteChannelConnector* (RCC) class. For each connection a core establishes with other cores, one socket and therefore one RCC object is instantiated on each of the two cores. The important interface functions and member objects/variables of the RCC are shown in listing 4.4.

To establish a TCP/IP connection between two hosts, one of them has to wait for the incoming connection request sent by the other one. After the connection was established, it can be used bidirectionally. So for every pair of RCC's that want to establish a connection between each other, one waits for the other to actively connect to it.

The RCC connecting to the other one uses the function *setupConnectingSocket()* followed by *_connect()*. The former simply creates a socket and specifies to which IP/port (specified by the private members *_ip_address* and *_port* which are set according to the architecture specification (see section 4.3) it will have to connect. *_connect()* then establishes the connection.

```

1 public :
2     int  setupConnectingSocket () ;
3     int  _connect () ;
4     int  _send () ;
5     int  _receive () ;
6     ...
7
8 private :
9     RemoteChannelMultiplexer* _multiplexer ;
10    RemoteChannelDemultiplexer* _demultiplexer ;
11    int  _socket ;
12    char* _ip_address ;
13    unsigned short int _port ;
14    ...

```

Listing 4.4: Part of the RCC interface.

The RCC functionality is illustrated by an example in figure 4.1. Processes $P_1 - P_4$ are running on core A. Each of them is synthesized in to a *pthread*. P_1 and P_2 are data sources and are connected with processes on core B (not shown). Processes P_3 and P_4 are data sinks and receive data from core B. The four edges in the corresponding KPN are synthesized in to a single socket pair as they all share the same physical channel. The processes write/read to/from RSF's/RRF's. Because there are processes on core A that communicate with processes on core B there is one RCC on each of the two cores. The RCC holds the socket (S) associated with the connection. The RCC also holds one object each of the types RemoteChannelMultiplexer (RCMU) and RemoteChannelDemultiplexer (RCDE). With this example, the RCC's *_send()* and *_receive()* functions are explained.

- ***_send()***: Internally, this calls functions of the RCMU. In a first step, the RCMU checks all RSF's associated with its RCC for data that was not sent to the other core yet. Unsent data is copied to the output buffer (held by the RCMU) and header information is added (see section 3.2.2). The RCMU then checks all RRF's associated with its RCC for consumed data using the RRF's *sync()* function. If data was consumed

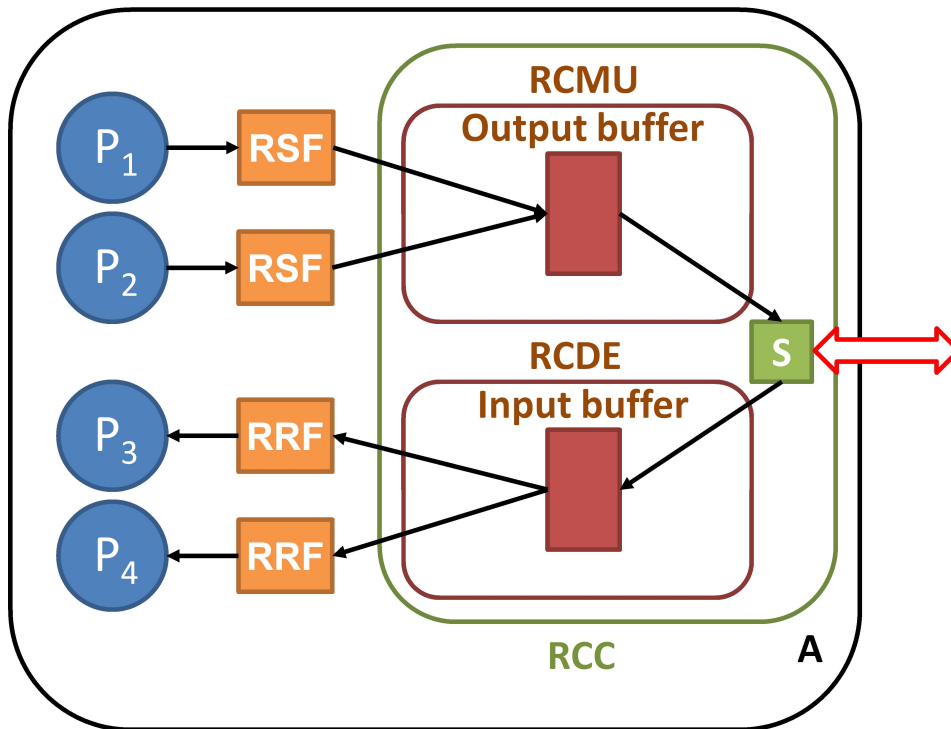


Figure 4.1: Illustration of RCC functionality.

since the last call to `sync()`, a synchronization message is generated in the output buffer. It contains the channel number and the number of consumed bytes. The second step is for the RCMU to send all the data accumulated in the output buffer via the socket over the TCP/IP connection.

- `_receive()`: Internally, this calls functions of the RCDE. In a first step, the RCDE receives data via the socket that was sent by the other core. Received data is stored in the input buffer. The second step is for the RCDE to demultiplex the data in the input buffer. Data is written to the corresponding RRF's. If synchronization messages were received, the RCDE calls the `clear()` function of the appropriate RSF's.

4.2.2 The Remote Channel Manager

The *Remote Channel Manager* (RCM) class manages all RCC's on a core. There is one instance of the RCM per core. There is a separate thread calling the RCM's functions. This is illustrated in figure 4.2. Assume a process P_1 sends data to two processes P_2 and P_3 . P_1 , P_2 and P_3 are mapped to cores

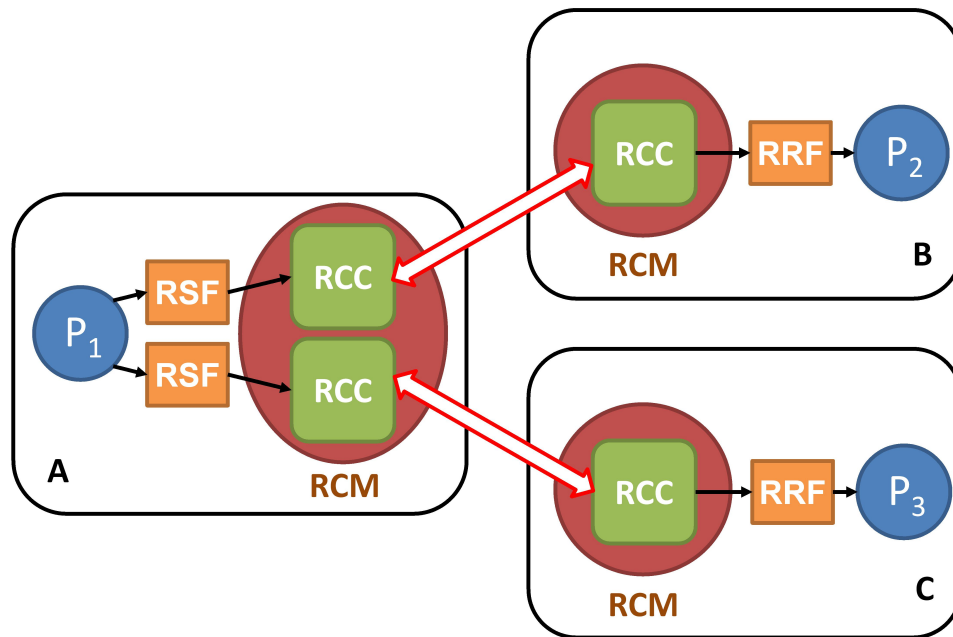


Figure 4.2: Illustration of RCM functionality.

A, B and C respectively. Core A has two RCC's corresponding to the two physical channels: from core A to B and from A to C. Core B and C have one RCC each. In the example, each core runs two threads: one executing a process and one for handling the remote communication. The RCM calls its RCC functions as described in the following.

The important RCM member functions and variables are shown in listing 4.5. The RCM holds two lists containing all RCC's on its core. There are two different lists so the RCM can tell if a RCC will initiate or wait for a TCP/IP connection (see section 4.2.1).

- **init()**: This function is called once before the application is executed. It establishes all TCP/IP connections of the RCC's held by the RCM. Initialization can only complete if the *init()* function is also called on the cores with the RCC's that a connection should be established to.
- **run()**: Calls the *select()* function (see [5]) to check if data can be sent and/or received from any sockets held by the RCM's RCC's without the socket function call blocking. This is necessary because non-blocking sockets are used to avoid deadlocks. For each socket where data can be sent and/or received the RCM calls the corresponding RCC's *_send()* and or *_receive()* function (see section 4.2.1). *run()* is called repeatedly by the thread handling the remote communication. Each time it is

called, as much data as possible, that is waiting to be sent or received, it sent or received.

```

1 public :
2     bool init () ;
3     bool run () ;
4     ...
5
6 private :
7     list <RemoteChannelConnector*>* _connectingConnectorList ;
8     list <RemoteChannelConnector*>* _acceptingConnectorList ;
9     ...

```

Listing 4.5: Important RCM member functions and variables.

This approach is scalable with respect to the number of processes running on one core because there is only one thread managing all remote communication.

4.3 Integration into the DOL

The DOL was extended so that it can automatically generate code, given the application code, the KPN, the mapping and the architecture specifications (see figure 1.1). The code generation procedure is explained in section 4.3.1. The required XML inputs are discussed with the help of an example application in section 4.3.2.

4.3.1 Code generation procedure

The DOL basically parses the XML inputs mentioned above and stores the information in corresponding classes. For example, there is a *Mapping* class which among other things provides a list of all processes used in the mapping. Information about a process is again managed by a class called *Process*. The DOL is written in Java. The generated code is C++.

Code generation consists of three parts which were each implemented in a separate class. A fourth class (called *SCCVisitor.java* for the SCC visitor), is used to call the functions of the other three classes. The three parts are discussed in the following.

- Create a Makefile (*SCCMakefileVisitor.java*): The Makefile can be used to easily compile the generated code. After compilation, there will be one executable per processor that has processes mapped to it in the mapping specification. This number can be retrieved from the

DOL *Mapping* class and is used to generate the Makefile accordingly. The Makefile is the only difference between the generated code for the SCC and a networked Linux cluster as target architecture. The SCC requires compilation with the Intel C/C++ compiler called *icpc*. The *g++* compiler can be used when compiling the application for a Linux cluster.

- Create process wrapper classes (*SCCProcessVisitor.java*): A process wrapper is a class that basically holds function pointers to the *init()* and *fire()* functions which were specified in the application code. Additionally it contains *Fifo* pointers which specify from/to what *Fifo* object the process will read/write. These will be set in the third part. A process wrapper class is generated for each process type in the application. The generated classes are derived from the *ProcessWrapper* base class.
- Generate mapping dependent code (*SCCModuleVisitor.java*): This is the main part of the code generation. Three separate files are generated for each processor in the mapping specification. Two of them are *processnetwork_partXXX.cpp* and *processnetwork_partXXX.h*. XXX is a number starting from 001, where the lowest number corresponds to the first processor in the architecture specification. These two files basically contain the instantiations of the necessary *Fifo*, RCC, RRF and RSF objects for the corresponding processor. The third file is *sc_application_XXX.cpp*. It contains the main routine for the corresponding processor. A more detailed explanation on how the process-network files are generated is given in the following.
 1. Each channel of the process network specification is assigned a unique ID (see section 3.2.2).
 2. Choose a processor from the processors that are used in the mapping.
 3. Get the processes that are executed on the chosen processor according to the mapping.
 4. For each of these processes, check if it communicates with processes on the same processor or with processes on different processors. If a process sends/receives data locally, a *Fifo* object is instantiated (see section 4.1.1). If a process sends/receives data remotely, a RSF/RRF is instantiated (see section 4.1.2).
 5. For each processor with which the chosen processor communicates, a RCC is instantiated. Each RCC requires its RSF's/RRF's with their corresponding channel ID. Furthermore the RCC needs the IP/port to which it connects to or from where the connection will be instantiated. The IP/port information can be extracted

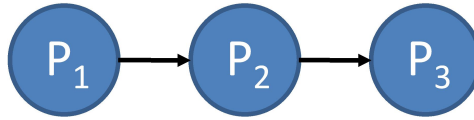


Figure 4.3: Simple example PN.

from a processor's name in the architecture/mapping specification (see section 4.3.2). As mentioned in section 4.2.1, for two RCC's on different processors to establish a connection, one must initiate the connection while the other one accepts it. For each instantiated RCC, it is arbitrarily chosen if it should establish or wait for the connection. When the corresponding RCC for the other processor is instantiated, it is set to wait for or establish the connection depending on what was chosen for the other RCC.

6. Instantiate a process wrapper object for each of the processes found in 3).
7. Assign the instantiated *Fifo*'s, RSF's and RRF's to the corresponding process wrappers to "connect" the processes.
8. Repeat steps 2)-7) for each processor in the mapping to obtain the processnetwork files for each processor.

4.3.2 Code generation for an example application

Consider the KPN shown in figure 4.3. P_1 sends data to P_2 which processes it and sends the results to P_3 . The source code of the processes is given in the files *process1.c*, *process2.c* and *process3.c*. The KPN specification (written in XML) is shown in listing 4.6. It contains an entry for each process, channel and connection between the two. The FIFO channel sizes are specified here.

```

1 <!-- processes -->
2 <process name="process1">
3   <port type="output" name="1"/>
4   <source type="c" location="process1.c"/>
5 </process>
6
7 <process name="process2">
8   <port type="input" name="1"/>
9   <port type="output" name="2"/>
10  <source type="c" location="process2.c"/>
11 </process>
12
13 <process name="process3">
14   <port type="input" name="1"/>
15   <source type="c" location="process3.c"/>
16 </process>
  
```

```

17
18 <!-- FIFO channels -->
19 <sw_channel type="fifo" size="10" name="C1">
20   <port type="input" name="0"/>
21   <port type="output" name="1"/>
22 </sw_channel>
23
24 <sw_channel type="fifo" size="10" name="C2">
25   <port type="input" name="0"/>
26   <port type="output" name="1"/>
27 </sw_channel>
28
29 <!-- connections -->
30 <connection name="p1-c1">
31   <origin name="process1">
32     <port name="1"/>
33   </origin>
34   <target name="C1">
35     <port name="0"/>
36   </target>
37 </connection>
38
39 <connection name="c1-p2">
40   <origin name="C1">
41     <port name="1"/>
42   </origin>
43   <target name="process2">
44     <port name="1"/>
45   </target>
46 </connection>
47
48 <connection name="p2-c2">
49   <origin name="process2">
50     <port name="2"/>
51   </origin>
52   <target name="C2">
53     <port name="0"/>
54   </target>
55 </connection>
56
57 <connection name="c2-p3">
58   <origin name="C2">
59     <port name="1"/>
60   </origin>
61   <target name="process3">
62     <port name="1"/>
63   </target>
64 </connection>

```

Listing 4.6: KPN specification of the PN in figure 4.3

Assume the processes will be mapped to two SCC cores. A possible architecture specification is shown in listing 4.7. As discussed in section 4.2 the IP

address and a port is required for each core to establish the TCP/IP connections. This information is not hardcoded, but can be specified by the user by choosing an appropriate processor name in the architecture specification. For example, if the second processor wanted to connect to the first processor, it would establish a connection to IP address 192.168.0.1 on the port 1337. One entry is required per processor used to run the application. If all SCC cores were to be used, the architecture specification contained 48 entries.

```

1 <processor name="192.168.0.1:1337" type="RISC">
2 </processor>
3
4 <processor name="192.168.0.2:1337" type="RISC">
5 </processor>

```

Listing 4.7: Partial SCC architecture specification.

Say P_1 is mapped to one core and P_2 and P_3 to the other core. The corresponding mapping specification is given in listing 4.8. One *binding* entry is required per *process* entry in the KPN specification. Each entry maps a process (defined in the KPN specification) to a processor (defined in the architecture).

```

1 <!-- processes bound to the first core -->
2 <binding name="process1" xsi:type="computation">
3   <process name="process1"/>
4   <processor name="192.168.0.1:1337"/>
5 </binding>
6
7 <!-- processes bound to the second core -->
8 <binding name="process2" xsi:type="computation">
9   <process name="process2"/>
10  <processor name="192.168.0.2:1337"/>
11 </binding>
12
13 <binding name="process3" xsi:type="computation">
14   <process name="process3"/>
15   <processor name="192.168.0.3:1337"/>
16 </binding>

```

Listing 4.8: Mapping the PN from figure 4.3 to two SCC cores.

After code generation, the code can be compiled using the Makefile. Then, the executables have to be distributed to their corresponding core (SCC) or workstation (Linux cluster). When all executables are started, TCP/IP connections are established and the KPN is executed.

5

Evaluation

This chapter shows the performance evaluation of the implementation. An example application is presented. It is executed on the SCC using a different number of cores. The results are compared to the *PaF* implementation (see section 4.1.1) which runs on a single core. The results are discussed and future improvements/extensions are suggested.

5.1 Test setup

For the evaluation, the application with the KPN shown in figure 5.1 was used. It consists of three different types of processes. The generator simply writes arbitrary numbers to its output channel. The computation process (*comp*) reads the numbers produced by the generator. It then performs some arbitrary computation on each number. The computation is in a loop with a configurable number of iterations. The number of iterations can be changed to change the communication vs computation ratio. The final result is written to the process's output channel. Finally the consumer reads the results produced by the *comp* process and prints them.

The number of *comp* processes is parametrized. This allows to easily run the same application on a different number of cores. While the number of processes in the KPN changes, if it has to be executed on different numbers of cores, the total amount of data that has to be processed remains constant. For example, assume that the generator is configured to produce 1000 numbers. If the KPN with two *comp* processes is chosen, each of the two will

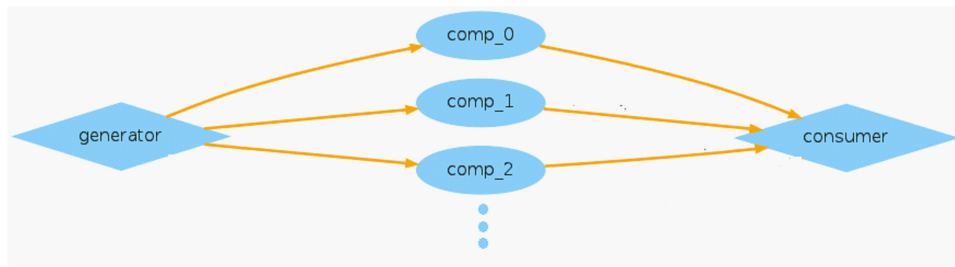


Figure 5.1: KPN of the application used for the evaluation.

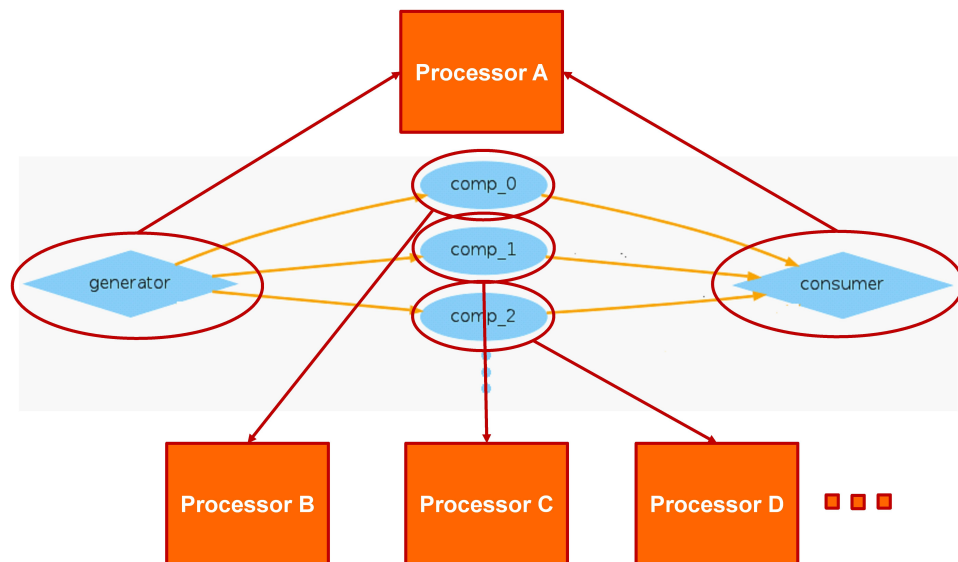


Figure 5.2: Mapping the application to the SCC.

have to perform computation on 500 numbers. If four *comp* processes are chosen, each of them processes 250 numbers.

The reference implementation is the *PaF* visitor. It allows the execution of a KPN on a single SCC core (see section 4.1.1). Only one *comp* process is used. How the application is mapped to the SCC is shown in figure 5.2. The generator and consumer are mapped to their own core. For each core that is used for the application, an additional *comp* process is generated and mapped to that core. For the evaluation, the application was executed on a range from 2 to 16 SCC cores.

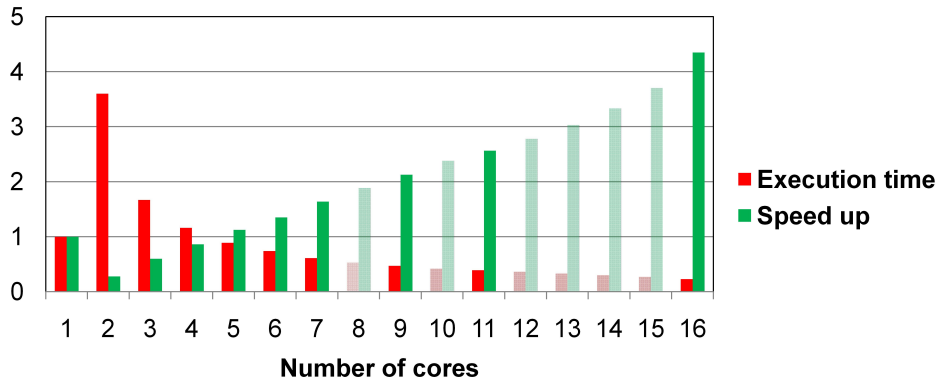


Figure 5.3: Evaluation results.

5.2 Results and discussion

The results are shown in figure 5.3. For the green bars, the Y-axis represents the speed up compared to the application running on a single core using the PaF visitor. For the red bars, it represents the execution time normalized by the execution time of the reference application (which is the inverse of the speed up). The bars at "number of cores = 1" correspond to the reference implementation.

The following observations were made.

- Ideally, one would expect a speed up of n when using n cores. This is however not the case (e.g. 16 cores only result in a speed up of around 4.3). The reason for this lies in the inefficiency of the thread managing the remote communication. It polls the RSF's for data to be sent by repeatedly calling the `run()` function of the RCM (see section 4.2.2). This results in wasted CPU time that could be used by the thread running the `comp` process. This could be verified by adding a `sleep()` command to the remote communication thread when no data could be sent. Using the `sleep()` command, almost linear speed up could be achieved.
- The implementation scales well with the number of cores for the example application. The transparent bars in figure 5.3 are not actual measurement but are extrapolated values using the formula

$$\text{Execution time}(n) = \frac{n-1}{n} \cdot \text{Execution time}(n-1)$$

which describes the execution time depending on the number of cores

when ideal scaling is assumed. As can be seen in figure 5.3, the extrapolated values fit well into the actual measurements which indicates good scaling.

5.3 Future work

5.3.1 Improving performance

For better performance, the remote communication thread must be implemented more efficiently. Basically, the thread should be suspended after no data could be sent or received. This avoids repeatedly trying to send data although none is available. There are two ways in which the suspended thread can be woken up again.

First, if a process writes to a RSF, the thread running the process could notify the suspended remote communication thread because data can be sent. However, there could be a situation where all processes on a core are waiting for incoming data. If the remote communication thread were suspended and waiting for a process to write to its RSF, a deadlock would occur.

In this situation, the remote communication thread should wait for incoming data by calling *select()* in the RCM's *run()* function. *select()* is used to check if data can be sent or received from a set of sockets. It returns instantly if either of the two are possible. Assume a set of sockets is only checked for data to be received. In this case, *select()* will only return when a other processor sends data to this processor. The thread is suspended while it is waiting for *select()* to return. However if the remote communication thread is suspended in this way, a deadlock could occur if e.g. all processors sending data to this processor are waiting for data from this processor.

A concept is required to allow the remote communication thread to decide how it can be suspended without causing a deadlock.

Performance could also be increased by using a different method for the remote communication that makes better use of the SCC's specialized hardware (see section 1.4).

5.3.2 Extension to DAL

The current implementation could be extended to work with the distributed application layer (DAL) [3]. The DAL allows the dynamic execution of KPN applications on a target platform. This makes it possible that an application can be stopped at any time, possibly be mapped differently to the platform,

and then started again. Dynamic mapping can be useful if e.g. a part of the MPSoC consumes a lot of power and heats up. In such a scenario, the application could be stopped, mapped to a cooler region of the platform and then started again.

6

Conclusion

In this thesis, the DOL was extended to allow the execution of KPN applications on the SCC and on a networked Linux cluster.

A remote FIFO model was discussed. It implements a KPN channel with a FIFO endpoint on each processor running the connected processes. The data transfer from one endpoint to another is managed by a channel manager on each processor. Furthermore it was shown that flow control is necessary to avoid deadlocks in the KPN. The content of the FIFO endpoint of the sender is not consumed when transferred to the endpoint of the receiver. The sender is notified when data was consumed by its receiver allowing it to remove the data also in the sender's endpoint. This was implemented with simple synchronization messages that are generated and handled by the channel manager. To reduce network traffic, only one TCP/IP channel is used per communicating processor pair. This requires the (de)multiplexing of data (from)to the channel. This is also handled by the channel manager.

A DOL code generator is provided for each target platform. The application programmer has only to provide a simple architecture and mapping specification. The architecture specification contains the IP address and a port for each SCC core/Linux workstation to be used to execute the KPN. The programmer can arbitrarily map the processes of the application to the processors in the architecture specification.

The performance of the implementation was evaluated. Due to inefficiency, the benefit of using multiple cores is limited. This has to be addressed in future work. However, scaling seems to be promising for suitable applications.



Toolchain guide

The toolchain guide is a step-by-step instruction on how to integrate the new visitors *SCC* and *LinuxCluster* into the DOL framework. It is also explained how to set up and run the example application used for the evaluation (see section 5.1). The required files can be found on the CD provided with this report.

- How to build DOL with the SCC and LinuxCluster visitors.
 1. Add the *SCC* directory (found in *SCC.tar.gz*) to your DOL directory */dol/src/dol/visitor/*.
 2. Add the *LinuxCluster* directory (found in *LinuxCluster.tar.gz*) to your DOL directory */dol/src/dol/visitor/*.
 3. Add the *build.xml* file to your DOL directory */dol/* or add the include entries *dol/visitor/LinuxCluster/lib/*** and *dol/visitor/SCC/lib/*** to your existing *build.xml* file.
 4. Add the *runexample.xml* file to your DOL directory */dol/examples/*.
 5. Place the architecture specifications for the SCC (*scc.xml*) and the Linux cluster (*linuxcluster.xml*) in your DOL directory */dol/examples/arch/*.
 6. Build DOL by running *ant -f build.xml all* in your DOL directory */dol/*.
- How to generate code to run the example application on the SCC.

1. Add the *examplescc* directory (found in *examplescc.tar.gz*) to your DOL directory */dol/examples/examplescc/*.
 2. Depending on the number of cores the application should be executed on, change the following parameters
 - In */dol/examples/examplescc/examplescc.xml*, adjust `<variable value="X" name="NUM_OF_PAR_PORTS"/>`. Replace X with the number of *comp* processes that should be generated.
 - In */dol/examples/examplescc/map_scc.xml*, comment out unused *comp_i* processes (e.g. if X=2 was chosen above, comment out everything except *comp_0* and *comp_1*). The default mapping is the one used in section 5.1. The mapping can also be adjusted if necessary.
 - In */dol/examples/examplescc/src/global.h*, set *NUM_OF_PAR_PORTS* to the value X chosen above. Other parameters can be adjusted. These are described in *examplescc.xml*.
 3. Build the example by running `ant -f runexample.xml -Dnumber=scc -Dgenerator=SCC -Darchitecture=scc -Dmapping=map_scc` in your DOL directory */dol/build/bin/main/*.
- How to compile and run the application on the SCC.
 1. Enable the Intel *icpc* compiler on the MCPC.
 - Copy the *crosscompile.sh* (found in *SCC.tar.gz*) file to your home directory on the MCPC using *scp*.
 - Execute the script by running *source crosscompile.sh*.
 2. Copy the directory containing the generated example to your home directory on the MCPC (examples generated for the SCC are located in */dol/build/bin/main/examplescc/SCC/*).
 3. Compile the application by executing the Makefile located in the example's directory with *make all* (make sure the parameters are adjusted to your application as described in the last point of this guide).
 4. Copy the required libraries to the SCC cores used for the example.
 - Copy the libraries *scclib.tgz* and the script *setupLibSCC.sh* (both found in *SCC.tar.gz*) to your home directory on the MCPC.
 - Adjust *setupLibSCC.sh* so it will only copy and extract the libraries to the cores used by the application (e.g. if three cores are used, comment out the second for-loop and adjust the first line to be *for N in 0..2*).

-
- Execute the script to copy and extract the libraries to the specified cores by running `./setupLibSCC.sh`.
5. Distribute the executables to the cores.
 - Copy the script `setupAppSCC.sh` (found in `SCC.tar.gz`) to the same directory on the MCPC where the Makefile of the application is located.
 - Adjust the script to that the correct number of executables are copied to the cores (same as for the `setupLibSCC.sh` script).
 - Execute the script by running `./setupAppSCC.sh`. It recompiles the application if necessary and then copies the applications to the corresponding SCC cores (i.e. `sc_application_001` to `rck00`, `sc_application_002` to `rck01`, etc).
 6. Execute the process network.
 - Copy the files `hosts.txt` and `run.sh` (both found in `SCC.tar.gz`) to the MCPC home directory.
 - Adjust the `hosts.txt` file to only contain the cores which are used by the application (e.g. `rck00`, `rck01` and `rck02` if the application is mapped to three cores).
 - Start the application by executing the run script with `./run.sh`. This script uses `pssh` to start all executables on the cores at the same time.
- How to generate and run the example application on a Linux cluster. The procedure is very similar to that for the SCC. The differences are listed in the following.
 1. Adjust the architecture specification `linuxcluster.xml` to specify the IP of a workstation and the port to which other workstations will connect to. One entry is required per involved workstation.
 2. Adjust the mapping in `/dol/examples/example SCC/map_linuxcluster.xml`. The processor names have to match those of the architecture specification.
 3. The `runexample.xml` file has to be run with the parameters `ant -f runexample.xml -Dnumber=scc -Dgenerator=LinuxCluster -Darchitecture=linuxcluster -Dmapping=map_linuxcluster`.
 4. The scripts provided for the SCC cannot be reused for a Linux cluster because the IP addresses will generally be irregular. Either adjust the scripts or manually copy and run the executables to/on their corresponding workstations.
 5. The application can also be tested on a single workstation by running the executables in a separate shell each. The architecture specification must then contain entries `127.0.0.1:PPPP`

where PPPP is a port and must now be different for all entries. After code generation, the *bind_port* argument of the RCC's has to be added manually in the *processnetwork_part_XXX.cpp* files. Two RCC's that connect to each other have to have the same *bind_port*. More information can be found in the RCC class header file *RemoteChannelConnector.h*. Furthermore, adjust the function *setupListeningSocket()* in *RemoteChannelManager.cpp* (instructions are given at the beginning of the function).

- Adjustable parameters: Adjustable parameters for both the SCC and the Linux cluster implementation can be found in the header file *Packet.h* which is located in the *lib* directory of the corresponding visitor. The most important parameter is *MAX_FIFO_SIZE* which has to be set to the largest channel size from the process network specification. Further parameters include the number of bytes in the packet header which needs to be adjusted depending on maximum possible message size and the total number of processes in the process network. For more information, refer to the explanations in *Packet.h*.

B

List of Acronyms

API	Application Programming Interface
DOL	Distributed Operation Layer
IP	Internet Protocol
KPN	Kahn Process Network
LUT	Look-up table
MCPC	Management Console PC
MIU	Mesh Interface Unit
MPB	Message Passing Buffer
MPSoC	Multi-processor System on Chip
PaF	PipeAndFilter
RCC	Remote Channel Connector
RCDE	Remote Channel Demultiplexer
RCM	Remote Channel Manager
RCMU	Remote Channel Multiplexer
RRF	Remote Receiving Fifo
RRS	Remote Sending Fifo
SCC	Single-chip Cloud Computer
SSH	Secure Shell
TCP	Transmission Control Protocol

C

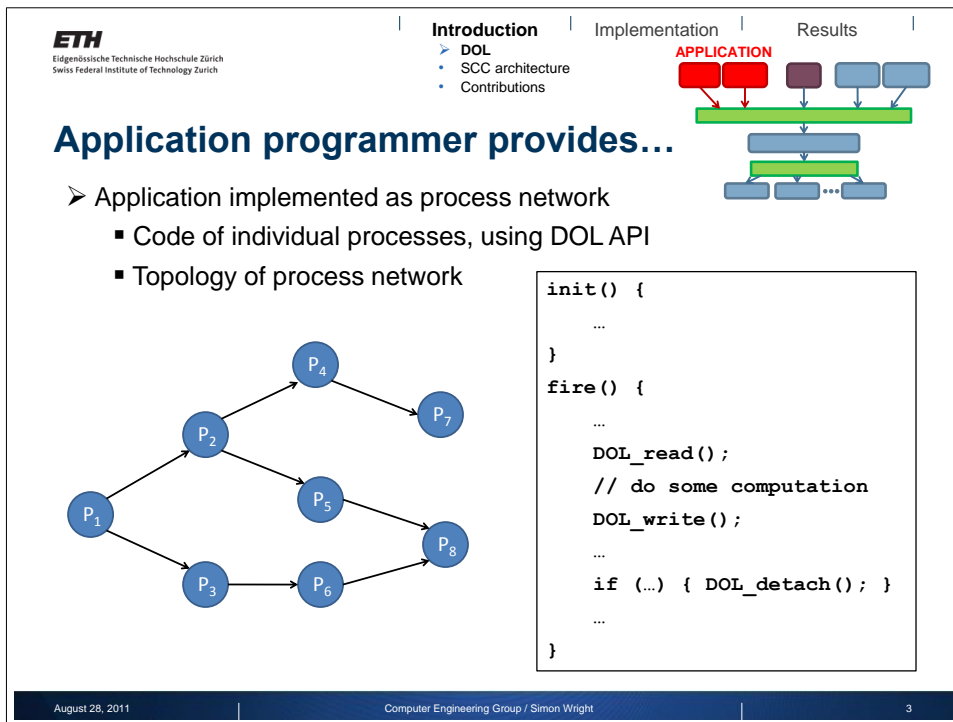
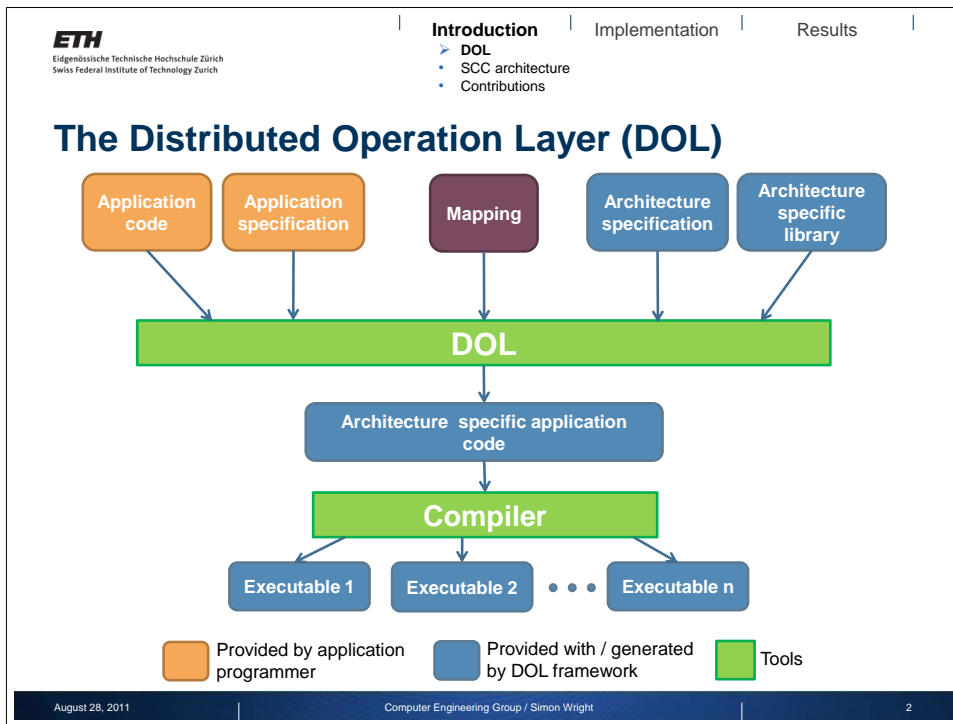
Presentation Slides

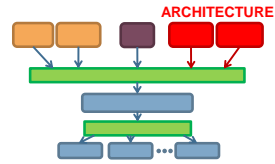
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Outline

- **Introduction**
 - DOL
 - SCC architecture
 - Contributions
- **Implementation**
 - Remote FIFO concept
 - Channel manager
 - Muxing / demuxing data
- **Results**
- **Conclusion**

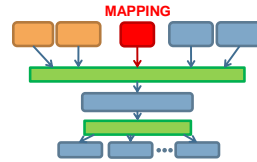
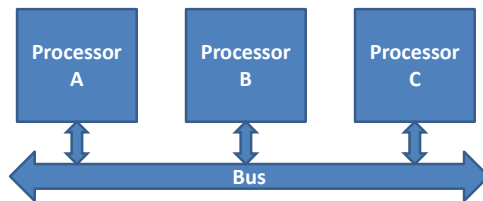
August 28, 2011 | Computer Engineering Group / Simon Wright | 1





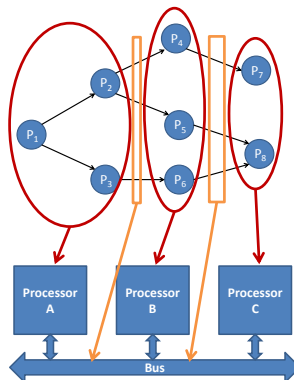
DOL provides...


- Structure of target platform
- Platform specific library
 - Implementation of `DOL_read()` and `DOL_write()`
 - Scheduling of shared resources



Mapping

- Elements of application specification mapped to / scheduled on target architecture
- Manual or automatic mapping





ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

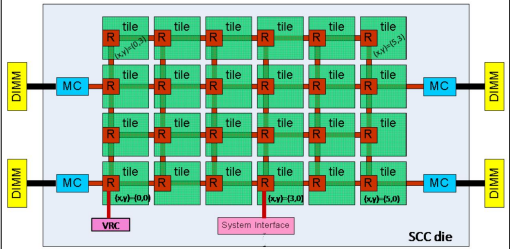
Introduction | Implementation | Results

- DOL
- **SCC architecture**
- Contributions

Intel Single-Chip Cloud Computer (SCC)


➤ SCC key features

- 48 cores (x86 instruction set)
- Network-on-chip: mesh topology
- L1 cache per core, shared L2 cache per tile
- Off-chip DDR3 memory
- Each core runs Linux OS



➤ **SCC is equivalent to a networked Linux cluster!**

August 28, 2011
Computer Engineering Group / Simon Wright
6



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

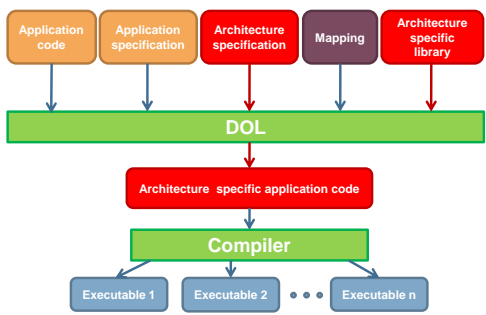
Introduction | Implementation | Results

- DOL
- SCC architecture
- **Contributions**

Contributions

➤ Execute process networks on SCC and Linux cluster

- Extend DOL framework for automation
 - Implement architecture specific library
 - Code generator for two architectures (given manual mapping)
- Scalable implementation

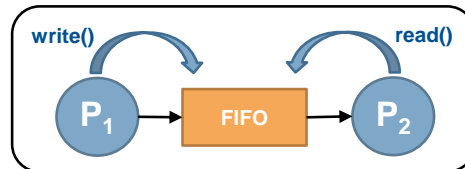


■ Provided by application programmer
 ■ Provided with / generated by DOL framework
 ■ Tools

August 28, 2011
Computer Engineering Group / Simon Wright
7

- Starting point
- Remote FIFO concept
- Implementation
- Muxing / demuxing data

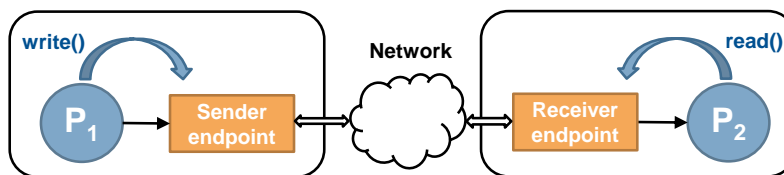
Starting point: PipeAndFilter DOL visitor



- All processes run on same processor
- Implementation using POSIX threads API
- Channels implemented as shared memory FIFO buffers
 - DOL_write(): write to FIFO
 - DOL_read(): read from FIFO

- Starting point
- Remote FIFO concept
- Implementation
- Muxing / demuxing data

Remote FIFO concept



- **Requirement:** processes can be mapped to different processors
- **Solution:** remote FIFO endpoints
 - Used by communicating processes on different processors
 - Uniform read/write interface
 - Transfer data over the network from sender endpoint to receiver endpoint

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction

Implementation

- Starting point
- Remote FIFO concept
- **Channel manager**
- Muxing / demuxing data

Results

Channel manager

➤ Channel manager (CM) orchestrates remote communication

- Collects / distributes data from / to its endpoints
- One CM per processor (scalability)

➤ Implementation

- CM as separate thread
- Communication via **socket API** (TCP) (compatible with both target architectures)

August 28, 2011

Computer Engineering Group / Simon Wright

10

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction

Implementation

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Results

Handling TCP/IP connections

➤ Non-scalable approach

- Each endpoint pair uses own TCP/IP connection

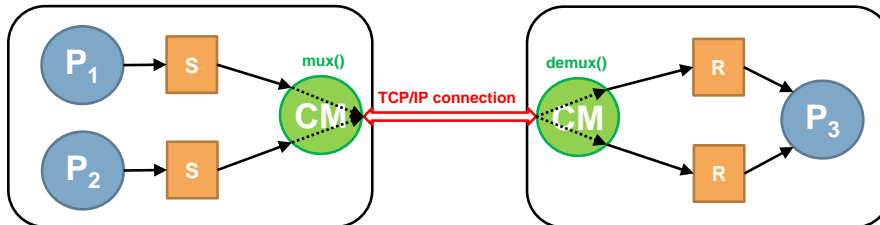
August 28, 2011

Computer Engineering Group / Simon Wright

11

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

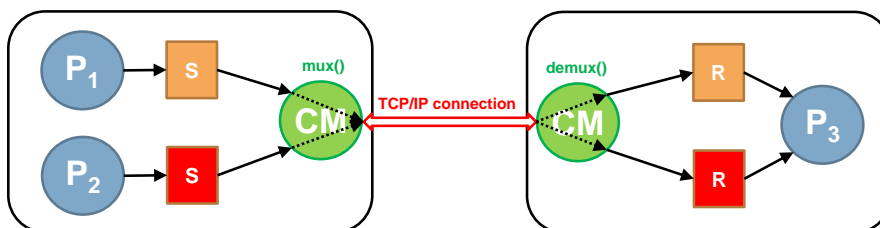
Muxing / demuxing data



- Non-scalable approach
 - Each endpoint pair uses own TCP/IP connection
- **Solution:** Only one TCP/IP connection between two processors
 - CM multiplexes data before sending
 - CM demultiplexes data after receiving
 - **Problem: Deadlock can occur!**

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Deadlock-free implementation



- **Solution: Synchronize sending and receiving endpoints**
 - Data not consumed when CM reads from sending endpoint
 - CM sends „synchronization messages“

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction | Implementation | Results

Example application

```

    graph LR
      generator{generator} --> comp_0((comp_0))
      generator --> comp_1((comp_1))
      generator --> comp_2((comp_2))
      comp_0 --> consumer{consumer}
      comp_1 --> consumer
      comp_2 --> consumer
  
```

August 28, 2011 | Computer Engineering Group / Simon Wright | 14

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction | Implementation | Results

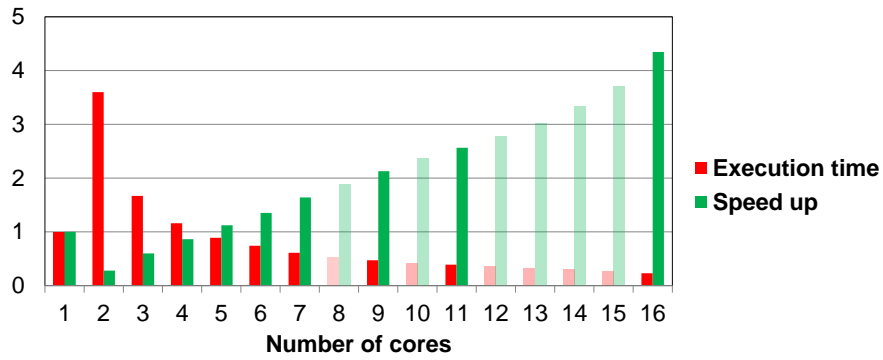
Mapping the application to the SCC

```

    graph TD
      subgraph Application
        generator{generator}
        comp_0((comp_0))
        comp_1((comp_1))
        comp_2((comp_2))
        consumer{consumer}
        generator --> comp_0
        generator --> comp_1
        generator --> comp_2
        comp_0 --> consumer
        comp_1 --> consumer
        comp_2 --> consumer
      end
      subgraph SCC
        Processor_A[Processor A]
        Processor_B[Processor B]
        Processor_C[Processor C]
        Processor_D[Processor D]
        Processor_A --- Processor_B
        Processor_A --- Processor_C
        Processor_A --- Processor_D
      end
      Processor_A --- generator
      Processor_A --- consumer
      Processor_B --- comp_0
      Processor_C --- comp_1
      Processor_D --- comp_2
  
```

August 28, 2011 | Computer Engineering Group / Simon Wright | 15

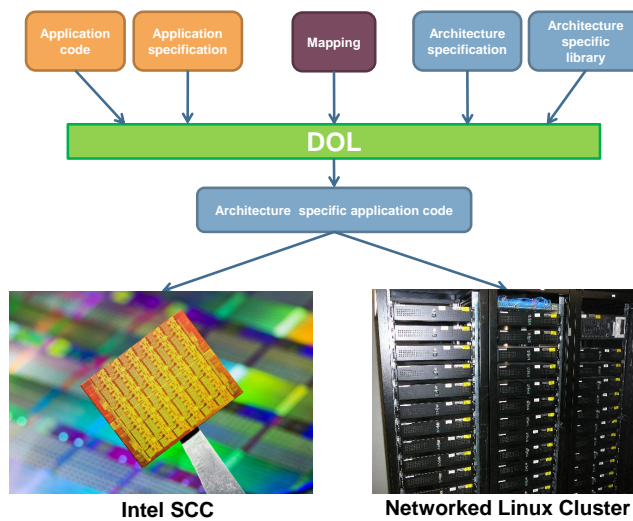
Results



- Poor speedup due to inefficiency in current CM implementation
- Good scaling with number of cores

$$\text{Execution_time}(n) = \frac{n-1}{n} \cdot \text{Execution_time}(n-1)$$

Conclusion



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Introduction

Implementation

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Results

B

Deadlock situation

➤ CM has input/output buffer

- Data is multiplexed to output buffer before sending
- Data is demultiplexed from input buffer after receiving

August 28, 2011
Computer Engineering Group / Simon Wright
18

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zürich

Introduction

Implementation

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Results

B

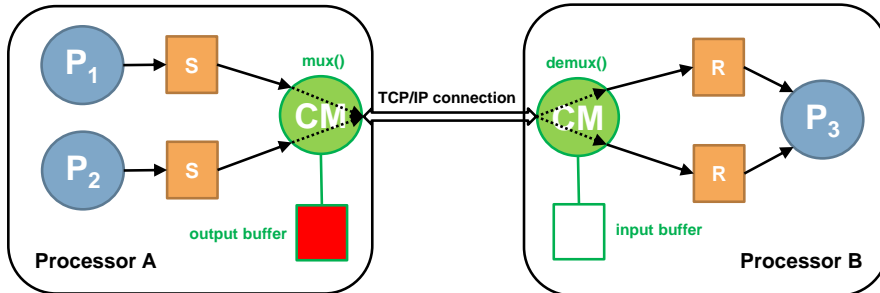
Deadlock situation

➤ Process 2 writes data to its sending endpoint

August 28, 2011
Computer Engineering Group / Simon Wright
19

- Starting point
- Remote FIFO concept
- Channel manager
- Muxing / demuxing data

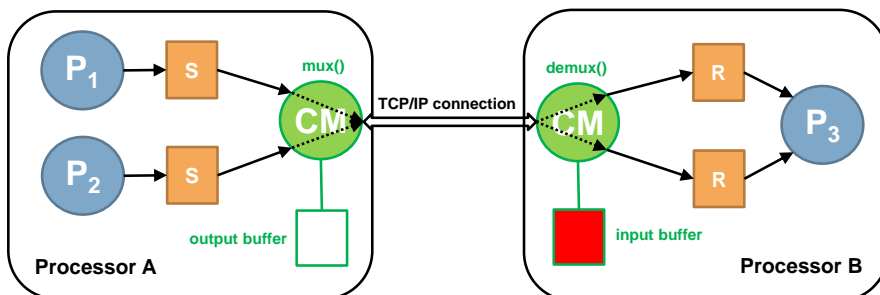
Deadlock situation



- CM A reads data, multiplexes to output buffer

- Starting point
- Remote FIFO concept
- Channel manager
- Muxing / demuxing data

Deadlock situation



- CM A sends data, data is received by CM B

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction

Implementation

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Results

B

Deadlock situation

➤ CM B demultiplexes data to receiving endpoint

➤ Process 3 can't consume data

- **Requires data from both input channels to do computation!**

August 28, 2011
Computer Engineering Group / Simon Wright
22

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction

Implementation

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Results

B

Deadlock situation

➤ Process 2 writes data again to its sending endpoint (before process 1)

August 28, 2011
Computer Engineering Group / Simon Wright
23

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction | **Implementation** | Results

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Deadlock situation

Processor A

Processor B

- Data is transferred
- Receiving endpoint is full, data remains in input buffer
- Input buffer is full, no more data can be received
- **Process 3 can never process data!**

August 28, 2011 | Computer Engineering Group / Simon Wright | 24

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction | **Implementation** | Results

- Starting point
- Remote FIFO concept
- Channel manager
- **Muxing / demuxing data**

Deadlock-free implementation

Processor A

Processor B

- **Solution: Synchronize sending and receiving endpoints**
 - Data not consumed when CM reads from sending endpoint
 - CM sends „synchronization messages“
- Avoids sending more data than fits in receiving endpoint
- **Input buffer can never be full!**

August 28, 2011 | Computer Engineering Group / Simon Wright | 25

Bibliography

- [1] Cell broadband engine. https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.
- [2] Daedalus. <http://daedalus.liacs.nl/Site/Daedalus%20home.html>.
- [3] Distributed application layer.
<http://www.tik.ee.ethz.ch/~euretile/dal.html>.
- [4] Mamps. <http://www.es.ele.tue.nl/mamps/>.
- [5] select(2) - linux man page. <http://linux.die.net/man/2/select>.
- [6] Tiler processor overview.
<http://www.tilera.com/products/processors>.
- [7] K. Vissers P. van der Wolf B. Kienhuis, E. Deprettere. An approach for quantitative analysis of application-specific dataflow architectures. *Proc. Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 338–349, 1997.
- [8] B. Barney. Posix threads programming.
<https://computing.llnl.gov/tutorials/pthreads/>.
- [9] J. Galowicz T. Bemmerl C. Clauss, S. Lankes. *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*. RWTH Aachen University, Chair for Operating Systems, December 2010.
- [10] T. Kangas et al. UML-Based Multiprocessor SoC Design Framework. *ACM Transactions on Embedded Computing Systems*.
- [11] B. Hall. *Beej's Guide to Network Programming - Using Internet Sockets*, version 3.0.14 edition, September 2009.
- [12] K. Huang L. Thiele I. Bacivarov, W. Haid. Methods and tools for mapping process networks onto multi-processor systems-on-chip. *Handbook of Signal Processing Systems*, pages 1007–1040, 2010.

- [13] Intel. Intel research :: Single-chip cloud computer.
<http://techresearch.intel.com/ProjectDetails.aspx?Id=1>.
- [14] Intel Labs. *SCC External Architecture Specification (EAS)*, revision 0.934 edition, April 2010.
- [15] Intel Labs. *The SCC Platform Overview*, revision 0.75 edition, September 2010.
- [16] Intel Labs. *The SCC Programmer's Guide*, revision 0.70 edition, August 2010.
- [17] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Information Processing '74: Proceedings of the IFIP Congress (1974)*, pages 471–475, 1974.
- [18] Wolfgang Haid Kai Huang. *C/C++ Coding Guide*. ETH Zurich, TIK laboratory.
- [19] L. Schor. Execution of process networks on the cell broadband engine. *Semester Thesis*, 2009.
- [20] R. van der Wijngaart T. Mattson. *RCCE: a Small Library for Many-Core Communication*, software version 1.0-release, document version 0.7 edition, May 2010.
- [21] TIK. Distributed operation layer.
<http://www.tik.ee.ethz.ch/~shapes/dol.html>.
- [22] I. A. Comprés Ureña. *RCKMPI User Manual*. Intel Braunschweig, January 2011.