Deterministic Task Transfer in Network-on-Chip Based Multi-Core Processors

THESIS

Jacobus Reinier van Kampenhout

Deterministic Task Transfer in Network-on-Chip Based Multi-Core Processors

---

THESIS

submitted in fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jacobus Reinier van Kampenhout
born in Beilen, the Netherlands

Berlin, 18 April 2011.

Computer Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Abstract

In this thesis we consider the application of multi-cores in safety-critical real-time systems, especially avionics. In our literature study we extract two major challenges. Firstly the unpredictability that comes from the concurrent access of shared resources (especially the on-chip interconnect) must be dealt with. To address this we propose to extend the concept of partitioning which provides fault containment, in combination with resource reservation at design time. The second challenge is to optimize the hardware usage without compromising on the determinism inherent to static mapping and scheduling. We propose mode-based mapping to deal with this, which allows to switch between multiple static schemes. We capture these concepts in a simple formal model. Mode-based mapping is enabled by task migration. The transfer time of tasks must be bounded, which requires guarantees on the Quality-of-Service (QoS) offered by the interconnect. Modern multi-cores feature Networks-on-Chip (NoC), which are packet-switching interconnects consisting of links and routers. Key to deterministic behaviour of NoCs this is avoiding contention, this can be achieved with flow control and buffering strategies based on resource reservation. We propose the use of transient modes to control the changes between the different modes in a NoC.

To evaluate different transfer methods we conducted a number of experiments on a 64-core processor that features a NoC. The experiments show that both data prefetching from the shared cache and the programmer accessible networks are suitable for deterministic task transfer. The former is twice as fast but the number and size of shared data objects must be limited because timing analysis of large coherent shared caches is not feasible. For all methods the maximum deviation from the mean values is constant (0,4 μs), and the standard deviation is under $\frac{1}{2}\%$ of the total transfer time. This shows that these methods are deterministic and that a tight bound on the transfer time can be determined. We conclude that private caches and scratchpads are suitable memory architectures for real-time systems, which can be supported by message passing and explicit communication through small shared memory regions. Mapping traffic at design time avoids contention, and isolation of traffic at the transfer level offers additional fault-tolerance. We propose a number of improvements for the transfer methods considered in our experiments that will enable guarantees on QoS. Our experiments confirm the feasibility of the proposed concepts.

Acknowledgements

This research was performed at: Fraunhofer Institute for Computer Architecture and Software Technology, FIRST
Kekuléstraße 7
12489 Berlin
Germany

Supervisor Fraunhofer FIRST: Dipl.-Inf. Robert Hilbrich

Supervisor TU Delft: Dr. Zaid Al-Ars

Contents

List of Figures

Chapter 1

Introduction

The research conducted for this master's thesis is driven by the increasing demand for computer-based systems in the aerospace industry. The term aviation electronics, or *avionics*, describes all the on-board electronic systems and components used in aircraft. In this thesis the focus is on embedded systems, a term which is used for computer systems that are integrated in their environment and closely interact with it. While avionics are the main focus of this research, the concepts that are presented can also be applied in fields where similar problems are faced.

1.1 Trends in the Avionics Domain

1.1.1 Federated Architectures

Embedded systems in the avionics domain must meet very strict requirements. The reason for this is that most on-board systems are *safety-critical*, which means that the failure of a system can have catastrophic consequences such as the loss of human lives. Thus the probability of a failure occurring in the hardware or software must be minimized by design. To achieve this a system must be *fault-tolerant* so that the system responds to a fault in such a way that complete failure is avoided. Fault-tolerance can be implemented by the addition of mechanisms that detect the failure and minimize its effect. A function is *redundant* if there are multiple independent instances of that function within a system. If the instances of a redundant function are implemented according to different design methodologies, the function is *dissimilar* which reduces the risk of design- and implementation faults.

An important aspect of safety-critical design is the *real-time* capability of systems. In real-time computing the correctness of a computation depends not only on the logical result, but also on the time at which this result is produced. Such dependencies are captured in timing constraints, a point in time at which a result must be available is called a *deadline*. An embedded

system is referred to as a real-time system if it is able to satisfy such timing constraints. Safety-critical systems on board of an aircraft are typically hard real-time, meaning that failing to satisfy the timing constraints has severe consequences. This is opposed to soft real-time, where the violation of a deadline does not endanger the system nor its user.

Another set of requirements originates from economic motives and concerns the size, weight and power consumption of embedded avionics systems. While the systems must be robust, they should be compact and lightweight so that an aircraft can carry as much valuable payload as possible. The main incentive for saving power is heat reduction. This is necessary because avionics systems must be able to run for a certain time without active cooling. Furthermore power savings lead to reduced fuel consumption and a reduction of the size and weight of power supply systems.

An approach to address these challenges that has been successful in the past decades is the use of a so-called "federated architecture". Federated architectures consist of a collection of independent computing platforms that are interconnected by a network. A distinct property of these architectures is that each function has its own dedicated computing resources.

1.1.2 Research Incentives

In every industry companies are keen to cut costs and increase functionality in order to gain advantage over their competitors. Manufacturers of avionics equipment are no exception to this. The costs for the embedded systems on board an aircraft can be split in development, operating and maintenance costs. The number of available functions and their individual complexity determine the overall functionality of an avionics system. In the avionics industry it is common practice to upgrade the functionality after some time in order to extend the lifetime of an aircraft; the costs associated with this will here be considered maintenance costs.

Development budgets are dominated by the costs for design and, because avionics systems are safety-critical, costs for certification. Design costs can be reduced by developing standardized components which, once certified, can be re-used with little additional (certification) costs. The certification costs are usually high because of the complexity of modern processors. Many processors exhibit unpredictable behaviour because of speculative features. This makes it difficult or even impossible to perform the software timing analysis that is required to certify real-time systems. In fact some modern processors cannot be certified for safety-critical systems at all. Thus the avionics industry favours deterministic processor architectures without speculative features.

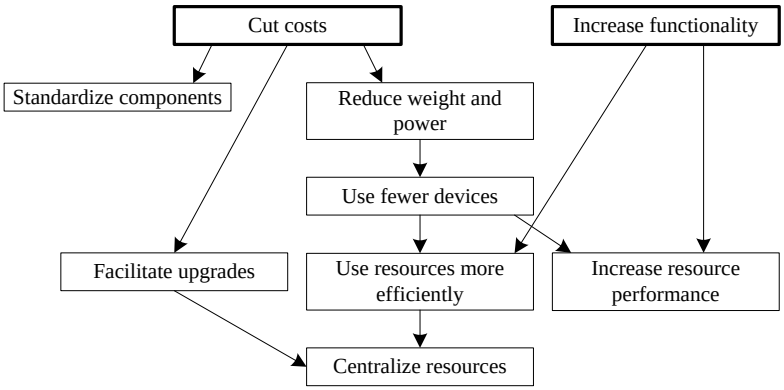

Figure 1.1: An overview of the research incentives in the avionics industry and their relations.

Considering the long lifetime of a typical aircraft, reducing the operating costs is an interesting option. One way to do this is to reduce the size, weight and power consumption of the avionics systems. For this the processor boards must either shrink in size, or their number must be reduced. To accomplish this, the hardware must be used more efficiently or the performance must be increased. Resource usage can be optimized if the amount of redundant hardware can be reduced. This can be achieved by *centralizing* resources so that fault-tolerant mechanisms can be shared among functions.

As most functional upgrades of modern avionics systems are software upgrades, the initial design should be able to facilitate these. The update of a software function should preferably not require re-certification of the whole system. Furthermore redundant computing capacity can be added to a system for future expansions or additions. A problem with this strategy in federated architectures is that computing capacity is fragmented, so the required spare capacity might not be available at the desired location. Again the centralization of computing resources is helpful, the spare capacity can then be shared between functions. This also corresponds with the desire to have fewer devices.

To increase the functionality of avionics systems one could simply add more hardware. This however contradicts with the desire to reduce space, weight and power consumption. Instead it would be better to aim for more efficient resource usage and increased performance of individual devices. A part of the gained capacity can then be used to increase the functionality while the total amount of devices can be reduced at the same time.

It has become clear that the incentives that drive the avionics industry

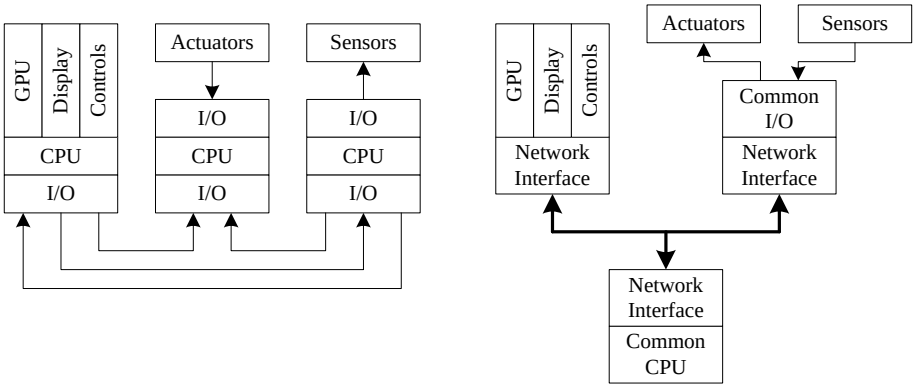

Figure 1.2: A federated architecture (left) versus Integrated Modular Avionics (right) [48].

lead to the desire to concentrate the functionality on fewer devices. This can be achieved either by optimizing or by increasing the performance of the devices. Furthermore standardization of components can help reduce both development costs and the complexity of the certification process. An overview is depicted in Figure 1.1.

1.1.3 Integrated Modular Avionics

The aforementioned considerations lead to a shift from federated architectures towards Integrated Modular Avionics (IMA), which aims to host multiple avionics functions on a shared computing platform [48]. Figure 1.2 depicts the architectural difference between the two approaches. The federated architecture on the left comprises of three CPUs, five communication modules and four physical communication channels. On the right an IMA architecture with the same functionality is shown. It consists of only one CPU, four communication modules and one common communication network. A description of the network is part of the IMA specification, often down to the hardware connectors. Whereas in the federated architecture the functions are deployed on separate CPUs which all have their own I/O modules and channels, in IMA the resources are shared between functions which leads to a reduction of hardware and thus weight and power. Although there are usually still multiple CPUs in a system, their number is greatly reduced compared to federated architectures. Because the resources are centralized software updates can be performed in less time which reduces the overall maintenance time.

To exploit those advantages there is however a new challenge that must be dealt with. Since functions are not physically isolated anymore, it becomes possible that they interfere with each other in a way that was not

intended by the system designers. A faults that occurs in one function can potentially affect many others. This is unacceptable for safety-critical functions and must be ruled out by design in order to build a safe system and to pass the certification process. Such *fault-containment* is a natural property of federated architectures, but is a major challenge in IMA.

To deal with this challenge, spatial and temporal *partitioning* of software is applied in order to guarantee isolation between functions. A system is considered to be strongly partitioned when a faulty function cannot interfere with or cause a failure in other functions [28]. In practice spatial partitioning is enforced by assigning each function a memory address space that cannot be accessed by others. Temporal partitioning is achieved by scheduling time slots in which one function has access to the resources while the others are suspended. These techniques push the actual problem of isolation to a higher abstraction level, namely to that of the Real-Time Operating System (RTOS) which must enforce the partition bounds. To standardize RTOS interfaces across the industry and thus enable re-use of components, the **ARINC Specification 653:** *Avionics Application Software Standard Interface* was developed [4].

The interfaces described in the ARINC Specification 653 establish a boundary between the functions and the RTOS. This means that the functions and the RTOS can be developed concurrently, and that the hardware platform can evolve independent of the applications [38]. The RTOS is responsible for allocating memory regions and processing time for each partition as well for as providing mechanisms for inter-partition communication. Fault-containment through isolation is crucial because it must be guaranteed that a faulty component cannot cause the failure of other components. Strong partitioning is often achieved through the combined functionality of the RTOS and the memory management hardware.

Although the main goal of IMA is to reduce the amount of total hardware, several other advantages have become apparent. Standardized interfaces facilitate the concurrent development of components. Furthermore it helps to avoid re-verification of the whole system when functionality is added later on and enables re-use of components. Abstraction improves the platform independence of software. The spare resources in an IMA architecture are centralized and can be allocated to any function. This flexibility leads to a reduction of the total spare capacity, thus reducing hardware requirements. A similar reduction can be obtained when mechanisms for fault-tolerance are shared between multiple functions. For example, a fault-tolerant memory controller can be used by all functions on a processor.

1.2 The Potential of Multi-Core Processors

1.2.1 Trends

In recent years the IMA approach has been successfully applied in aircraft such as the Boeing 787 and Airbus 380 [39]. In the future however power consumption and weight must be reduced even more, while the demand for functionality still increases. In order to do this, IMA must be taken one step further. The software architecture applied in IMA is suitable for this because it is composable; functionality is contained in partitions which are abstracted from the architecture on which they are deployed.

On the hardware side this approach requires an increase in the capacity of individual processor boards. Advances in technology ensure that new processors, memory chips and interconnect technologies become available frequently which allows the design of more advanced processor boards. In the past the increase in hardware performance has mainly been achieved by increasing the clock frequency. It is however impossible to continue in this fashion because the limits of Instruction Level Parallelism (ILP) have been reached and due to the so-called "power wall". This power wall means that the increase in clock frequency results in a disproportional growth in power density which in turn leads to an unacceptable increase in temperature. Therefore it has become common practice to increase processor performance by placing multiple computational cores on one die, the *multi-core* processors. This means however that the temporal scheduling of partitions on single-core processors, which is one of the foundations of the IMA approach, cannot be continued.

There are many new challenges that must be faced with the shift to multi-cores, especially in the field of software engineering. In multi-core processors a number of instructions can be executed at a given time which means either several applications can run concurrently on the same chip, or thread-level parallelism can be exploited to speed up a single application [25]. If there is sufficient parallelism in the application code, multi-core architectures provide an increase in processing power compared to single-core processors that run at the same clock speed.

When the avionics domain is considered, multi-cores are a promising technology to meet the increasing requirements on the short term. Even more, on the long term manufacturers may stop further development of single-core processors so that the use of multi-cores becomes inevitable. Thus the potential benefits of multi-core technology for avionics must be identified and the new challenges that arise must be addressed. This brings us to the first objective of this thesis, which is:

To investigate how multi-cores can be successfully applied in safety-critical systems, with special regard to the efficient usage of hardware.
The rest of this section describes some required background information after which Section 1.3 will gives the outline of this thesis.

1.2.2 Overview of Architectures

A multi-core processor contains several computational cores that are interconnected by a communication medium. Homogeneous multi-core processors feature a number of cores with the same architecture and Instruction Set Architecture (ISA). Heterogeneous multi-cores on the other hand contain at least one core with an architecture and instruction set that differs from the others. The architecture of the cores depends on the intended use, which can for instance be general purpose computing, signal processing or graphics processing. The complexity of the cores and the feature size determine how many can be placed on a single die. It is expected that future chips will contain hundreds or even thousands of cores [12]. There is a trend to brand processors with a several tens of cores or more as "many-cores", but in this thesis we will stick to the term multi-cores and consider many-cores as a subset.

Single-core processors usually feature a large off-chip memory, complemented with an on-chip cache hierarchy to hide the latency. In multi-cores this kind of organization is called a centralized *shared memory* architecture. Because the memory bandwidth is limited however, the memory access quickly become a bottleneck if the number of cores is scaled up. One solution to this is to implement distributed shared memory, in which the memory and address space are split up in order to divide the traffic over multiple paths. Another approach is to provide each core with a smaller private memory. While this is scalable, software engineering becomes more challenging because data consistency must be guaranteed. When each memory has its own address space the data must be moved explicitly. If the memories are implemented as caches, a cache coherency scheme is required which may be implemented in hardware. In many designs local memories as well as a larger central memory are used in order to combine the advantages of both. Any form of centralized memory requires global control mechanisms which limits the scalability. On the other hand, sharing data between cores requires more effort when private memories are used.

A central topic that multi-cores introduce to processor design is inter-core communication. There are two different paradigms for communication which depend on the memory architecture, namely shared memory and *message passing*. In memory mapped communication a common address space is

used from which shared data items can be accessed by any core. With message passing all communication must be programmed into the software explicitly. These communication principles have already been used for some time in multi-processor machines such as supercomputers. Such machines feature multiple computational cores that are spread over different chips, in contrast to multi-cores. This difference in physical scale means that on-chip multi-cores have some distinctive properties that separate them from multi-processors. The local proximity for instance leads to very short communication times, and transmission faults are less likely to occur because the physical environment is much better controlled. Thus the complexity of protocol stacks can be greatly reduced and latencies are in the order of a few clock cycles, which enables many new possibilities for parallel software design.

1.2.3 Multi-Cores in Avionics Systems

To exploit the potential benefits that multi-cores offer, avionics software must be ported to these new architectures while satisfying the requirements for safety and certification. Avionics software is *safety critical* as well as *real-time*, in this thesis we will further use these terms to clarify which aspect we mean. Real-time applications consist of sets of tasks that each have their own deadline. To make efficient use of the available hardware, these tasks must be scheduled sequentially on a computational core so that each task completes its execution before the deadline expires. Scheduling can be performed dynamically (online) or statically (offline). In safety-critical real-time systems static scheduling is applied because the correctness of such schedules can be guaranteed. In order to be able to schedule a set of tasks, execution time analysis is needed to determine the Worst Case Execution Time (WCET) and Best Case Execution Time (BCET) of those tasks. Those bounds must be safe and tight, i.e. they must never underestimate (overestimate) and the overestimation (underestimation) should be as small as possible. This can only be achieved if the behaviour of the hardware is predictable, which is not always the case in modern processors with speculative features such as branch prediction and out-of-order execution.

Thus, safety critical real-time systems need a *deterministic* hardware architecture. An event is deterministic if it is causally determined by an unbroken chain of prior events. This means that when a system is in a certain state and receives a certain input, the response must be predictable. Furthermore the time that elapses between receiving the input and the finalized transition to the next stable state must be predictable, or at least have an upper bound.

One of the challenges introduced by the application of multi-cores is the sharing of resources. When multiple computational cores are placed on a

single die, each will have some private hardware such as registers, a communication interface and perhaps a local memory. There are however also resources which are shared such as the main memory and the on-chip interconnect. This leads to conflicts when multiple cores try to gain access to a resource at the same time, which results in undeterministic behaviour if no additional measures are taken. Because all current multi-core architectures have shared resources, this challenge must be studied to be able to analyze its impact on the execution of real-time software. We will further go into this in Chapter 2.

1.3 Summary and Thesis Outline

In this chapter we introduced the avionics domain, which is the focus of this master's thesis. We discussed the unique requirements of this domain as well as the incentives for past and current research. The trends in avionics lead to an increasing interest in the application of multi-core processors. We presented the first objective of this thesis, which is to investigate how multi-cores can be successfully applied in safety-critical embedded systems with special regard to the efficient usage of hardware. Furthermore we introduced the terminology and presented a short overview of the main concepts in multi-core computing to provide the reader with sufficient background information.

In Chapter 2 we identify two major challenges that must be addressed when applying multi-cores in safety-critical real-time systems. The first concerns the predictability of hardware in multi-core processors, especially that of the interconnect. The second is related to optimizing hardware usage for which the deployment of software is discussed in more detail. In Chapter 3 we propose a combination of concepts from different fields to deal with these challenges. We capture these in a formal model and identify the direction in which we continue our research, after which we refine the objective accordingly. In Chapter 4 we go deeper into a technique that concerns all of our proposed concepts, namely task migration. We review related work and investigate the challenges of applying this technique in multi-core processors. This results in a list of requirements, one of which is new particular in this unique combination of domains. This leads to the second and final refinement of the objective.

In Chapter 5 we propose a number of experiments to evaluate the ideas presented in this thesis. We describe the approach and the experimental setup in detail, as well as the limitations. We present and analyse the results of these experiments in Chapter 6, and give the conclusions and recommendations in Chapter 7.

Chapter 2

Multi-Core Processors in Real-Time Systems: Challenges

In this chapter we present two major challenges that must be addressed when multi-cores are applied in safety-critical real-time systems. Section 2.1 focuses on the predictability of hardware in multi-cores, which is required for real-time applications. The main new component that may cause unpredictability in multi-cores is the on-chip interconnect, we go deeper into this subject in Section 2.2. The second challenge concerns the deployment of software on multi-cores, which determines the efficiency with which the hardware can be used. In Section 2.3 we discuss the mapping and scheduling of real-time applications on multi-cores.

2.1 Predictability

Whether the behaviour of real-time software is deterministic or not depends on the hardware of the computer system on which it is deployed. When the hardware behaves perfectly predictable, the execution of the software is deterministic and timing analysis can be performed to find an upper bound on the execution time. In this section we determine which elements in multi-core architectures could lead to unpredictable behaviour.

2.1.1 Execution Time Dependencies

The execution time of software on a multi-core processor depends on three main components: the computational cores, the memory hierarchy and the on-chip interconnect. Previous research has focused on timing analysis of single-cores to extract safe bounds on the execution time [18]. Such analysis is however increasingly difficult due to the rise in complexity of modern pro-

cessors [46]. Exploitation of instruction level parallelism such as superscalar and out-of-order execution lead to timing anomalies and complex dependencies between instructions. Thus cores with architectures that do not contain timing anomalies (i.e. a local WCET does not contribute to the global WCET) and have limited speculative features are recommended for use in real-time systems [50]. When many cores are placed on a single die, the complexity of the individual cores is usually less than the computational units in modern single-core processors. Existing timing analysis techniques should be sufficient to analyze the execution time of a sequence of instructions executed on a core.

The second main contribution to the execution time of a task comes from memory accesses. In [50] three recommendations for memory hierarchies on single-cores are given that enable the extraction of tight bounds on memory timing:

- either scratchpad memories or caches with a Least Recently Used (LRU) replacement policy should be used;
- instruction and data memory should have separate L1 caches;
- flat linear byte-oriented memory without paging is preferred.

In multi-core processors however the memory hierarchy is often more complex. Many architectures combine shared and distributed memories, and caches are applied at different levels to hide the latency gap between the cores and main memory. While the recommendations for single-core processors are also valid for each individual memory, shared memories introduce additional challenges. The interference of cores on any shared resource can lead to undeterministic behaviour, which makes WCET analysis hard and only feasible under certain conditions [42]. Therefore it is recommended to limit the use of shared memory and especially shared caches when possible, although the shared memory model enhances the programmability of multi-cores.

The third contributor to the execution time is the on-chip interconnect. In single-core processors the contribution and complexity of this part are relatively small, but in multi-cores the situation is quite different. Traditionally the computational core is the bottleneck of most computer systems. With the rise of multi-cores however, a large amount of computational power becomes available on a single chip. As the number of cores grows, so does the complexity of the interconnect that connects the cores to each other and to resources such as memory and I/O. The interconnect thus quickly becomes the new bottleneck because it must move around all the data that is produced and consumed by the cores. Traffic properties such as bandwidth and

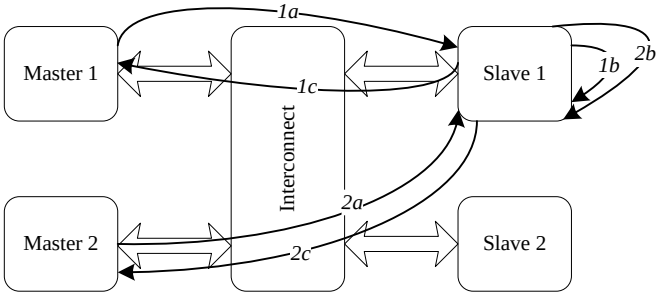

Figure 2.1: Two masters and two slaves communicating over a shared interconnect.

latency limit the speed at which computations can be performed. Therefore the architecture of modern multi-core processors becomes *communication-centric* in contrast to traditional computation-centric single-cores.

This paradigm shift to communication-centric design has major consequences for the software design and execution time analysis. Because an interconnect is a shared resource itself, it must be carefully designed in order to rule out undeterministic behaviour. Even if the interconnect hardware is designed properly, the complexity of inter-task communication leads to significant difficulties in traffic management. Proper design and programming of the interconnect is considered a major challenge for modern chip design [7]. Its impact on the predictability of the entire system is critical for the deployment of real-time software on multi-core architectures. In this thesis the timing analysis of memories and cores is considered trivial. These are however all connected to a interconnect, and the execution time depends heavily on the time required for communication. Therefore the we focus on the interconnect and the impact it has on the predictability of multi-core processors.

2.1.2 Modelling Interconnects

From the previous subsection it has become clear that access to shared resources is the main challenge in timing analysis of multi-cores. This access is provided by the interconnect, which is a shared resource itself. The remaining part of this section will therefore focus on the details of this problem after which Section 2.2 presents actual modern interconnects.

The situation where multiple entities access a shared resource can be modelled as masters accessing a slave via an interconnect. A master always initiates a transfer and will in this comparison be represented by a core. Slaves on the other hand are passive and only respond to requests, memory

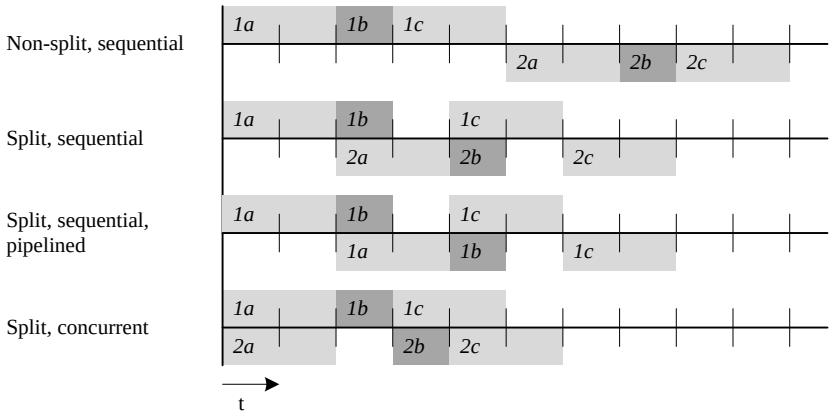

Figure 2.2: A scheme of the transaction phases of different connection types.

interfaces and I/O devices are typical examples of those. Cores can however also respond to other cores, so they can be master, slave or both depending on the program code they execute. Interconnects can be implemented by a bus, point-to-point or switched architecture. Figure 2.1 depicts a schematic view of two masters accessing the same slave. The different phases of the transaction of master one are labelled $1a$ through $1c$ and that of master two $2a$ through $2c$. A transaction consists of a request issued by the master followed by the acceptation and transportation of the request by the interconnect (phase a). The slave then accepts and executes the request (phase b). An optional response by the slave (phase c) is similar to phase a but goes in the other direction.
A basic interconnect is non-split, non-pipelined and sequential, which means that both master and interconnect can handle one transaction at a time. In Figure 2.2 the scheduling of the transactions of Figure 2.1 is depicted in discrete time slots It is clear that there is no concurrency in non-split sequential transactions. An improvement on this are *split* transactions, in which the request and response phases are decoupled. This means that a master is not blocked between sending the request and receiving the response, and that the interconnect can interleave requests and responses from different masters, see Figure 2.2. *Pipelining* allows multiple outstanding requests of a single master and thus increases the throughput of that master, split transactions are a prerequisite for this. An interconnect features *concurrency* when it can handle multiple requests and responses at the same time, which is independent from splitting and pipelining. This is again depicted in Figure 2.2, the communication is handled concurrently but the slave can only handle one request at a time. Note that non-split concurrent transactions are not pictured, those only bring an advantage when the masters communicate with different slaves.

2.1.3 Interference on Shared Resources

From here on the masters and slaves will be referred to as *nodes*, and requests and responses as *messages*. The predictability of the modelled hardware depends on the separate components and on the interaction between them. Concurrent access to shared resources occurs when multiple nodes issue a message to the interconnect at the same time, and when multiple messages arrive at one node at the same time. This includes the situation where one message is currently being processing when a new message arrives. In Figure 2.1 concurrent access to shared resources occurs when $1c$ and $2a$ are issued to the interconnect concurrently, or when $1a$ and $2a$ arrive at the slave at the same time. These events are deterministic only when the response of the system is predictable and when there is an upper bound on the response time. If the interconnect and the nodes can process multiple messages simultaneously, the problem persists if the maximum number of concurrently processable messages is exceeded.

The issue of concurrent access to shared resources is often dealt with by applying an *arbitration* mechanism. Such a mechanism accepts one message and delays the others. This can for instance be implemented by a "first-come-first-serve" or "round-robin" arbitration. The consequence of delaying messages is that a node has to wait before it may try again, which must be included in the response time boundary. If there is however no guarantee that the delayed node succeeds the second time, the response time becomes unpredictable.

2.2 On-Chip Interconnects

In Section 2.1 it has become clear that on-chip communication is crucial for the predictability of multi-core processors. In recent years advances in technology forced hardware designers to turn to a new class of interconnects based on networking principles. Such networks are the only scalable interconnects so far [24]. The concept of these will be presented here after which issues with the predictability will be analyzed.

2.2.1 Networks-on-Chip

As feature sizes continue to shrink, it becomes harder to interconnect the Intellectual Property (IP) blocks on a chip. While the number of transistors increases quadratically, the number of wires into an area only increases linearly because those are located at the edges of the blocks. At the same time the increase of clock frequency reduces the distance that a signal can travel

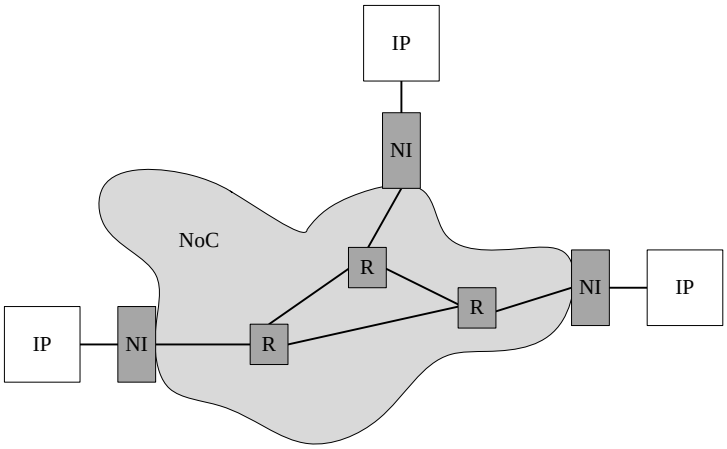

Figure 2.3: Schematic view of three IP blocks interconnected by a Network-on-Chip that consists of Network Interfaces (NI), routers (R) and links.

in a single clock cycle. The scalability of an interconnect in terms of chip area and capacity is crucial to facilitate large numbers of IP blocks. The named problems however cannot be addressed with traditional interconnects such as buses.

A concept that provides a solution is the Network-on-Chip (NoC), which is commonly used to interconnect IP blocks in a System-on-Chip (SoC) [17]. Such networks use packet switching to transport data, similar to off-chip networks such as local area networks. Because wires are shared the chip area dedicated to the interconnect is used more efficiently than with buses. For example, when two nodes are communicating on a bus all the others are usually blocked. In a NoC on the other hand only the links between them will be used and even on these other packets can be interleaved. This leads to the notion that NoCs are currently the only scalable solution for communication-centric design of modern chips [24]. When we consider the master-slave model presented in Section 2.1 we see that traffic is *split*, *pipelined* and *concurrent*.

A Network-on-Chip consist of three basic components. Each IP block must feature a Network Interface (NI) or network adapter that connects it to the actual network. The network itself contains a number of *routers* that forward packets arriving at the inputs to the correct output. The routers are interconnected by physical links. Figure 2.3 depicts three IP blocks interconnected by a NoC. When the IP blocks all have the same size, as is the case with multi-core processors, a regular array of square tiles can be laid out that each contain an IP block, an NI and a router with up to five connections. One router link connects the router to the NI while the others lead to

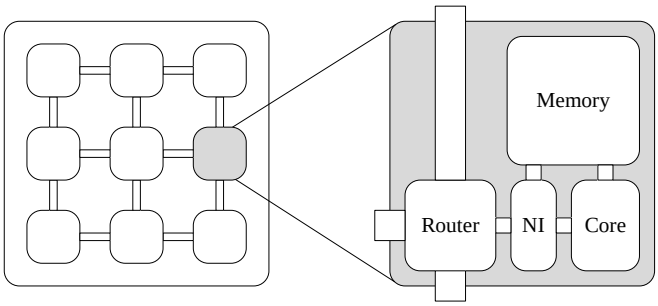

Figure 2.4: A two-dimensional mesh network with tiles that each contain a router, network interface, computational core and memory.

the neighbouring tiles in all four cardinal directions. Several network topologies have been proposed such as ring, cube, butterfly and fat tree networks. For regular on-chip networks however a two-dimensional mesh topology is the most advantageous as it is scalable and naturally fits onto a die. An example of such an architecture is depicted in Figure 2.4. It is obvious that the cost of external memory accesses increases as the number of cores grows, especially for cores in the middle. Besides the physical increase in distance each access will occupy the sparse resources at the edges of the chip, thus blocking others.

A common network concept used in NoC design is protocol layering. This offers several advantages through decomposition, abstraction and sharing. Decomposition of problems helps to manage the complexity. Abstraction hides details so that lower-level services can be used without having to worry about implementation details. Common services can be shared between different users so that they have to be implemented only once. A model for protocol layering that is often applied in NoC design is the Open Systems Interconnect (OSI) model. This model defines seven layers that each use the services of the underlying layer to implement their own service, which is then again offered to the layer above it. Five of those layers usually suffice to model a NoC. Those five layers, their description and the concerned components are listed in table 2.1.

2.2.2 Networking Concepts

A central problem in all networks is finding a path between two endpoints, or *routing*. Routing can be optimized with respect to certain parameters such as path length or traffic balance. When deterministic routing is applied, the path between source and destination is always the same. In adaptive routing on the other hand a path is determined while taking the network state into account. Path computation can be done either all at once, in which case the

Layer name	Components involved	Description
Application	IP block and NI	Offer various communication services to IP block
Transport	NI	Flow control, error correction
Network	NI and router	Send packet over a logical end-to-end link: routing, buffering, packetization
Datalink	Router	Send packet over a single logical link: access control, flow control
Physical	Router and link	Data transport over a single physical link

Table 2.1: Description of the OSI layers and their relation to the components of a NoC.

entire path is stored in the packet header, or incrementally in each router along the path. Routing deadlocks are the result of circular dependencies and should be avoided. A deterministic routing strategy that is often used is *dimension ordered routing*, which is deadlock-free provided there are no cycles when the network topology is traversed in a single dimension.

The links in a NoC and the buffers in the routers are shared resources. As explained in Section 2.1, sharing can lead to problems when multiple users request a resource at the same time. When multiple packets request the same link at the same time *contention* occurs. Contention can possibly cause *congestion* because packets have to wait for other packets, even if they do not want to use the link with contention. Congestion leads to large buffer requirements, increases latencies and renders the performance unpredictable. This hinders the adaptation of NoC based architectures in real-time systems, so congestion must either be bounded or avoided.

Congestion can be limited or prevented with proper *flow control* techniques that manage traffic streams. Circuit switching for instance is an example of buffer-less flow, and is similar to the connections in traditional telephone networks. With buffered flow control on the other hand one packet is forwarded over the requested link while the others are delayed and temporarily stored in buffers. Examples of this are store and forward, virtual cut through and wormhole flow control. While these techniques can reduce congestion, additional measures are necessary to be able to give guarantees on traffic properties.

2.2.3 Quality-of-Service

The term Quality-of-Service (QoS) is used to express the ability of a NoC to offer a certain level of performance to its nodes. A number of properties determine the performance of network communication as perceived by the nodes, such as:

- data correctness, delivery and ordering;
- bandwidth;
- latency;
- jitter (latency variation).

If these properties can be guaranteed, the traffic is deterministic and an upper bound can be determined on the communication time. Real-time software needs such "guaranteed service" to be sure that the deadlines are met. "Best effort" traffic on the other hand offers no guarantees on QoS and is therefore only of use for non-critical applications. These two types of traffic can be combined within one system [36].

With buffered flow control each router must have physical buffers that implement a certain *buffering strategy* such as input buffering, output buffering or virtual circuit buffering. A router furthermore features a switch that connects the input (buffers) to the output (buffers). A non-blocking switch can connect multiple inputs to multiple outputs simultaneously. Blocking switches introduce extra dependencies between the flows and are thus not favoured. Guarantees on QoS can only be provided with the right combination of a buffering strategy, switch architecture and switch arbitration.

It can be concluded that NoCs are switched interconnects that themselves pose a complex shared resources problem. This is because traffic may have to travel over multiple hops, and each link and buffer on the path is a another shared resource which might suffer from contention. Buffering schemes may reduce or eliminate buffer sharing at the cost of extra hardware. But even with an expensive buffering scheme and a non-blocking switch the wires are still shared. Thus an additional end-to-end scheduling strategy is required in order to guarantee QoS. Chapter 4 presents several solutions that offer guarantees on QoS.

2.3 Deployment of Software

The deployment of software on a multi-core processor has a temporal and a spatial component. Scheduling is the temporal allocation of tasks on a computational core. Mapping on the other hand is the spatial allocation

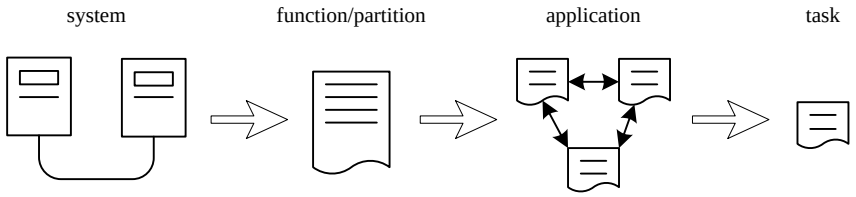

Figure 2.5: Overview of abstraction levels in embedded avionics systems.

of software artefacts to cores [37]. On a cluster of single-core processors, mapping would be the process of distributing the applications on these processors. With multi-core processors however the granularity is smaller, as software must be distributed among the cores on one processor. This section will elaborate on the challenge of deploying real-time software on multi-cores, the focus will be on the mapping process in particular.

2.3.1 Parallelism

In order to utilize the potential of multi-core processors in a system the applications must contain parallelism. That is, there must be multiple elements that can be executed concurrently. Parallelism can be found at different abstraction levels. On board an aircraft there are different physical embedded avionics *systems*. These each contain one or more *functions* such as providing cockpit information, communication services or radar image processing. A function can potentially be spread over multiple *partitions*, which are isolated (groups of) applications. For simplicity we assume a one-to-one relation between these levels here. Each function is implemented by one or more software *applications* that in turn consist of a number of *tasks*. In practice tasks are for instance implemented by processes or threads. Figure 2.5 presents an overview of these different abstraction levels.

To find a solution to the mapping problem, the abstraction level at which software should be mapped must be determined. Although there are often multiple applications within one system that can be executed in parallel, their number is most likely not sufficient to fill a multi-core processor when each application is mapped to one core. Thus, going one level lower and exploiting *task-level parallelism* is the approach used so far to tackle this problem [33, 13]. Still there are lower levels at which parallelism can be found. While these are too fine-grained for current multi-core processors, they might be interesting to consider for future processors which could feature hundreds of cores. In this thesis each partition is considered to consist of one application. This does not affect the validity of the proposed concepts because those are based on task-level parallelism.

A partition thus consists of multiple tasks which can communicate with each other. Because partitions are isolated from each other, special mechanisms are required if tasks from different partitions need to communicate. This is usually achieved through a specialized Application Programming Interface (API) that isolates the partitions from faults in the communications and thus allow inter-partition communication. Because of the concentration of functionality that is pursued in multi-cores, we assume in this thesis that all the elements of a single partition are mapped on the same processor. The kernel of the Operating System (OS) usually schedules the partitions, which themselves may have local schedulers to schedule the tasks [40]. Another approach is to schedule the tasks directly in the kernel, but this requires special care to protect the system from faults. Chapter 3 elaborates on partitioning.

2.3.2 Static Mapping

The input of a mapping process consists of the hardware and software properties as well as external requirements and constraints. An approach that has been successful in mapping real-time tasks onto processors is *static* mapping. Static mapping is performed at design time and tries to find a valid mapping that satisfies all the requirements, given that such a mapping exists.

For safety critical systems, static mapping is often combined with static scheduling to generate a scheme whose correctness can be guaranteed. Tasks are assumed to be cyclic with a fixed period so that a scheme can then be composed for the duration of a so-called hyperperiod. The period of the hyperperiod must be equal to or greater than the least common multiple of all task periods. This hyperperiod will be repeated in a cyclic fashion so that it can be guaranteed that each task always meets its deadline. A statically mapped and scheduled scheme can for instance be generated by applying constraint logic programming [19].

An example of static mapping and scheduling is depicted in Figure 2.6. In this example five partitions labelled S_0 through S_4 are scheduled on a processor with nine cores, which are laid out in a regular two-dimensional mesh structure. The hyperperiod is split in three time slots labelled t_0 through t_2 in which the partitions are mapped and scheduled. Some of the partitions need two time slots. The darker shaded links represent paths used for communication between partitions, while the lighter links are used for inter-task communication within a partition.

With static mapping it can be guaranteed that all the timing requirements are satisfied, which is a prerequisite for safety-critical systems. An obvious disadvantage is that the mapping process must be repeated when tasks are

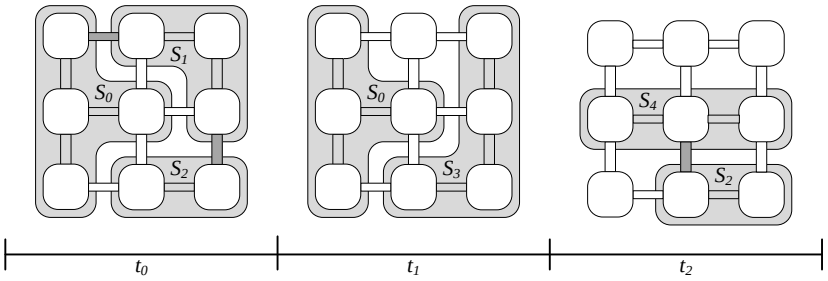

Figure 2.6: A scheme that depicts the static mapping and scheduling of five partitions $\mathcal{S}_0..\mathcal{S}_4$ on a 2-d mesh in a hyperperiod with three time slots $t_0..t_2$

added or deleted, or when the execution time of a task changes. In the avionics domain however changes to the system are often part of an upgrade which requires re-certification anyway, compared to which the generation and verification of a new schedule is a negligible effort.

Another disadvantage is that the hardware cannot be used optimally. This is due to three factors. Firstly, a time slot with the duration of the WCET will be reserved for a task in every hyperperiod, even when a task finishes much earlier on the average. Secondly, we need to distinguish between *periodic* (synchronous) and *sporadic* (asynchronous) tasks, the latter are triggered by external events such as the arrival of data. Those do not fit in the model because at design time it is unknown when they need to be started. Hence time slots must be reserved to be able to cope with the worst case. Thirdly, it might not be possible to occupy all resources due to the fixed number of tasks and their relations. In Figure 2.6 for example there are no tasks to occupy the empty cores in t_1 and t_2. We conclude that multi-core processors will probably not be utilized optimally under static mapping because resource reservation for the worst-case scenario is overly negative. This is a major limitation of static mapping.

2.4 Summary

In this chapter we pinpointed two major challenges for the adoption of multi-cores in safety-critical real-time systems. Firstly the execution of real-time software must be deterministic, which requires predictable hardware behaviour. Previous research addresses predictability and timing analysis of execution pipelines and memory systems. We found however that the predictability in multi-cores is largely determined by a new hardware component, namely the on-chip interconnect. The interconnect of multi-cores is much more complex than that of single-cores and is based on new concepts.

We presented a simple model of communication between nodes which showed that the unpredictability comes from the simultaneous access to a shared resource. In such a scenario one node must wait on the other to finish, which potentially introduces unpredictable delays. Deterministic behaviour of the interconnect requires careful design of both hardware and software to deal with this concurrent access of shared resources.

The only scalable on-chip interconnects used until now are Networks-On-Chip, which we described in Section 2.2. The links and buffers in a NoC are shared resources. When multiple users (packets) request the same resource at the same time, contention occurs. Contention can lead to congestion, which means that packets have to wait for other packets. This can in turn lead to unpredictable delays. The performance level that a network can offer is described by the "Quality-of-Service", a real-time application for instance needs guarantees on latency and bandwidth. This requires that contention and congestion are avoided or bounded. Certain combinations of routing, buffering and flow control can offer guarantees on QoS so that an upper bound on the time required for communication can be determined.

The second main challenge that we deduced concerns the deployment of software on multi-cores, which consists of mapping and scheduling. We found that parallelism on the task-level can be exploited for parallelization of software on multi-cores, but that those tasks must be encapsulated in partitions to guarantee fault-containment through isolation. In safety critical systems tasks are usually statically mapped and scheduled because it can be proved that such schemes meet the requirements. The main disadvantage of this approach is that hardware cannot be used efficiently because resources must be reserved for the worst case scenario, which is overly negative on the average.

Chapter 3

Partitioning, Mapping and Scheduling

This chapter introduces a combination of concepts from different fields in order to address the challenges described in Chapter 2. In Section 3.1 we present a formal model of software partitioning, which is used in the avionics domain to achieve fault containment. To deal with the challenge of interference on shared resources we extend this concept to include hardware in Section 3.2. In Section 3.3 we present an approach to match the software and hardware partitions which allows to optimize hardware usage. Combined with the formal model, this approach offers a solution to the challenges, but we also identify the need for further research and focus on that by refining the objective of this thesis.

3.1 Software Partitioning

In Section 1.1 we mentioned partitioning as a method to isolate applications in both space and time. Only isolation can guarantee that faults are contained and that the behaviour of an application is independent of that of others. Partitioning rules out mutual interference between tasks as possible cause for undeterministic behaviour, which is vital for real-time systems [21]. It has been successfully applied in IMA architectures with multiple single-core processor boards. In this section we present a formal description of partitioning which we will then adapt to multi-core processors. We also introduce an example that will be used throughout this chapter.

3.1.1 Temporal Partitioning

The partitioning process decomposes software into partitions that are supervised by an OS [28]. Each partition consists of a number of software components that are functionally related and closely interact with each other

during run-time. When multiple partitions are deployed on a single-core processor they are usually statically scheduled in time. Each partition is granted a predefined number of time slots in which it has access to the hardware resources. We denote the set of software partitions as:

$\mathcal{P}_{sw} = \{\mathcal{S}_0, \mathcal{S}_1, ..., \mathcal{S}_{k-1}\}$, where k is the number of software partitions.

Continuing this formal description, each partition consists of a set of software components (tasks) and a set of connections that describe the communications paths within a partition:

$\mathcal{S}_i = \{\mathcal{T}_i, \mathcal{A}_i\}$, for $i = 0, 1, ..., k-1$.

We describe the set of tasks $\mathcal{T}_i$ that each partition contains as follows:

$\mathcal{T}_i = \{\tau_{i0}, \tau_{i1}, ..., \tau_{i(l-1)}\}$, for $i = 0, 1, ..., k-1$ and where l is the total number of tasks in a partition i. We will elaborate on the communication paths described by $\mathcal{A}_i$ later.

The temporal scheduling process uses the absolute deadline and the WCET of each task to find a feasible schedule. The scheduling of real-time tasks is a complex problem which usually involves many more parameters such as the arrival time and task period. Here we will only cover the very basic scheduling principles needed to explain the concept of partitioning. To avoid unnecessary complexity in the model we make several assumptions are made that are unrealistic for most embedded systems. A more elaborate description of scheduling can be found in [5, 3] and particularly for multi-cores in [8].

We start with the assumption that the time axis is divided in hyperperiods that are repeated periodically. These hyperperiods contain a finite number of discrete time slots in which the partitions are scheduled. We denote a hyperperiod as:

$\mathcal{HP} = \{t_0, t_1, ..., t_{m-1}\}$, where m is the number of time slots in a hyperperiod.

We furthermore assume that a task must be scheduled exactly once during each hyperperiod for the duration of at least one time slot We can now express the relative deadline D and the worst case execution time E of each task in time slots:

$\tau_{ij} = \{D_{ij}, E_{ij}\}$, for $i = 0, 1, ..., k-1$ and $j = 0, 1, ..., l-1$, with $0 \leq D < m$ and $0 < E \leq m$. The deadline expires at the end of the time slot indicated by D.

In this simplified model the scheduling process must distribute the tasks over the time slots of a hyperperiod so that each task meets its deadline. The tasks that we consider are not pre-emptable, that is, a running task may not be suspended and continue execution later. The formalism presented so far is recapitulated in the following example, in which one partition that contains two tasks that must be scheduled:

$\mathcal{P}_{sw} = \{\mathcal{S}_0\}$
$\{\mathcal{S}_0\} = \{\mathcal{T}_0\}$
$\{\mathcal{T}_0\} = \{\tau_{00}, \tau_{01}\}$
with $\tau_{00} = \{3, 1\}$ and $\tau_{01} = \{2, 3\}$.

The hyperperiod consists of four time slots:

$\mathcal{HP} = \{t_0, t_1, t_2, t_3\}$.

The scheduling function denoted by $Sched\{\mathcal{P}_{sw}, \mathcal{HP}\}$ can now create a feasible schedule by assigning the tasks to the available time slots:

$Sched\{\mathcal{P}_{sw}, \mathcal{HP}\} = \{\tau_{01}, \tau_{01}, \tau_{01}, \tau_{00}\}$.

3.1.2 Spatial Partitioning

In single-core processors spatial partitioning concerns the memory and external resources such as sensors, actuators and I/O interfaces. Spatial partitioning of memory is achieved by assigning different memory areas to each partition, and blocking tasks in one partition from accessing the memory area of another. Access to external resources is inherently partitioned in time but may additionally only be available to specific partitions. Mechanisms for access restriction should preferably be implemented in hardware so that illegal access attempts are physically blocked. Thus fault are contained because a fault in one partition cannot affect the others. An example of this is protection from "babbling idiots" on an I/O interface [40].

A new dimension is added to spatial partitioning when it is applied in multi-core processors. In single-core processors access attempts to shared resources such as memory are inherently separated in time, and thus spatial partitioning is not very complex. In multi-cores multiple partitions are executed concurrently and may try to access memory or resources simultaneously. Thus more sophisticated blocking mechanisms are required. Furthermore tasks need to communicate with one another. As we already noted however, the interconnect that offers access to these resources is a shared resource itself. Thus it is required to partition the traffic on the interconnect, which avoids

simultaneous access of resources altogether so the blocking mechanisms can be similar to those in single-cores.

For such spatial partitioning of traffic the communication requirements of tasks must be captured. This allows us to *map traffic onto the interconnect* in addition to the mapping of tasks onto cores. If isolation between traffic streams can be guaranteed, we have extended the concept of spatial partitioning to include the interconnect and avoid contention at the shared memory and external resources.

It is a major new challenge to analyse and model the communication requirements of tasks. From here on we will only consider communication between tasks because those can act as both master and slave, but we need to keep in mind that communication with memory and external resources is very similar. Traffic between tasks can be split in *intra-* and *inter*-partition communication. We need the latter for the extension of spatial partitioning, but the former offers us a method to increase the predictability of task execution within a single partition. There are two different ways in which tasks can communicate with each other:

- through local memory if they are mapped on the same core;
- through the on-chip interconnect if they are mapped on different cores.

We assume here that all communication operations on an interconnect will only span one hop. This simplification means that two communicating tasks must be mapped either on the same or on two adjacent cores. We model these communication paths as a set of abstract *connections* contained in each partition. Details such as the traffic direction and bandwidth requirements are omitted here but can be attached to the connections as required. The connections of each partition are described as follows:

$\mathcal{A}_i = \{a_{i0}, a_{i1}, ..., a_{i(n-1)}\}$, for $i = 0, 1, ..., k-1$ and where n is the number of connections in a specific partition i.

There is one additional set which describes all the inter-partition connections, and is therefore added to the set of software partitions:

$\mathcal{P}_{sw} = \{\mathcal{S}_0, \mathcal{S}_1, ..., \mathcal{S}_{k-1}, \mathcal{A}_p\}$, with k the number of software partitions and where p is an identifier (not a number).

Many additional constraints can be thought of, a task can for instance be bound to a specific core because it needs access to a hardware resource only available at that location. Such parameters will not be considered here to limit the complexity of the model.

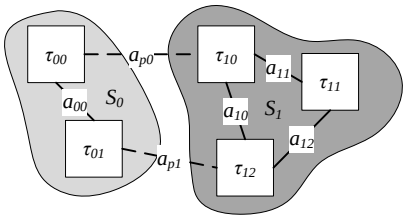

Figure 3.1: An example of two partitions with five tasks, four intra- and two inter-partition connections.

Inter-task communication between tasks mapped on different cores leads to an additional temporal constraint, namely that these tasks must run concurrently for a number of time slots. The amount of time slots can be expressed for every connection in $\mathcal{A}_i$. In the example presented in this section it is assumed that communicating tasks must execute concurrently for at least one, arbitrary time slot. This assumption is unrealistic for many systems because tasks must often communicate at a specific point during their execution.

An example of software partitioning that will be used throughout this chapter is depicted in Figure 3.1. The software consists of five tasks that are grouped in two partitions, $\mathcal{S}_0$ and $\mathcal{S}_1$. Connections between two tasks within a partition are represented by solid lines, while connections between tasks from different partitions are dotted. Note that the inter-partition connections are captured in a separate set of connections, $\mathcal{A}_p$.

3.2 Hardware Partitioning

In Section 3.1 we introduced the partitioning of software in time and space, possibly supported by hardware mechanisms. We then proposed to extend this in order to deal with the concurrent execution of multiple partitions, for which it is necessary to model the communication. The mapping of tasks onto cores and connections onto the interconnect are new concepts that must be addressed with the introduction of multi-cores. Such mapping requires a model of the hardware that we name *hardware partitions* onto which the software partitions can be mapped. In this section the concept of hardware partitioning is presented and the formal description is extended accordingly.

3.2.1 Abstraction of Resources

We now present an abstract model of the hardware and add it to the formal description. A multi-core processor is described with basic components, cores and links. The mapping process assigns these elements to a task or connection in each time slot. Thus spatial and temporal partitioning is accomplished. First of all the availability of multiple computational cores must be expressed in the formal description. A set of cores will be described as:

$\mathcal{C} = \{c_0, c_1, ..., c_{n-1}\}$, where n is the number of cores.

The mapping process is required to know the locations, so each core contains a set of x, y coordinates:

$c_i = \{x, y\}$, for $i = 0, 1, ..., n-1$.

Before the hardware is partitioned all the cores will be available. Hardware partitions to which the software partitions can be mapped must be composed by extracting a subset of cores for each software partition. The set of hardware partitions is denoted as follows:

$\mathcal{P}_{hw} = \{\mathcal{H}_0, \mathcal{H}_1, ..., \mathcal{H}_{k-1}\}$, where k is the number of hardware partitions.

It was already mentioned that partitioning the computational cores is not enough. To ensure deterministic behaviour, the communication operations must also be scheduled. Because all these operations go through the interconnect, it is crucial that the access to and behaviour of this interconnect is predictable. This can be achieved by also partitioning the interconnect in time and space. An interconnect can be described as a set of links:

$\mathcal{L} = \{l_0, l_1, ..., l_{q-1}\}$, where q is the number of links.

Each link connects two cores:

$l_i = \{c_u, c_v\}$, for $i = 0, 1, ..., q-1$ and u, v any two adjacent cores.

In a regular 2-d mesh network with n cores there are $2n(n-1)$ links in each dimension. Each hardware partition contains a set of links and a set of cores and is denoted by:

$\mathcal{H}_i = \{\mathcal{C}_i, \mathcal{L}_i\}$, for $i = 0, 1, ..., k-1$.

The subset of links for each partition is extracted from the set of available links similar to the assignment of cores to partitions. Note that each core

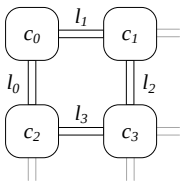

Figure 3.2: Partitioning of four cores and interconnecting links.

and link can be part of multiple partitions because temporal partitioning is also applied. We assume here that memory and external resources can be designed deterministically and therefore pose no additional challenge. An example of the partitioning of four cores and four links is depicted in Figure 3.2.

3.2.2 Composable Timing

In real-time systems the timing of software must be *composable*, that is, the execution time of a task must be independent of that of others. We propose to achieve composable processor timing by partitioning the hardware in time and space. Each hardware partitions consist of a number of dedicated components which are sufficiently isolated from each other. Within a software partition that is mapped onto a hardware partition different tasks and connections are scheduled locally. This is quite similar to composable *virtualization*, which can for instance be implemented through Time-Division Multiplexing (TDM) scheduling [31]. This same work presents four requirements for timing composability, namely:

- the schedule repetition period must be constant;
- this period must be divided in slices with the same size;
- the tasks must be assigned a constant amount of time slices;
- the same scheme must be executed in each period.

The model presented in Section 3.1 where a static scheme is repeated each hyperperiod on partitioned hardware satisfies these requirements. A notable difficulty with this approach is that all hardware components must be synchronized on a cycle-accurate level. This is not trivial in modern chips which often feature multiple clock domains.

We will now introduce temporal partitioning of hardware so that each core has its own set of m time slots in a hyperperiod:

$\mathcal{HP}c_i = \{t_0c_i, t_1c_i, ..., t_{m-1}c_i\}$ for $i = 0, 1, ..., n-1$ and where n is the number of cores.

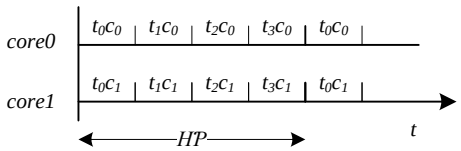

Figure 3.3: A hyperperiod partitioned in four time slots.

Because links must also have their own schedules, those also have a set of m time slots associated with them:

$\mathcal{HP}l_j = \{t_0l_j, t_1l_j, ..., t_{m-1}l_j\}$ for $j = 0, 1, ..., q-1$ and where q is the number of links.

An example of two cores and a hyperperiod consisting of four time slots is depicted in Figure 3.3.

3.3 Mode-Based Mapping and Scheduling

Now that we defined a model of the partitioned software as well as the hardware, we can look in more detail at the mapping and scheduling processes that combine these two to find construct a scheme. It is easy to see that static mapping and scheduling is feasible. This however leads to sub-optimal usage of the available resources as we described in Chapter 2. To improve the efficiency of resource usage we propose an extension of the static mapping approach in this section.

3.3.1 Optimizing Resource Usage

Each function in an avionics system is assigned a Design Assurance Level (DAL) based on the results of a safety-assessment that determines the criticality of a failure of that function. The DAL determines the software development process that must be followed as well as the allowed failure rates of the hardware, which is reflected in the required amount of redundant hardware. Because a lower DAL usually reduces the resource requirements, functions can sometimes be split and assigned different DALs to lower the overall requirements. This is called *segregation* and can be enforced through partitioning.

Section 2.3 described the concept and limitations of static mapping, namely that some of the available processing power in multi-cores is not used. This is caused by the dynamic behaviour of the software which cannot be addressed sufficiently with a static approach. The WCET might for instance

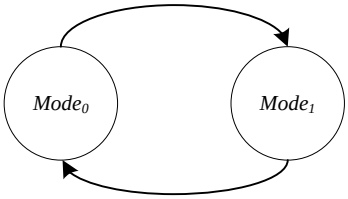

Figure 3.4: A state diagram of two modes.

be overly negative on the average, and tasks that are invoked sporadically might not need their time slots at all. We will try to exploit these dynamic elements to optimize the resource usage. Partitions with high criticality levels however must still have the same deterministic access to resources as they would get when statically mapped on a single-core.

From here on we differentiate between partitions that need to be statically mapped because of their high criticality, the *synchronous* partitions, and all others that are less critical, the *asynchronous* partitions. We propose a *mode-based* mapping and scheduling approach that improves the resource utilization while still guaranteeing deterministic access to resources. The optimization is made possible by varying the shape, size and location of asynchronous partitions. Each of these scenarios can be captured in a *mode*, a concept which is presented under the name use-case in [22].

3.3.2 The Mode-Based Approach

With a mode we mean an particular instance of the set of partitions. Static mapping and scheduling is performed at design time for each mode. Between modes the allocated resources for the asynchronous partitions may change, but those of the synchronous partitions remain the same. Thus we guarantee the advantages of static mapping for the synchronous partitions, but optimize the resource usage by exploiting changes in asynchronous partitions. During design time it cannot determined when a mode switch should occur so this decision is made during run-time, for instance triggered by a sporadic event. It is crucial that such a switch is a deterministic event. There can only be a finite number of modes, and for each of these a scheme is produced at design-time. Mode-based switching between static schemes thus compromises between static and dynamic scheduling and mapping. The software structure of the example depicted in Figure 3.1 has two modes which is illustrated in Figure 3.4.

An overview of the concepts that were presented in this chapter is depicted in Figure 3.5. Starting from the top, the software partitions are composed of tasks and connections. The resource requirements of each task are expressed

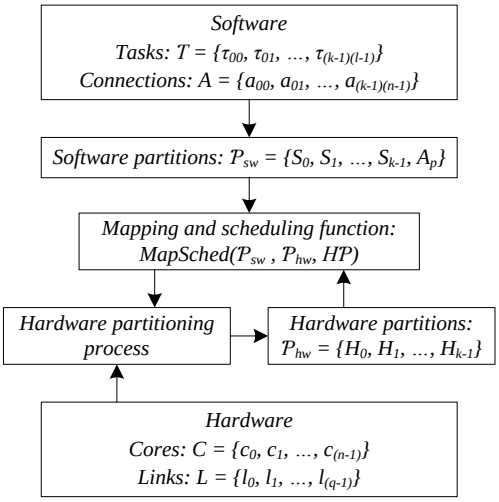

Figure 3.5: An overview of the mapping and scheduling process of software onto hardware partitions.

in the deadline, WCET, and the connections to other tasks. The mapping function instructs the hardware partitioning process to compose partitions that meet these requirements. This hardware partitioning process is part of the *MapSched* function, but is depicted as a separate block in Figure 3.5 to clarify the work flow. The scheduling function assigns time slots to the tasks and connections so that each task finishes before its deadline and overlaps with the tasks it must communicate with. The process can be iterative as the cycle in the figure indicates. Because the mapping and scheduling function strongly depend on each other they are combined in a single function:

$MapSched\{\mathcal{P}_{sw}, \mathcal{P}_{hw}, \mathcal{HP}\}$

Upon completion this process produces a scheme for each core and each link, described as:

$Scheme - c_i = \{t_0c_i = \tau_{uv}, t_1c_i = \tau_{wx}, ..., t_{m-1}c_i = \tau_{yz}\}$ for $i = 0, 1, ..., n-1$, where n is the number of cores and where $u...z$ are defined by the *MapSched* function.

$Scheme - l_j = \{t_0l_j = a_{uv}, t_1l_j = a_{wx}, ..., t_{m-1}l_j = a_{yz}\}$ for $j = 0, 1, ..., q-1$, where q is the number of links and where $u...z$ are defined by the *MapSched* function.

In the mode-based mapping approach we consider multiple different configurations of the software partitions, which require different schemes that are switched during run-time. Each mode consists of a complete set of schemes

	$Mode_0$		$Mode_1$	
Tasks	D_{ij}	E_{ij}	D_{ij}	E_{ij}
τ_{00}	3	4	3	4
τ_{01}	2	3	2	3
τ_{10}	3	3	2	2
τ_{11}	3	2	3	2
τ_{12}	2	2	3	3

Table 3.1: Two different modes for software consisting of five tasks distributed over one synchronous partition ($\mathcal{S}_0$) and one asynchronous partition ($\mathcal{S}_1$).

produced by the mapping and scheduling functions:

$Mode_i = \{Scheme - c_0, Scheme - c_1, ..., Scheme - c_{n-1}, Scheme - l_0,$ $Scheme - l_1, ..., Scheme - l_{q-1}\}$, for $i = 0, 1, ..., r-1$, where r is the number of modes, n the number of cores and q the number of links.

The software structure of the example depicted in Figure 3.1 has two modes, as illustrated in Figure 3.4. The different requirements for all tasks in each mode are given in Table 3.1. A hyperperiod consists of four time slots. It can be seen that partition $\mathcal{S}_0$ is synchronous while $\mathcal{S}_1$ is an asynchronous partition with changing requirements. The tasks of $\mathcal{S}_1$ must be re-mapped and re-scheduled because of their changed deadline and WCET. A mode switch is described by:

$ModeSwitch\{Mode_y, Mode_z\}$ for arbitrary y and z.

Such a mode switch results in one or more *task migrations* and *connection reconfigurations*. These two processes are formally described by:

$\mathcal{M}(\tau_{wx}, c_{source}, c_{dest})$ for arbitrary w, x, and

$\mathcal{R}(a_{wx}, l_{source}, l_{dest})$ for arbitrary w, x.

A possible mapping of the tasks onto the cores depicted in Figure 3.2 is given in Figure 3.6. The time axis depicts two hyperperiods. In the first hyperperiod the scheme of $Mode_0$ is executed, in the second a mode switch to $Mode_1$ has occurred. The lighter shaded tasks belong to static partition $\mathcal{S}_0$, their mapping and scheduling remains the same. The scheme $\mathcal{S}_1$ consisting of the dark shaded tasks however changes. This re-mapping of tasks comes at a cost because task τ_{11} must migrate to another core:

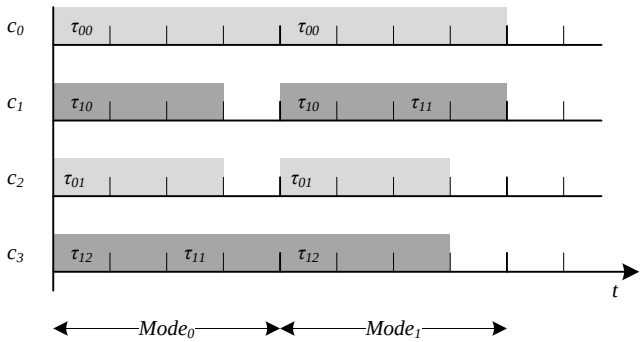

Figure 3.6: Five tasks scheduled on four cores in two different modes.

$\mathcal{M}(\tau_{11}, c_3, c_1)$

Such a migration costs and time and resources. Task migration is a critical component of mode-based mapping and must also be deterministic. We will go deeper into this subject in Chapter 4.

The connections can be mapped onto the links using the information from Figure 3.1 and 3.6. Let us assume in this simple example that the migration cost is one time slot on one link. Figure 3.7 depicts the scheduling of the links in the two different modes. In this case no connection is relocated, but one is removed and another added instead:

$\mathcal{R}(a_{11}, l_2, del)$ and $\mathcal{R}(a_{12}, add, l_2)$

The migration of the task takes place in the block labelled with M. While there happened to be an available time slot on the required link in this particular scheme, this time slot must be reserved even when there is no mode switch. Alternatively one or more *transient modes* may be defined, more on this in Chapter 4.

3.4 Summary

This chapter started with the definition of a simplified formal model of temporal and spatial partitioning of software, which is a concept used in IMA to achieve fault-containment. We then proposed to extend spatial partitioning to include inter-task communication so that the traffic can be mapped onto the interconnect. This requires to reserve the resources for communication, which solves the problem with unpredictability that results from simultaneous access to the interconnect. In our formal model a software partition

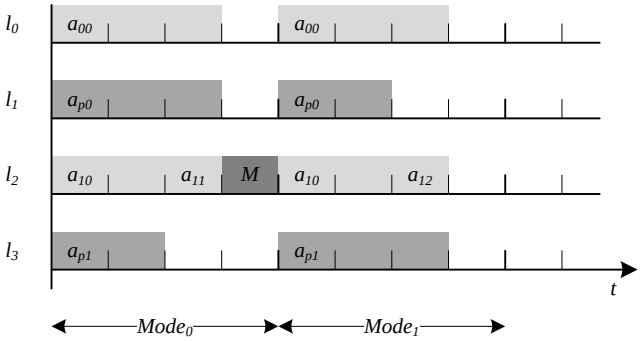

Figure 3.7: Six connections scheduled on four links in two different modes. The task migration is labelled with an M.

consists of a set of *tasks* as well as a set of *connections*. We furthermore proposed the concept of hardware partitions onto which the software partitions can be mapped. Such partitions consist of a set of cores and a set of links. Each of these elements is furthermore partitioned in a number of time slotswithin a hyperperiod in order to have composable timing.

The formal model of software and hardware partitions allows us to statically map and schedule the former onto the latter. Partitioning is enforced by isolation, which allows functions with different criticality levels to be mapped onto one multi-core processor. We use this flexibility to address the second major challenge, the sub-optimal usage of resources inherent to static mapping. We propose a mode-based mapping and scheduling approach which offers static mapping for critical, synchronous partitions but allows to switch between multiple, pre-computed configurations of the less critical asynchronous partitions. This mode switching requires the migration of tasks and reconfiguration of links, which must be accounted for in the pre-computed schemes.

In Chapter 2 we extracted two major challenges from our main objective. Firstly the unpredictability resulting from simultaneous access to the on-chip interconnect must be addressed, and secondly we must deal with the sub-optimal hardware usage that results from static mapping and scheduling. The concepts proposed in this chapter offer solutions to both of these challenges. The application of these concepts in this new field however still requires much research, which is too much to cover in one thesis. Therefore we focus on one of the main enabling techniques that we discussed, namely the migration of tasks. Thus we come to the second objective of this thesis: **To determine the prerequisites for deterministic task migration over a Network-on-Chip.**
This theme requires us to deal with all the new concepts introduced by

multi-cores: tasks must be mapped onto the cores, and to perform the actual migration the traffic must be mapped onto the interconnect. This implies that we must partition the interconnect and therefore deal with fundamental networking concepts. In Chapter 4 we look at task migration over a NoC in detail, after which we perform a number of experiments to put our approach to the test.

Chapter 4

Task Migration

In this chapter we present the concept of task migration, which is required for mode-based mapping. We start with an overview of the motivations for task migration in Section 4.1. In Section 4.2 we describe the basics of task migration and discuss several recent implementations. We then investigate how these concepts can be applied to achieve deterministic migration in NoC-based multi-cores in Section 4.3.

4.1 Motivation

In this first section we present several motivations for task migration. From Chapter 3 it has become clear that the main reason for our interest in task migration is to optimize hardware usage by means of *load balancing*. There are however several other potential benefits. Firstly, considering data locality can reduce the overall communication requirements and thus improve performance. Furthermore task migration enables software management of the power consumption and temperature. And last but not least, the flexibility in fault-tolerant systems can be greatly improved.

4.1.1 Load Balancing

In Chapter 2 we elaborated on the challenges that are faced when multi-cores are applied in safety critical systems. One of the major problems is that hardware usage is not optimal with static mapping because of the dynamic behaviour of software. The solution to this that we presented in Chapter 3 allows to balance the load, which means spreading the workload as evenly as possible over the cores so that the overall usage of the processor is optimized [30, 45]. This can be achieved by analyzing run-time variations in the workload and changing the mapping so that tasks *migrate* from overloaded cores to idle ones. The mapping and migration processes however have certain overhead so there cannot be an infinite amount of migrations.

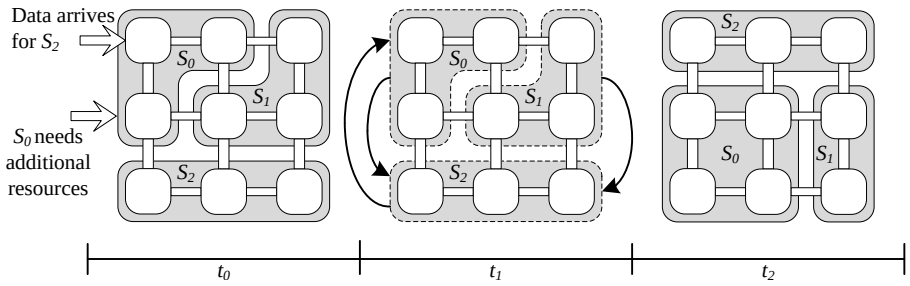

Figure 4.1: A mode switch: three partitions are transformed and relocated, balancing both the traffic and computational load.

The migration time of real-time tasks must furthermore be bounded, which means the migration process must be deterministic.

Load balancing is not possible with static mapping. Dynamic mapping on the other hand is very well able to respond to changes the behaviour of tasks and the resulting variation in resource requirements. Recently several mapping algorithms based on heuristics have been proposed [2, 15, 32]. Because it is difficult to prove that such algorithms always find a feasible mapping, dynamic mapping is not yet considered a solution for safety-critical systems.

The mode-based mapping approach that we presented in Chapter 3 allows to switch between different static mappings and is thus a compromise. Within one mode the mapping process can balance the load. A profile of the software must be determined at design time to uncover properties such as the WCET and traffic requirements. The resource requirements of asynchronous partitions vary over time. Such a partition might for instance monitor an external device and start multiple computation intensive tasks when an event is detected. Thus the shape, size and location of asynchronous partitions vary between modes according to the resource requirements. Figure 4.1 illustrates this with an example. In time slot t_0 partition $\mathcal{S}_0$ requires additional computational resources. This triggers a mode switch in time slot t_1, in which all three partitions are transformed and relocated. The mode switch is completed in time slot t_2, partition $\mathcal{S}_0$ is assigned an additional core. The changes to partition $\mathcal{S}_2$ are discussed later on.

4.1.2 Data Locality

In a NoC-based multi-core with shared memory the interconnect must handle at least three types of traffic:

- shared memory accesses;
- communication between cores;

- accesses of external resources such as I/O interfaces.

We assume that the shared memory and external resources are off-chip, and that each core has a small local scratchpad memory or cache. Data accesses can then be arranged according to their speed as follows, starting with the fastest:

- data in the local scratchpad or cache;
- data provide by another core;
- data provided by the main memory or external resources.

As each data access contributes to the WCET, it should be known at design time in which memory the data lie. If the data are not in the local memory, they must be retrieved via the interconnect. This leads to the previously described problem of concurrent access to shared resources. As mentioned this can be dealt with by reserving the path between the requesting core and the source of the data. Having these paths as short as possible reduces both the latency and the number of reserved links. Thus we establish two rules of thumb: data accesses should be localized whenever possible, and otherwise the path length between two communicating actors should be minimized.

The first rule means all static data should be copied into the local memory prior to task execution. Data produced by another core or obtained from an external source must be taken into account by the mapping process, which should map the tasks so that the path length is minimized. Note that this is opposite to an approach commonly used in SoCs, where traffic is analyzed in a statically mapped scenario after which the link capacities of the NoC are adjusted [11].

The validity of this approach is challenged if the traffic requirements change over time. It could then be desirable to change the mapping online to balance the traffic, very similar to load balancing for varying computational requirements. The one-time migration of a task can then avoid many communication operations over a long distance. In other words, the program code is moved to the data instead of vice versa. Switching between modes that are optimized for different scenarios can again provide a solution that is acceptable for safety-critical systems. An example is a partition that needs to interact heavily with an external resource which is connected to an I/O interface at the edge of the chip. Relocating the partition can then reduce the path distance, latency and overall communication requirements. This is also depicted in Figure 4.1, where data destined for partition $\mathcal{S}_2$ starts to arrive in time slot t_0 at the top left core. By relocating $\mathcal{S}_2$ to this location it can directly process the data so that the traffic does not have to go through the other partitions.

4.1.3 Power Management

Embedded systems are often subject to strict constraints on the power consumption. Techniques such as power gating and the dynamic scaling of voltage and frequency can be used to save power by slowing down or shutting down computational cores. Such power management must closely cooperate with mapping and scheduling to avoid unacceptable performance degradation. If the mapping can be changed online it might be possible to concentrate the running tasks on a number of cores in some modes so that other cores can be slowed down or shut down completely.

Power management also enables regulating the temperature of a processor. If a certain part of the chip is overheated, tasks can be moved away from those "hot" cores. A dynamic mapping approach can benefit from this more than mode-based switching because every mode in the latter is extensively tested during design time to avoid such situations. It is not unthinkable however that an emergency mode is activated in case a processor is overheated, in which all the non-critical tasks are shut down and the critical tasks are spread across the chip using task migration.

All the concepts mentioned so far require that the mapping process optimizes with respect to multiple parameters. There should be clear priorities, as some requirements may conflict. Power saving might for instance pack as many tasks on one core as possible, while thermal management and load balancing requirements at the same time indicate that tasks should be spread more evenly. A comprehensive analysis of mapping strategies is outside of the scope of this thesis, but it is clear that mapping and scheduling are not trivial problems.

4.1.4 Redundancy

Safety critical systems must be *fault tolerant*, that is, they must be able to continue operation when a part of the system fails. This is often achieved by some form of *redundancy*, for instance by replicating hardware and software components. The level of redundancy that a function requires depends on its criticality. A highly critical function might need to be replicated several times on multiple independent hardware units. Functions with lower criticality may have to be only partly replicated, possibly on the same chip. Fault-tolerance and performance usually have contrasting requirements.

Multi-core processors offer a platform where software can easily be replicated to achieve fault-tolerance. This will at the same time cancel out a part of the expected gain in performance. There is however one major difference with legacy systems: in those the hardware that provides redun-

dancy is often statically reserved for that purpose. In multi-core processors the trade-off between fault-tolerance and performance can be more dynamic because replicated tasks can be mapped and migrated as any other. This flexibility in assigning redundant hardware to partitions is another advantage enabled by task migration [16].

Trends in technology scaling allow the placement of more and more cores on a single chip. This however also increases the sensitivity to external interferences such as transient faults [44]. The flexibility in assigning redundant resources is a suitable concept to deal with this problem. A function must be protected against several possible faults based on the criticality level. Some faults affect the entire operation of the chip, such as the failure of a power supply circuit. Those cannot be resolved by implementing redundancy within the processor. Transient and persistent faults however usually affect only a small area of a chip and therefore it is likely that the execution sequence of only one core will be influenced. Precisely those faults can be resolved by executing a task concurrently on multiple cores.

4.2 Related work

There has been interest in task migration since the early days of multiprocessing. In this section we present task migration and give an overview of relevant work. We start with the basics and then evaluate several high-level approaches to the problem. Finally we present a number of recent implementations involving real-time aspects and Networks-on-Chip.

4.2.1 Basics

We use the term *task* to describe an instance of a program that is executed on a core. On the operating system level the term *process* is commonly used, which also comprises the stack and OS-specific state. Here we focus is on low-level task migration or task *transfer* discard and OS-level objects that might belong to a process. In this view a task consists of the following elements:

- executable *code*, located in the program memory;
- task-specific *data*, located in the data memory;
- *context* which contains the processor state, e.g. the registers contents.

Migration is the act of transferring a task from a source to a destination node. Both nodes must support the instruction set that the task needs to execute, but do not have to be completely homogeneous [43]. The mechanism that performs the migration can be implemented at different levels,

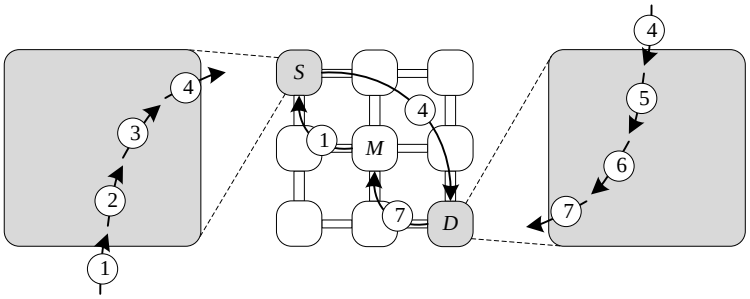

Figure 4.2: Impression of the seven basic migration steps on a NoC-based multi-core processor with nine cores laid out in a two-dimensional mesh architecture.

ranging from the OS-kernel to the application level. We describe a migration with the following basic steps:

1. initiation of the migration by the mapping process;
2. halting the task;
3. storing the context and modified data on the source node;
4. transferring the code, data and context to the destination node;
5. restoring the context on the destination node;
6. restarting the task;
7. signalling the completion of the migration.

Figure 4.2 shows these steps on a nine-core processor with a two-dimensional mesh architecture. The mapping process is indicated with an M, the source node with an S and the destination node with a D. Although quite simplistic, these steps suffice for a rudimentary model of task migration in embedded systems. A comprehensive survey of more elaborate migration techniques can be found in [30].

In processors with a distributed memory model all data and code memory must be transferred explicitly. When shared memory is used only a part of the data and code might be loaded locally (e.g. in a cache) while the rest is located in the main memory. In this situation the modified ("dirty") data must be transferred from one core to another but the address space does not have to be moved, so the cost is much lower.

Different classes of migration are listed in [45]. Tasks can be migrated while running or while inactive, for instance when suspended by the scheduler. In

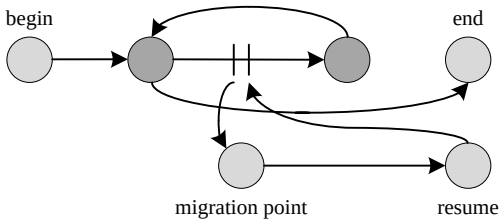

Figure 4.3: A task with a migration point in its main loop [9].

the first case the task must be halted and context must be stored as described, but in the latter the task can probably be migrated without steps two, three, five and six. If a task has finished rather than being suspended, there might be no context at all.

The motivations presented in Section 4.1 promise that task migration enables several advantages. It has now become clear that these come at a cost, and that a sensible trade-off should be made. The overhead of each step involves the use of several hardware components, namely:

- the mechanism that triggers the migration;
- the source core and its memory and router;
- the interconnect;
- the destination core and its memory and router.

The total migration time consists of the times for which each of these components are used.

4.2.2 Strategies

Since systems are seldom designed with task migration in mind from the beginning, the addition of a transparent migration mechanism is often not trivial [30]. For transparency it must be ensured that a task's references to memory, other nodes and external devices *and vice versa* are updated correctly. This may greatly increase the size of the context and the migration time when tasks can be migrated at any point during execution. Moreover, it might result in unpredictable behaviour which makes it impossible to determine an upper bound on this time.

A viable solution to this problem are migration mechanisms based on *code checkpointing*. In this approach migration points are inserted in the tasks which are executed at regular intervals [9]. These points can for example be inserted in the main control loop of the task, see Figure 4.3. Every time such a point is reached it is checked if a migration was triggered, if this is the case

a migration is initiated. At a migration point all state information (context) that is needed to resume task execution at the destination node is explicitly specified. This is generally much less than the complete context. Migration points can be inserted in the original code by hand or by the compiler [27]. The state information required for a migration must be identified at these points, which may not be trivial. Thus code modifications are required for this strategy which lead to overhead in the execution time, the size of which depends on the frequency of the checkpoints.

With code-checkpointing there is a trade-off between the design-time and run-time overhead, response time and context size. An approach which reduces the run-time overhead is dynamic code modification [45]. The code that performs a migration is then only inserted at a checkpoint if the migration is actually triggered. Yet another approach based on hardware supported migration is presented in [34], where a number of registers originally intended for debugging are used for migrating tasks. An interrupt is raised when the program counter reaches a value in one of those registers, they are however only activated by the OS when a migration is triggered. When this happens *and* the set value of the program counter is reached an interrupt is generated and the migration is performed in the interrupt service routine. Thus there is no overhead during execution and the OS effort is minimized. This approach is obviously not portable but might be implemented in a similar way with other interrupt mechanisms.

In order to evaluate migration strategies we must evaluate the cost, which consists of the following three main factors:

- the increase in design effort;
- the run-time overhead when no migration is performed;
- the overhead of an actual migration.

Furthermore the transparency and re-usability of the migration mechanism should be traded off with design effort and overhead. When migration is applied at a high level such as in the user application, the implementation is usually less complex than an implementation at a low level such as the OS-kernel. Performance, transparency and re-usability are however better for implementations at a low level [35].

For each separate case it should be determined whether the cost of task migration is justified. For instance, the effect of load balancing can be evaluated by plotting the wall clock time against the task's CPU time for both an unbalanced situation and a scenario where tasks migrate [9]. A break-even point where the cost of task migration is compensated by the increase in

performance can then be determined. Every change in the system however leads to a different situation where those trade-offs must be re-evaluated.

4.2.3 Implementations

An assessment of task migration on embedded real-time systems is presented in [1]. Tasks are allocated during run-time to maximize performance and optimize energy consumption. The cores have both a private memory and access to a shared memory. All components are interconnected by a shared bus. The migration strategy is based on code checkpointing which is implemented at the OS and middleware level. The software under test consists of a streaming multimedia application with soft real-time constraints and requirements that vary for different scenarios. This paper evaluates the long-term improvements enabled by migration versus the inherent short-term overhead. Experiments show that the timing overhead of the checkpoints is under 1% when no migrations are triggered, so performance is hardly affected when the system is in steady-state. Furthermore it was found the impact of migration on the software performance can be hidden with properly designed communication buffers. The overall test setup is sophisticated, and many aspects such as the changing software requirements support the assumptions that we made in Chapters 2 and 3 of this report. A major difference is that soft real-time applications are considered, so there is no partitioning and a deep analysis of inter-task communication. Furthermore the shared bus used for on-chip communication is not a scalable solution fit for future multi-cores.

The approach in [6] also considers a hybrid memory architecture with local scratchpads in addition to a global shared memory located at the centre of the chip. The nodes communicate over a Network-on-Chip using message passing. Communication energy is then minimized by deciding whether to obtain the code from the source node or from shared memory at run-time. This hybrid approach results in lower energy consumption and, in some cases, reduced task migration times because of the decreased distance. Although large on-chip memories seem somewhat unrealistic, it is interesting to see that a proper trade-off can lead to savings in traffic volume and migration times of up to 24% and 29% respectively.

In [14] the impact of task migration on NoC-based multi-cores is evaluated for soft real-time applications. The workload is dynamic and besides the main goal of meeting real-time constraints, the consumption of energy is minimized. Migration is based on a copy model where the full context is migrated along with the code and data, triggered by an interrupt. The results show that despite the overhead, task migration over a NoC pays off in terms of overall system performance and energy savings.

The research presented in [41] identifies the need for task migration to satisfy performance requirements. At the same time migration poses a significant challenge to timing predictability because of cache warm-up and an increase in NoC traffic. A scheme that enables real-time tasks to meet their deadlines in presence of task migration based on *push-assisted migration* is presented. Each core is assumed to have an L2-cache in which a task fits completely. The novelty of the migration mechanism lies in that cache lines are copied from the source to the destination node *prior* to the invocation of the task on the destination, thus not affecting the execution time. This is opposite to regular cache behaviour where data are pulled into the cache one at a time upon a cache miss. The effectiveness of this scheme is influenced by the complexity of the algorithm, the slack time that is available for copying, and whether the cache is pushed partly or as a whole. The results show that the migration overhead can be bounded to less than 33% of a tasks execution time, which makes software-assisted cache migration a feasible solution in this particular scenario.

4.3 Deterministic Task Migration

In Chapter 2 we identified the possibly unpredictable behaviour of the interconnect as a major challenge. So far the concepts presented in this chapter did not address the challenge of *deterministic* task migration. This can be achieved when the non-determinism caused by the sharing of resources is eliminated. In this section we start with an analysis of the possible causes for such undeterministic, and present an overview of the requirements for deterministic task migration. We then investigate how these requirements translate to modern NoC-based multi-cores.

4.3.1 Requirements

A migration is always triggered by the mapping process which signals the source core to start. The task is then physically and logically migrated as described in section 4.2. The destination core must signal the mapping process upon completion. These events correspond to the basic migration steps 1, 4 and 7 which are depicted in Figure 4.2. We deduce a number of requirements that are necessary for deterministic behaviour:

1. the mapping process and the control traffic must be predictable;
2. the time required for the task transfer must be bounded;
3. other software components may not malfunction during the migration;
4. the restarted task must have exactly the same WCET as it had on the source;

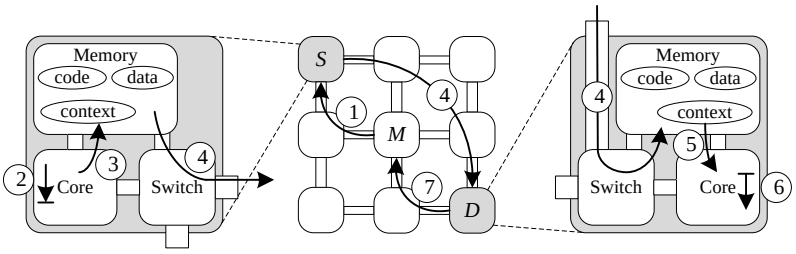

Figure 4.4: Detailed view of the seven basic migration steps on a processor with nine cores.

5. other software components may not be affected after the migration is complete.

These requirements are not trivial and each imply a set of measures that affect different layers of a system. We will now describe each requirement in more detail and discuss potential solutions.

Predictable behaviour of the mapping process can be ensured with the mode-based mapping approach that we proposed in Chapter 3. For the control traffic to be deterministic the more fundamental problem of dealing with the shared interconnect must be addressed, we will look into this in detail later this section.

To understand how the time required for the physical migration can be bounded, consider the basic migration steps 2 to 6 again which are depicted in more detail in Figure 4.4. The time required for step two depends on the migration strategy, when code checkpointing is used this is the time to reach the next checkpoint. If a task can be migrated at any point, this time will be zero. In step three the contents of the registers and possibly other modified data are written to a context datastructure in memory. It should be no problem to derive an upper bound on the time required for this if the context is clearly specified. If we are migrating a task that is not currently running, step two and three can be omitted. Step four poses a great challenge because a potentially large amount of data must be copied over the interconnect in a bounded amount of time. This is only possible when Quality-of-Service can be guaranteed, several NoC concepts that can do this are be discussed later this section. In step five the context is restored from memory so that the task can be restarted in step six. The timing behaviour of those steps should again be quite trivial. In conclusion, the main difficulty in meeting this requirement seems to be in the deterministic transportation of context, code and data over the interconnect.

It is inevitable that other components within a system are affected by a

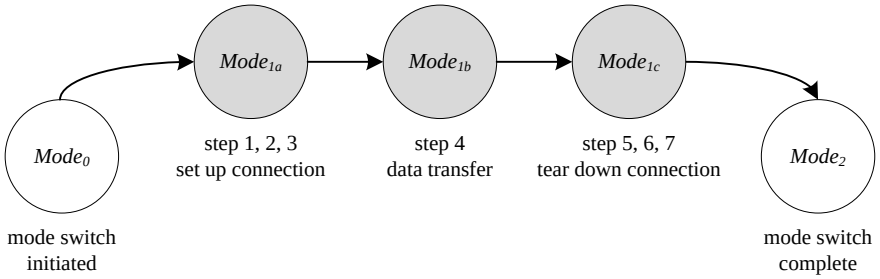

Figure 4.5: A mode switch from $Mode_0$ to $Mode_2$ via three transient modes denoted as $Mode_{1a...c}$ in which a connection is set up, a task is transferred, and the connection is torn down.

migration. To avoid unacceptable variations in the execution time of other tasks, the consequences of a migration must be clearly defined and accounted for in the WCET analysis. To do this we propose to define a number of *transient modes* that are placed between two actual modes. At least three of those are needed to complete the physical migration:

a. stop the task and store the context on the source, prepare the destination for receiving the task, and set up the interconnect to provide a connection between the two;

b. send the data over the connection;

c. destroy anything that is left on the source, restore and restart the task on the destination, and tear down the connection between the two.

A mode switch from $Mode_0$ to $Mode_2$ via the transient modes captured in $Mode_{1a...c}$ is depicted in Figure 4.5. Not only the basic migration steps fit into those transient modes, also the usage of a connection consists of three phases . A connection between source and destination must be set up before it can be used, and torn down after the migration is complete. During the transient nodes the same level of service must still be provided to the tasks that are not affected by the mode switch.

For a migrated task to have the same WCET before and after the migration there are several sub-requirements. In homogeneous architectures the timing of instructions that access registers or local memory will be the same on any two cores. A significant contribution to the WCET however comes from shared memory accesses, inter-core communication, and the access of external resources. The timing of those is dependent on the location and *does* vary between two cores. This leads to the following requirements:

a. all memory accesses that were local on the source must also be local on the destination core;

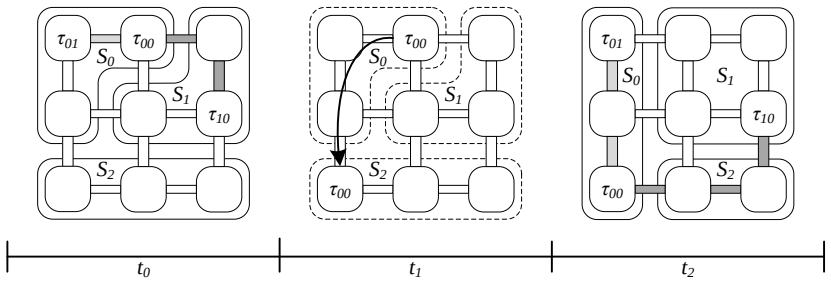

Figure 4.6: Task τ_{00} must communicate with τ_{01} and τ_{10} via different paths after it is migrated.

b. the upper bound on the timing of communication operations with other cores and external resources must be the same.

This first requirement can be satisfied when the complete dataset is copied to the destination core. The second requirement means that the derivation of the worst case time of a communication operation must always consider the maximum distance that two nodes can ever be apart from each other. An example involving a NoC is shown in Figure 4.6, where task τ_{00} communicates with task τ_{01} which is in the same partition (lighter shaded link) and task τ_{10} in partition $S1$ (darker shaded links). After task τ_{00} is migrated to the bottom left, the traffic must be re-routed. In this case the distance to both other nodes is increased. The same QoS must be guaranteed on the communication before and after, which usually means that the network is configured differently before, during and after the migration.

Not only the migrated task but also all other tasks that it communicates with must have the same execution time as before the migration. Therefore all the references to the migrated task must be updated in those tasks. This means the other tasks must somehow learn that a task has migrated and on which core it now runs. With mode-based mapping it suffices to notify all cores of the switch, after which they can individually update their internal references. This will certainly affect the WCET of those tasks, and therefore again the derivation of the worst case time of communication operations must be based on the largest possible distance between two nodes. Partitioning enforces isolation between groups of tasks and the use of interfaces for inter-partition communication, which abstracts from the internal configuration of a partition. Thus partitioning is a powerful mechanism for protecting tasks from events in other partitions such as migrations.

We can now give a complete overview of all the requirements and potential solutions presented so far:

1. The mapping process and the control traffic must be predictable.

Solution: Mode-based mapping and guarantees on QoS.

2. The time required for the task transfer must be bounded.

   Solution: Ensure an upper time bound on the halting and restarting of tasks and have guarantees on QoS.

3. Other software components may not malfunction during the migration.

   Solution: Define transient modes that capture the details of the migration.

4. The restarted task must have exactly the same WCET as it had on the source.

   (a) All memory accesses that were local on the source must also be local on the destination core.

   Solution: Copy the complete dataset.

   (b) The upper bound on the timing of communication operations with other cores and external resources must be the same.

   Solution: Always consider the maximum possible distance between two nodes and maintain the level of QoS.

5. Other software components may not be affected after the migration is complete.

   Solution: Update the references and always consider the maximum possible distance between two nodes.

With these requirements we have defined the prerequisites for deterministic task migration and partly addressed the second objective. From the presented requirements it however becomes clear that the main challenge of deterministic task migration is to *transfer* tasks deterministically over a Network-on-Chip, which requires guarantees on the QoS. The mapping of traffic that we presented in Chapter 3 offers a high level solution to concurrent access of an interconnect. A further challenge lies in applying this approach in modern NoCs, which we consider the main new challenge in achieving deterministic task migration. We therefore we on this subject and define the third and last objective of this thesis:
How to transfer tasks over a Network-on-Chip deterministically.

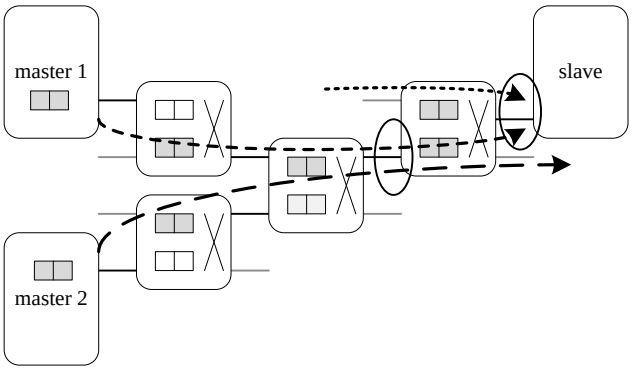

Figure 4.7: Two cases of contention: two traffic streams want to use the same link (left circle) and two streams converge on a single slave (right circle).

In the remaining of this chapter we will look into QoS in NoCs in more detail, after which we present a number of experiments to verify these concepts in Chapter 5.

4.3.2 Deterministic Communication

We can only determine an upper bound on the time needed for a communication operation if the behaviour of the interconnect is deterministic. This means we must avoid the unpredictability coming from the random access to shared resources, which can be achieved by separating the access attempts in either time or space. The master-slave interconnect model presented in Chapter 2 showed that sequential traffic is isolated in time, and that spatial isolation can be achieved by sending traffic via different paths. Unpredictable behaviour may occur if two nodes request a channel at the same time, or when two messages that were sent via different paths arrive at a destination at the same time.

Networks-on-Chips are complex switched architectures in which both these situations can appear because traffic is split, pipelined and concurrent. A single link handles packets sequentially, but there are multiple end-to-end paths that transport packets in parallel. Concurrent access of shared resources occurs when multiple packets want to use the same link. Another problem appears when multiple traffic streams converge on a single slave. Both situations can lead to contention and potentially to congestion, which is depicted in Figure 4.7 indicated by two circles. Therefore the available capacity of both the interconnect and the slaves must be partitioned in time and space. To serve varying user demands and still be able to guarantee network access and QoS to the nodes, capacity must be *reserved* at design time [20].

Between the different modes the NoC must be reconfigured. This may include reprogramming the routers and the notification of end users. The mapping process closely cooperates with the mechanism that reconfigures the interconnect, both are usually centralized and executed on one of the cores [1, 20]. Such an approach is not scalable when many cores are placed on a single chip because such a central control mechanism will become a traffic bottleneck, but it suffices for current multi-cores.

When a mode switch occurs the mapping of both the tasks and the connections is changed. Connections can be added, removed and modified. Programmable networks offer this flexibility as described in [23], which gives three requirements for NoC reconfiguration:

- the architecture must be programmable;
- the configurations must be specified and computed either online or at design time;
- facilities are needed for controlling the change and leaving the NoC in a consistent state after modification.

This last point is crucial: when moving from one mode to another the system is momentarily in a transient state in which the service to tasks that are still running needs to be continued to avoid deadlocks and corrupted data. Therefore we propose to split a mode switch up in different transient modes. The control traffic must again have guarantees on QoS.

When multiple partitions are mapped on one processor there must be strict isolation in order to meet certification requirements. This is because a malfunctioning partition might try to access all resources it is connected to (the "babbling idiot" syndrome). Thus the traffic of different partitions must be separated, preferably on a hardware level. To achieve traffic isolation not only the links and routers that transport packets for a particular partition must be reserved, also the other partitions must be blocked from accessing these resources. One approach to traffic isolation is to use separate physical networks. Although this greatly increases cost, it ensures complete isolation. The routers of the different networks can possibly be combined to reduce the total cost if packets are sufficiently separated internally.

4.3.3 Guarantees on Quality-of-Service

In Chapter 2 we explained end-to-end *flow control* for managing traffic streams and avoiding or bounding congestion, and *buffering strategies* which define the architecture of each router. Several combinations of these two techniques can offer guarantees on QoS in a NoC. The basic idea is that traffic streams from different nodes should not influence each other.

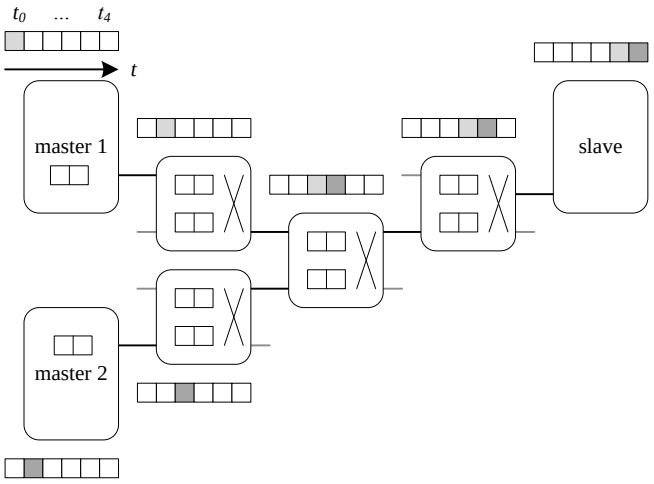

Figure 4.8: The concept of time-division-multiplexing, the time slots are depicted above each node and router.

An approach to such isolation with buffer-less flow control is *circuit switching*, where a circuit is set up that exclusively transports traffic between the two nodes at the end points. This requires the switches inside the routers must be non-blocking, which means multiple packets can be distributed to multiple outputs at the same time. In circuit switching there is no contention because all the intermediate links and buffers are reserved. Therefore this technique is a natural candidate for mapping traffic onto a NoC. The sharing of wires is however cancelled with this approach, which is one of the main advantages that NoCs offer. Also there is the overhead of setting up and tearing down connections, so frequent changes should be avoided. All in all this solution is suited best for long-lived connections that require a lot of bandwidth.

To overcome these disadvantages *multiplexing* can be applied. Frequency multiplexing is common in off-chip networks but is not suited for implementation in digital chips, except perhaps in optical NoCs. *Time-Division Multiplexing* (TDM) can however be applied to circuit switching, and has been proven to be able to provide guarantees on QoS [20]. It requires all the routers to be synchronized or in lock-step, an example is depicted in Figure 4.8. The first master sends a packet in time slot t_0, the second in t_1. The packets are shifted in time when they arrive at the second router stage and therefore do not collide. The advantages of TDM are that no buffers are needed except in the end points, that there is isolation between the flows, and that bandwidth and latency can easily be guaranteed. Disadvantages are that the mean latency is high and that admission control is difficult.

The formal model of the hardware that we presented in Chapter 3 could be extended to include TDM circuit switching, each connection will then be allowed to use a certain *percentage* of the link capacity. We believe that this is a promising solution for efficiently mapping traffic onto a NoC.

Circuit switching avoids contention altogether. Another approach is to bound contention by delaying one of the packets when two arrive at the same time at the same place. This requires *buffered* flow control. It is then possible to guarantee QoS if a non-blocking switch is used in combination with an output buffering scheme and Store-And-Forward (SAF) or Virtual Cut-Through (VCT) flow control. With output buffering there must be a buffer for each input-output combination which leads to a cost of N^2 if there are as many inputs as outputs. With SAF and VCT flow control an element of a packet (flit) is only forwarded to the next router if it has sufficient space in its buffers to store the complete packet, as opposed to e.g. wormhole routing. This means that when congestion appears, a packet will be stuck in one router and cannot be spread over multiple buffers which would potentially block multiple links. This combination of techniques can offer guarantees on QoS only if a feasible scheduling strategy is applied per link as well as for a series of links (end-to-end). Resources are not reserved in this approach which makes it impossible to map traffic, but if it can be guaranteed that contention is bound it might be possible to achieve deterministic communication anyway.

Another approach to guarantee QoS is the application of a non-blocking switch in combination with *virtual circuit buffering*. In such a scheme a buffer is reserved for each circuit (traffic stream) in each router on the path. While this is the most expensive buffering solution, QoS can be guaranteed per flow and traffic is isolated from other streams. Note that congestion might still occur if the slave is too slow in consuming packets. Therefore again a feasible end-to-end scheduling strategy is needed. This approach is again a natural candidate for mapping traffic onto a NoC, the disadvantages of circuit switching are avoided.

An end-to-end scheduling scheme suitable for these two buffered flow-control schemes is *aggregate resource allocation*, which regulates the traffic flows at the edges of the network. Thus the regularity of flows and the maximum latency and buffer requirements can be controlled. However, flows can still become bursty inside the network due to contention. *Rate-controlled* scheduling is a more advanced strategy that regulates the flows at the edges as well as inside the network so that the latency bound is lowered. This can be combined with virtual circuit buffers in order to have isolation between the flows [10]. End-to-end scheduling should include admission control, which is the allocation of NoC capacity for a new flow without compromising on service

to existing flows. Admission control fits well to the mode-based mapping that we proposed because network capacity is explicitly reserved in each mode.

4.4 Summary

In Chapter 3 we found that load balancing can be achieved by mode-based mapping, which requires task migration. At the beginning of this chapter we presented several additional motivations for task migration: traffic reduction by exploiting data locality, management of power and temperature, and flexible redundancy. After that we gave an overview of related work. Starting with the basics we divided a migration into seven steps, and presented several migration strategies and relevant implementations.

To address the determinism that is required in real-time systems, we deduced a number of requirements for deterministic task migration and discussed potential solutions for each of those. It became clear that the mode-based mapping of traffic onto the interconnect sufficiently addresses the challenge of accessing a shared interconnect on a high level. We proposed the use of *transient modes* to deal with the complexity of changing the mapping of the interconnect between different modes. Our analysis of the requirements shows that the deterministic *transfer* of tasks over the interconnect is the main new challenge of task migration in multi-cores. In Chapter 2 we already mentioned that such transfers require guarantees on QoS. To address this subject we introduced the third objective: to find out how tasks can be deterministically *transferred* over a NoC.

To address this objective we analyzed Networks-on-Chip with help of the interconnect model from Chapter 2. We found that contention occurs when multiple packets request access to a shared resource such as a link, buffer or destination node. There are several solutions to this. First we presented circuit switching, which is quite suitable for the mode-based mapping of traffic onto a NoC. Multiplexing techniques such as TDM can be applied to increase the usage of hardware resources. Some buffered flow control techniques (e.g. Store-And-Forward, Virtual-Cut-Through) can also offer guarantees on QoS, but seem difficult to use in combination with resource reservation. Virtual circuit buffering on the other hand is again a natural candidate because buffers are reserved for traffic flows in this approach.

Chapter 5

Task Transfer Experiments

In the first three chapters we analysed the complexity of deploying safety-critical real-time software on modern multi-core processors. In Chapter 4 we focussed on one specific aspect in which multiple challenges are combined: the deterministic transfer of tasks over a Network-on-Chip (third objective). In this chapter we describe the experiments that we conducted to verify the concepts presented in this thesis. The goal of the experiments is to evaluate task migration over a NoC using different *transfer methods*. We need a basic task migration mechanism for this, which we introduce in Section 5.1. Then we present the hardware and the setup of the experiment, and describe the migration mechanism in detail. We conclude with an assessment of the validity of the experiments and identify the limitations of our approach.

5.1 Approach

In this section we present the migration strategy that we use in the experiments. We discuss the different data transfer methods in detail. Furthermore we describe the software framework that implements the migration strategy.

5.1.1 Migration Model

In Chapter 4 we stated that task migration can be implemented on different levels such as the application or OS-kernel level. For the experiments we implemented a simple migration mechanism on the application level. There are two reasons for this:

- the complexity of such an implementation is limited because it does not require modification of an OS or hypervisor;
- the implementation is portable.

Even more, we use an OS for setting up the system but not for performing the experiments. As a result the size of the context is limited because there are no OS-related datastructures.

Furthermore we only consider *periodic* tasks that are executed at least once during a hyperperiod. Because of the real-time character of the tasks we do not consider the migration of tasks while running feasible, and therefore only allow tasks to migrate when they are *stopped*. This avoids saving and restoring context altogether and reduces the complexity of the migration mechanism. For the transfer of a task over an interconnect the division between data, code and context is merely an administrative issue. Thus we consider it reasonable to omit the context without affecting the validity of the results. We realize that many aspects of task migration are not covered by our experiments, we explicitly focus on task *transfers* only.

The code for the experiments was implemented in the C language (language level gnu99). In that language, a natural candidate to represent tasks are *functions* (the term has a different meaning here than in Chapter 2). On the assembly level, a function is a section of program code. If a function is called, the program counter is set to the functions starting address and the CPU starts executing that block of instructions. The very last instruction of a function is usually a jump back to the address from which the function was called plus one. In the C language a function may be called with a number of arguments. Those can contain or refer to a larger dataset, alternatively data may be declared "global" so that they are accessible by every function. Upon completion a function might have overwritten such global data but can also explicitly return data to the calling function. Thus a function has input and output, and can also work on shared data. This suffices to implement task migration in our experiments.

Task management requires a mechanism to start functions, which must also take control when a function has finished. Information about the starting address of a function can be communicated conveniently by using *function pointers*. Because no OS was used, we implemented an administrative task that accepts pointers, starts the corresponding functions and takes control again when the function finishes execution.

5.1.2 Transfer Methods

Because our tasks have no context they only consist of a block of *code memory* and a *dataset*. We assume that both the code and private dataset completely fit and remain in the cores local memory. We believe this is a reasonable assumption for the relatively small control tasks which are common in embedded systems. Even in more data intensive tasks, such as for

instance radar image processing in the avionics domain, it is plausible that the code can be kept in the local memory while fresh data is obtained periodically via a reserved connection. When a migration is triggered the code memory and dataset must be transferred from the source to the destination core. There are two fundamental approaches to data transfer, one of which can be subdivided based on the memory architecture [25]:

- shared memory supported by:
  - a private cache at each core;
  - a shared cache architecture;
  - a scratchpad memory at each core;
- message passing.

We will now go deeper into each method.

If private caches are used, cores are forced to get all data from a shared (usually off-chip) main memory. In a shared cache architecture a core can obtain data from another core's local cache. When scratchpad memories are used it is possible to construct the address space in such a way that a programmer can copy data from another core's memory. We will focus only on core-to-core migrations, for which an architecture with private caches is not suitable. In future multi-core processors with hundreds or thousands of cores the cost of accessing off-chip memory will raise tremendously [12]. With task migration data locality can be optimally exploited in order to reduce traffic.

With a shared memory architecture the transfer of a task is initiated by executing the program code. Each instruction that is executed and every data element that is accessed will be pulled to the core by the cache subsystem. This is a sunny scenario from the programmers standpoint because it requires no facilities in software at all. From the previous chapters however it has become clear that random access of shared memory leads to unpredictability. Especially distributed shared cache architectures either lead to overly negative WCETs or cannot be analyzed at all [50]. A main cause for this unpredictability is that the execution time of the task is affected by cache operations until eventually all code and data is accessed. This effect is known as the *cache-warmup*. The complexity further increases when a new task is started on the source core which starts to evict cache lines. All in all the regular cache-pull method is not suited for safety-critical real-time systems.

An approach that proposes a solution to this disadvantage of shared caches is presented in [41], namely *push-assisted* migration. Here a task is not

started directly, but first the code and data are *prefetched* so that they are present in the local cache when the task is started. This impacts the software because these prefetch operations must be included. Now the WCET analysis of the task can be based on local data accesses only. The transfer time is completely separated from execution time an must be accounted for in the mapping and scheduling scheme. All memory operations now appear in one block so it is easier to model the traffic and map it onto the interconnect. To determine an upper bound on the migration time, the data transfer operations must be deterministic. This should be possible if it can be guaranteed that the cache system can get the data from the source core before they are overwritten.

Another approach to shared memory is to provide each core with a local scratchpad memory, into which data must be explicitly copied. Some caches can be "locked" so that they act as scratchpad. Transferring tasks is then similar to prefetching, but now the source of the data must be addressed explicitly. Because copy operations involve the CPU this approach might be slower. Timing analysis is simpler because the memory architecture is less complex. In combination with mode-based mapping it is possible to achieve deterministic behaviour if data operations are scheduled in time slots. This approach results in more software design effort because the code and data must explicitly be copied from the source core.

The last approach is independent from the type of local memory as data is transferred via *message passing* rather than through the shared memory. The source core must packetize the code and data and send those through the interconnect, after which the destination core receives and stores them locally. This can be combined with regulated access to the interconnect because source and destination can negotiate about their communication. A disadvantage is that both cores are busy during the transfer of the task, although they might do useful work while waiting for the interconnect. Sending and receiving data must be explicitly done in the program code, so there is some overhead in this approach. In conclusion, there are three approaches have the potential to offer deterministic task migration:

1. push-assisted migration in shared caches;
2. explicit copy from and to local scratchpad memories;
3. message passing.

5.1.3 Framework

To perform the experiments we designed a software structure that we refer to as the *framework*. It allows us to execute tasks statically at cores (e.g.

task under test or time measurement). We furthermore use it to control the experiments, that is, the task transfers. The framework allows us to evaluate the impact of three different parameters:

- the transfer method;
- the distance between source and destination;
- the size of the code and dataset.

The main challenge we faced while developing the framework is to perform reliable measurements while still offering flexibility. The quantity we measure is *time*, therefore any interference with the experiments must be avoided. For instance, interrupts must be disabled on all cores.

From the previous chapters it has become clear that simultaneous access of shared resources influences the timing behaviour of the tasks. We estimated that the development a complete partitioning scheme for the experiments is too costly and time consuming. Therefore we considered it sufficient for these experiments to make sure that the framework does not interfere with the measurements. We especially avoid unnecessary access of the interconnect by framework tasks during the experiments.

The framework sets up the network and memory on each core. After that the core starts its dedicated function, one core is the designated *mapping* core. The mapping function instructs the other cores when to start their experiments, and triggers the migrations. The remaining cores are split in *worker* cores and an equal amount of *measurement* cores. Each worker core is next to a measurement core so they form pairs that can communicate over just one link. This minimizes the network traffic required for the measurements. The worker cores wait until the mapping core initiates a test sequence. In each test sequence a task is migrated from the mapping core to that worker core multiple times.

5.2 Experimental Setup

In this section we introduce the multi-core processor that we used for the experiments. The implementation details of the software are discussed next, which are closely related to the hardware architecture. Finally we describe how we implemented the transfer methods from Section 5.1 on our test platform.

5.2.1 Hardware Architecture

For the experiments we used a TILE*Pro*64™processor from the Tilera® corporation, to which we will refer as the *test platform*. It has 64 identical

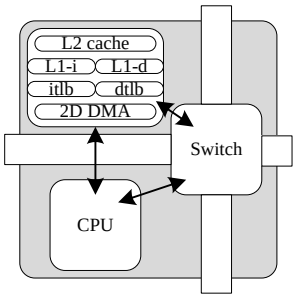

Figure 5.1: The architecture of a tile.

cores and is very well suited for embedded systems because of its power efficiency [47]. The cores are interconnected by multiple NoCs with a two-dimensional mesh architecture that are used for both inter-core communication and the access to memory and external resources. Because to date NoCs are the only type of interconnect that have been proven to be scalable, we consider this processor very suitable for the study of the challenges that are faced with the introduction of multi-cores [24]. Furthermore the it supports both shared memory and message passing, which allows to evaluate fundamentally different transfer methods.

The cores have a 32-bit Very Large Instruction Word (VLIW) architecture that can bundle up to three instructions per cycle. There are three pipeline stages for integer and fixed point operations, there is no dedicated floating point unit. Each CPU is placed on a "tile" that also contains a local cache system and a switch that offers access to the network. The architecture of a tile is depicted in Figure 5.1.

There are four memory controllers that provide access to off-chip DDR2 type memory. Each core has a local cache architecture consisting of:

- an L2 cache for both instructions and data;
- an L1-instruction cache;
- an L1-data cache;
- a separate Translation Look-aside Buffer (TLB) for both instructions and data.

More details about the caches are can be found in Table 5.1. A hardware mechanism combines the aggregate of all on-chip L2 caches into a shared L3 cache named the "Dynamic Distributed Cache^{TM}". Based on a directory protocol, this cache provides a coherent shared-memory view . A memory page can be assigned a home tile where it will be cached upon access, even

	Size	Associativity	Line size	Allocate policy	Write policy
L1-I	16 kB	direct mapped	64 B	On read miss	N/A
L1-D	8 kB	2-way	16 B	On load miss	Write through
L2	64 kB	4-way	64 B	On load/store miss	Writeback

Table 5.1: The properties of the cache architecture.

if the access comes from another core. The caches of our test platform can be configured in different ways: as shared cache, as private caches or they can be locked so that they behave as scratchpad memories.

The Network-on-Chip actually consists of six two-dimensional 8x8 mesh networks. Details about the network are described in [49]. Each network is intended for a different purpose:

- the User Dynamic Network (UDN) can be directly accessed by the programmer;
- I/O devices can be accessed via the I/O Dynamic Network (IDN);
- the Memory Dynamic Network (MDN) is for memory data transfers in general;
- the Coherence Dynamic Network (CDN) is for cache-coherence invalidate messages;
- the Tile Dynamic Network (TDN) is for memory data transfer between tiles;
- the Static Network (STN) is a programmable low-latency network optimized for streaming.

Each switch has five connections with three-entry input buffers and an all-to-all (non-blocking) crossbar switch. The routing strategy in the dynamic networks is dimension-ordered wormhole routing. Packets are first sent in the x dimension, and then in the y dimension once the destination column is reached. Wormhole routing means that a packet is split in a number of *flits*, which are forwarded immediately by the receiving router as soon as the next router has space for a flit. This strategy offers low latency (the transmission time of one flit) and has low buffering requirements (a minimum of one flit per link), but a blocked packet can occupy multiple switches which potentially increases congestion. Between the switch and the CPU there is additional buffering for end-to-end flow control. In the static network the routing decisions at each router are programmable which allows to set up communication channels in a circuit-switched manner.

There are three approaches to inter-core communication:

- shared memory (using the MDN, CDN and TDN);
- message passing using the UDN;
- message passing using the STN.

All those networks have link-level credit-based flow control. The approach to high-level flow control however is quite different in each network [49]. A high level flow-control technique based on buffer-preallocation in the memory controllers makes sure that the memory networks are free of deadlocks and congestion. Because buffers are physically limited the amount of data requests of each node must be bounded for guarantees on the latency, which is essentially resource reservation. To determine that amount an analysis of the complete memory architecture including the networks is necessary. Such a complex process is out of the scope of this thesis.

In the UDN there is flow control on the link level only so deadlocks due to insufficient buffering capacity are possible. There is a mechanism for deadlock recovery. Therefore higher level flow control must be implemented in software. Because the programmer has direct access to this network we consider it possible to implement circuit switching in software, without hardware support for setting up circuits (connections). Because the switches of the dynamic network cannot be programmed, each core must maintain a table of all the connections. This solution is therefore not trivial and has overhead in software. If the cores can be sufficiently synchronized however, virtual TDM circuit switching could be implemented which has significant advantages .

The STN implements circuit switching in hardware, so guarantees on QoS can be provided once a connection is established. Each link can however only be used by one connection, which usually means the network usage is far from optimal. Adding and removing connections involves the reprogramming of switches. Circuit-switching is therefore a suitable strategy for long-lived connections that require a lot of bandwidth. It is not fit for short lived connections because of the overhead, and also not for connections that require little bandwidth because too much capacity reserved. This last case is commonly found in measurement and control functions which can typically be found in embedded systems. For such connections TDM circuit switching is a much better solution because links can be shared between connections.

There is a interesting trade-off between regular circuit switching and the virtual TMD circuit switching that we suggested. Let us assume that a block of packets must be transferred periodically, but that it is too costly to maintain one long-lived connection. If the period is short and the block

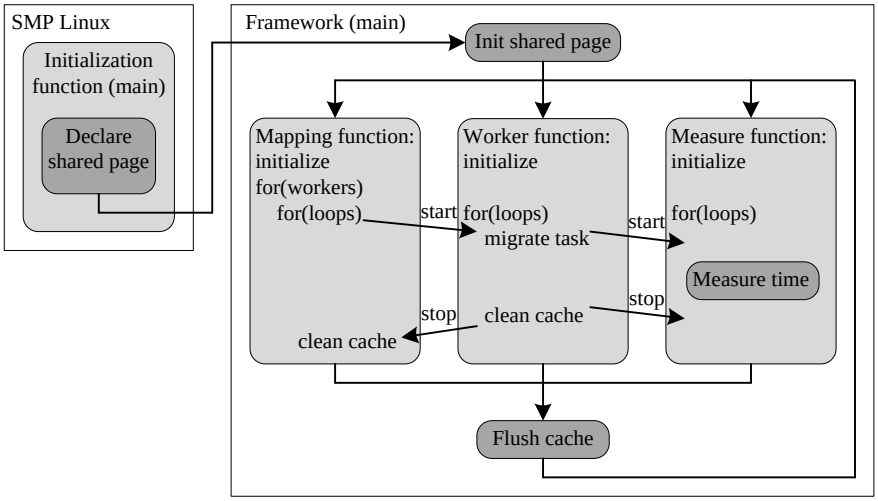

Figure 5.2: An overview of the functions in the framework.

size small, the overhead of simulated TDM circuit switching could be outweighed by the flexibility it offers. If the period and/or the block size grow there will be a point at which the overhead of hardware-supported circuit switching becomes smaller. Every connections could be assigned to one of these techniques so that their advantages are combined. We will not go into the details of such a software implementation of TDM here, but it is certainly an interesting subject for future research.

5.2.2 Software Architecture

We presented a high-level view of the framework in Section 5.1, here we will further go into the details. A symmetrical multiprocessing version of the Linux OS is available for our test platform. It offers the possibility to declare "dataplane" tiles, which continuously execute a single thread without interference by the scheduler. We evaluated this mode because it is suited for high performance and time-critical applications. However, we found that the access to hardware is too restricted in this mode to conduct the proposed experiments. Therefore we implemented the framework on "bare metal", which means there is no OS or hypervisor. This significantly increased the design effort. In the experimental setup the Linux OS still runs on eight of the 64 tiles to perform basic hardware configuration. After startup it initializes the NoC and one huge page (16 MB) of shared memory, after which it informs the framework on each tile about their dedicated function and the shared page. During the experiments the OS is idle.

An overview of the software architecture at the function level is depicted in Figure 5.2. The body of the framework is executed in a loop to perform

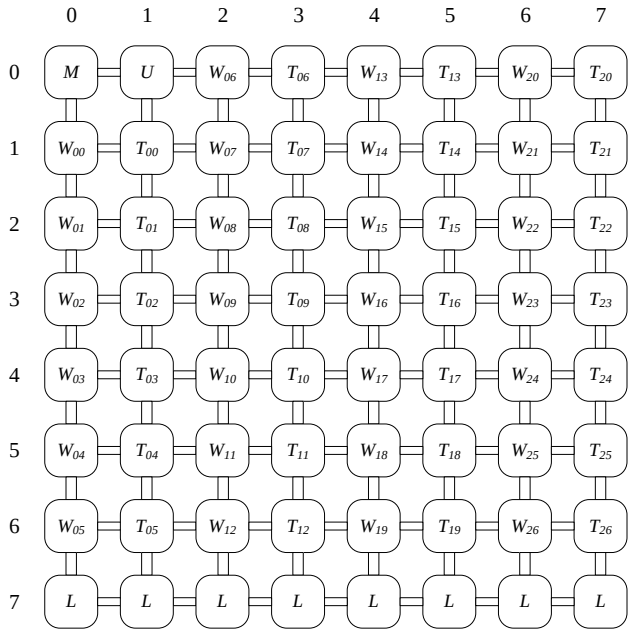

Figure 5.3: The mapping of tasks onto the cores(M = mapping core, U = unused, W = worker core, T = measurement core, L = Linux core).

measurements with varying code and dataset sizes. Between every iteration the L2 and L1 caches are flushed completely so that measurements cannot influence each other. Between each iteration within a function the code and data of the task are also removed from the cache. The mapping function controls the worker functions by sending control messages over the UDN. Every worker tile repeats the measurements for *loops* times after which the mapping core signals the next worker. These messages are sent just before and after the actual experiments to ensure that those are not influenced. The workers themselves signal their dedicated measurement core with control messages. The actual sequence of control messages is more complex than depicted in Figure 5.2 to ensure a synchronous start and detect erroneous messages.

Of the 64 cores, eight are used by the OS and one is dedicated to the mapping function. The remaining 55 cores can be used by the worker and measurement functions. Because these operate in pairs, one core is not used in this setup. An overview of the mapping is depicted in Figure 5.3.

The task that is actually migrated consists of instructions and a dataset. We do not want to perform WCET analysis, therefore we used tasks with a constant execution time. These were implemented as a sequence of “nop” instructions, which has the additional advantage we can easily adjust the

size of the code. The size of the instructions and the dataset ranges from 1000 bytes to 8000 bytes, iterating with a step size of 1000 bytes. The upper value is bounded by the size of the L1-data cache (8 kB), because we work with the assumption that both instruction and data memory fit and remain in the L1 cache.

5.2.3 Implementation of Transfer Methods

In our experiments we focus on step four of the basic steps of a task migration as presented in Chapter 4, namely the transfer of data from one core's local memory to that of another. In Section 5.1 we found that three transfer methods are suitable for deterministic task migration, two of which assume a shared memory model. In the first method a transfer is performed by prefetch operations, while in the second data must be explicitly copied.

The first transfer method that we implemented is the regular cache-pull method. Note that this method is *not* suitable for deterministic task migration and we did not consider it in detail in Section 5.1, we implemented it just to compare it to the others. At the start of the experiment we load the task under test into the L2 cache of the mapping core. The mapping function then signals the first worker function which starts executing that task, thus forcing the cache subsystem to transfer the task while simultaneously executing it. This process is repeated in a loop, during which the time is measured. The body of the loop consists of the following code lines:

```
//Start the timer on the measurement tile
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_START);
//Run the task
(*shared_task_instr_ptr)();
//Stop the timer on the measurement tile:
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_STOP);
//Clean the L1-I and L2 cache:
tmc_mem_invalidate_icache(shared_task_instr_ptr, current_task_size);
tmc_mem_finv(shared_task_instr_ptr, current_task_size);
```

For the sake of the example the code sections in this chapter show only the transfer of code memory. The transfer of the dataset and all code lines that are non-crucial are omitted.

The second transfer method that we implemented is to *prefetch* the task prior to execution. The body of the loop is as follows:

```
//Start the timer on the measurement tile
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_START);
//Prefetch the task in the L2 cache
tmc_mem_prefetch(shared_task_data_ptr_64, current_task_size);
//Run the task
(*shared_task_instr_ptr)();
//Stop the timer on the measurement tile:
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_STOP);
//Clean the L1-I and L2 cache:
```

```
tmc_mem_invalidate_icache(shared_task_instr_ptr, current_task_size);
tmc_mem_finv(shared_task_instr_ptr, current_task_size);
```

The prefetch function only works with data memory, therefore the memory page on which the program code resides is loaded into the data TLB as well as in the instruction TLB. Thus it we can use a data pointer for prefetching: *shared_task_data_ptr_64* and *shared_task_instr_ptr* point to the same address. Because the page has the same virtual address in both TLBs we must clean the L2 cache only once. The prefetch function uses a pseudo instruction that loads a byte into the "zero" register, which is not an actual operation but instructs the hardware to get that byte into the cache.

Thirdly, we explicitly copied data into the local memory. Because the data and code sizes are smaller than the cache capacity, the task will never be evicted from the cache. While this is very much like the use of scratchpad memories, the actual transfer is still performed by the cache subsystem. This method is therefore a mixture of prefetching and the explicit copying of data. Still this method is useful to compare with the others, we will see why in Chapter 6. We copy data into the local memory with explicit copy operations to a dummy variable, as shown in the following code:

```
*(dummy_ptr_64) = 0;
//Start the timer on the measurement tile
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_START);
for(int k = 0; k < current_instr_count; k++){
        *(dummy_ptr_64) = *(shared_task_data_ptr_64 + k);
}
//Run the task
(*shared_task_instr_ptr)();
//Stop the timer on the measurement tile:
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_STOP);
//Clean the L1-I and L2 cache:
tmc_mem_invalidate_icache(shared_task_instr_ptr, current_task_size);
tmc_mem_finv(shared_task_instr_ptr, current_task_size);
```

Again a data pointer must be used to access the instruction memory. Although this method is somewhat similar to prefetching, here the CPU must however wait for the data to arrive and then execute an actual copy operation whereas a prefetch operations only triggers the cache hardware.

These three previous transfer methods assume a shared memory model. The remaining two implementations are based on the transfer of data via message passing, in this case via the UDN. There is now explicit communications between a worker core and the mapping core that is the source node. To minimize overhead the maximum packet size of 20 words is used for this strategy. The code is as follows:

```
//Prefetch the memory area to which the task will be written
tmc_mem_prefetch(shared_task_data_ptr_32, current_task_size);
//Start the timer on the measurement tile
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_START);
//Receive the task via the UDN
```

```
for(int k = 0; k < (current_instr_count/10); k++){
        *(shared_task_data_ptr_32) = udn1_receive();
        *(shared_task_data_ptr_32 + 1) = udn1_receive();
        ...
        *(shared_task_data_ptr_32 + 19) = udn1_receive();
        shared_task_data_ptr_32 += 20;
}
//Run the task
(*shared_task_instr_ptr)();
//Stop the timer on the measurement tile:
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_STOP);
//Clean the L1-I and L2 cache:
tmc_mem_invalidate_icache(shared_task_instr_ptr, current_task_size);
tmc_mem_finv(shared_task_data_ptr_32, current_task_size);
```

First we prefetch the memory range to which the task is written so that the measurement is not influenced by data operations that are not related to the migration. Then the mapping core and worker core synchronize, this code is omitted here. All data is received as 32-bit words and written to the locally cached memory.

The fifth and last method is also based on message passing but now via the STN:

```
//Prefetch the memory area to which the task will be written
tmc_mem_prefetch(shared_task_data_ptr_32, current_task_size);
//Start the timer on the measurement tile
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_START);
//Receive the task via the UDN
for(int k = 0; k < (current_instr_count/10); k++){
        *(shared_task_data_ptr_32) = stn_receive();
        *(shared_task_data_ptr_32 + 1) = stn_receive();
        ...
        *(shared_task_data_ptr_32 + 19) = stn_receive();
        shared_task_data_ptr_32 += 20;
}
//Run the task
(*shared_task_instr_ptr)();
//Stop the timer on the measurement tile:
tmc_udn_send_1(measure_header, UDN0_DEMUX_TAG, TIMER_STOP);
//Clean the L1-I and L2 cache:
tmc_mem_invalidate_icache(shared_task_instr_ptr, current_task_size);
tmc_mem_finv(shared_task_data_ptr_32, current_task_size);
```

This loop very similar to that of transferring data via the UDN. There is no function to send a packet of 20 words as there was with the UDN, now use a loop in which 20 packets are sent to allow a fair comparison. The framework takes care of configuring the network which comprises the programming of a number of registers on each core, this code is omitted here. Note that there is no central mechanism for reconfiguring the STN, so setting up and tearing down connections requires the cooperation of all the concerned tiles.

As mentioned, the instruction memory can only be read and written with data operations if the page is manually loaded in both the instruction and data TLB. Such "self modifying code" is not allowed to run under the Linux

OS, which is the main reason that the experiments were performed on bare metal. In Section 5.1 it was stated that the migration is implemented at the application level. This is still true, but it should be noticed that the framework performs services that would normally be done by an OS or hypervisor.

In conclusion, we implemented five transfer methods:

- shared memory supported by a coherent shared cache architecture with:
  - cache-pull;
  - prefetch;
  - explicit copy;
- message passing using the UDN;
- message passing using the STN.

Of those only the last four can potentially transfer tasks deterministically.

5.3 Limitations

In this section we relate the proposed experiments to the concepts and requirements that were presented in the previous chapters. Furthermore we discuss the limitations of our approach and their impact on the validity of the experiments.

In our experiments we focus on step 4 of the basic migration steps, which corresponds to our third objective: the deterministic *transfer* of tasks over the interconnect. We implemented the other steps but do not perform time measurements on those. In our test setup there is no operating system or hypervisor, and tasks can only be migrated when stopped. Therefore there is no context at all, and only data and code memory must be transferred. We believe this is a feasible scenario for embedded tasks that are executed strictly periodically. The objective of the experiments is to investigate which transfer method can offer the required guarantees on QoS.

We evaluate five different transfer methods on our test platform. In Section 5.2 we concluded that use of the shared memory requires a comprehensive analysis of the memory architecture, including the NoC. The UDN uses the same routing and buffering strategies as the memory networks, but here additional end-to-end flow control like circuit switching could be implemented in software. We estimated however that the implementation effort is too high to include such a strategy in this thesis.

The tasks that we use in the experiments are not a realistic representation of embedded real-time software. Firstly, the code does not do anything useful because it consists of "nop" instructions. Secondly, the task sizes were picked so that the tasks fit in the L1-cache, and are not based on an analysis of real software. We expect however that on-chip memories will continue to grow in size and that multiple embedded tasks will fit the code of in one local memory. Furthermore we believe that our test platform is a suitable representation of future multi-core processors because of its low power usage and of the NoC that can be used in different ways. The experiments focus on the NoC, which is the fundamental new element in multi-cores.

We did not implement temporal and spatial partitioning, but perform the experiments while no other tasks are active on the processor. While this shows if a particular transfer method has the potential to offer guarantees on QoS and allows to compare the methods, it does not show the isolation of traffic. In other words, our experiments do not show the *limitations* of a transfer method with respect to interference from other traffic.

Traffic can be separated on different levels. In the mode-based mapping approach that we presented in Chapter 3 the traffic is separated by design. This is also the case in our experiments, so our results are valid in that respect. Such a strategy does however not provide protection from faults (e.g. "babbling idiots"), which can only be achieved with hardware supported *isolation* of traffic on a lower level. Our test platform features hardware support for isolating traffic on the UDN and STN, the so-called *hardwall*. This hardwall could be used to implement circuit switching on the UDN.

5.4 Summary

In this chapter we described our approach to the experiments whose objective is to investigate which transfer methods are suitable for deterministic task migration. We implemented a basic migration mechanism, our tasks consist of C-functions that are periodically executed and can migrate between executions. Because there is no context our migration model is very simple and focuses on the actual transfer of code and data memory. We analyzed different transfer methods and concluded that there are three approaches which can potentially offer deterministic task migration:

1. push-assisted migration in shared caches;
2. explicit data transfer using local scratchpad memories;
3. message passing.

We developed a software framework to support our experiments, which allows to switch between the transfer methods and adjust various parameters.

Our test platform comprises a TILE*Pro*64$^{\text{TM}}$processor with 64 identical cores. The cores are interconnected by six Networks-on-Chip which have a 2-d mesh structure. Inter-core communication can go through shared memory or via either the UDN or STN using message passing. The shared memory architecture uses three networks which are free of congestion because of high-level flow control, to obtain a bound on the latency however a comprehensive analysis of this memory system would be necessary. The UDN is physically the same as the memory networks but has no high-level flow control. It is however programmer accessible, so software flow control such as for instance circuit switching could be implemented. Hardware circuit switching is implemented in the STN, which therefore offers guarantees on QoS by construction. The considerable overhead and suboptimal network usage inherent to hardware circuit switching however means that it is only suited for long-lived connections that require a lot of bandwidth. This could be improved by multiplexing such as for example Time Division Multiplexed circuit switching.

We then presented our software architecture in more detail. In our "bare metal" implementation we only use an OS to set up the shared memory and network, after which our framework takes over and starts the experiments. The actual tasks consist of sequences of "nop" instructions, the size of the code and dataset ranges from 1000 to 8000 bytes. We implemented five different transfer methods:

- shared memory supported by the coherent shared cache architecture with:
  - cache-pull;
  - prefetch;
  - explicit copy;
- message passing using the UDN;
- message passing using the STN.

In Section 5.3 we related the proposed experiments to the concepts and requirements from the earlier chapters. We focus on one step of a migration, namely the deterministic transfer of tasks.

The tasks that we use in our experiments are not a realistic representation of embedded real-time software. We believe however that our test platform is a plausible representation of future multi-core processors and that our setup is suitable for the research of fundamental techniques. Our experiments can show differences between the transfer methods, but not the isolation of traffic.

Chapter 6

Experimental Results

In this chapter we present the results of the experiments described in Chapter 5. We start with determining the precision of the measurements. Then we compare several metrics of the different transfer methods. Firstly, we look at the absolute transfer times in Section 6.2. In Section 6.3 we evaluate the effect of varying the distance between the source and destination. Then we show the impact of a migration on the execution time of a task in Section 6.4. Finally we analyse the variations between multiple repetitions of the experiment in Section 6.5.

6.1 Precision of the Measurements

In each experiment we measure the elapsed time. The timer that we use for this is based on a cycle-accurate counter that is present in every core. To determine the precision of the timer we performed a series of experiments in which we used a task with a known execution time. We varied the execution time to cover the expected range of the experimental results. Furthermore we repeated each experiment 10.000 times so that we can determine the mean value and standard deviation.

To test the timer we executed the framework that is also used in the migration experiments. We replaced the actual migration by the execution of the following task:

```
void execute_nops(uint32_t number){
        for(int i=0; i < number; i++){
                asm("nop");
        }
}
```

Thus, we execute given number of "nop" instructions and use the framework to measure the elapsed time. This C code translates to the following assembly code:

```
000106a0 <execute_nops>:
```

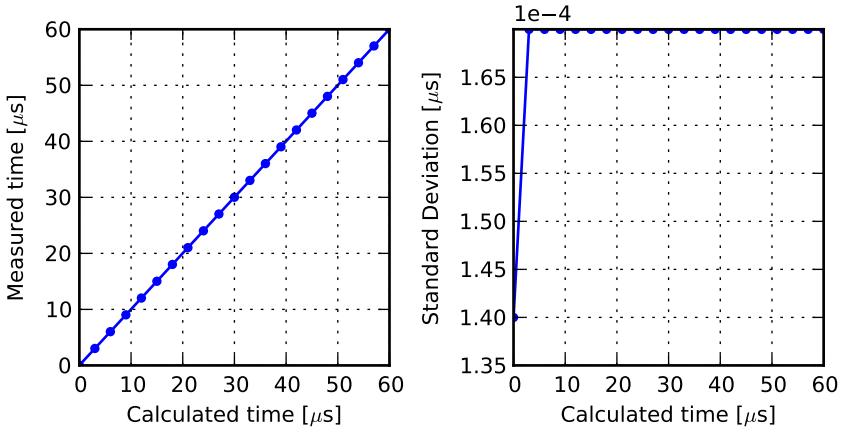

Figure 6.1: Results of the timer experiment. The left graph depicts the calculated time versus the measured time, the right the standard deviation.

```
...
    106c0:      { nop }
    106c8:      { addi r3, sp, 0 }
    106d0:      { addli r4, sp, 8 ; lw r3, r3 }
    106d8:      { addi r5, sp, 0 ; lw r4, r4 }
    106e0:      { addi r3, r3, 1 }
    106e8:      { slt_u r3, r3, r4 ; sw r5, r3 }
    106f0:      { bnzt r3, 106c0 <execute_nops+0x20> }
    106f8:      { addi sp, sp, 16 ; jrp lr }
```

We see that the loop starts at 0x106a0 + 0x20 = 0x106c0 and ends at address 0x106f0, and is seven instructions long. As the processors clock speed is 700 MHz, these seven instructions cost $7 * \frac{1}{700*10^6} = 10$ nanoseconds (ns). We performed the experiments for $number = \{0, 300, 600, ..., 6000\}$, so we expect the results to be the ten folds of these numbers on a nanosecond scale.

The results of the measurements are depicted in Figure 6.1. We see that the measured time is exactly the same as the calculated time, and that the standard deviation is $< 0{,}2$ ns in this time range. We consider this adequate for our experiments. The result of the measurements with $number = 0$ is 31 ns, which gives us the overhead of calling the timer. Furthermore we noted that the maximum absolute difference that was measured between measurements with the same theoretical execution time is 17 ns. We consider it good practice to have this difference smaller than 1% of the measured value, which means our measured values should be over 170 ns.

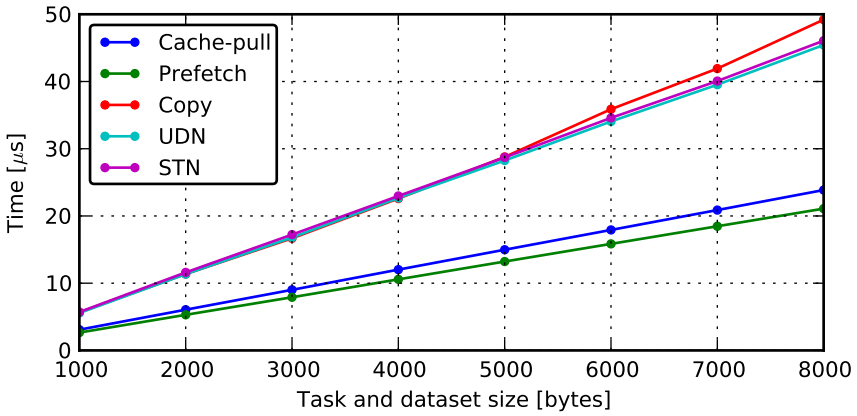

Figure 6.2: The transfer time of the different transfer methods with varying task and dataset sizes.

6.2 Transfer Time

We evaluated the different transfer methods described in Chapter 5 in a series of experiments. In each experiment we perform a migration, followed by one execution of the task. The execution consists of executing the migrated code and accessing all the elements in the migrated dataset. All experiments are performed at worker core number zero and repeated 10.000 times, of which the mean value is calculated to produce the graph. We vary the size of both the code and dataset with steps of 1000 bytes.

The results are depicted in Figure 6.2. The prefetch strategy is clearly the fastest because there the memory hardware is used optimally. A core can continuously instruct the cache subsystem to get the next item, and has no other activities. Thus the interconnect is constantly occupied with satisfying data requests. In the regular cache-pull approach the requests to the cache system are issued while the function is executed and the data elements are accessed. This is slightly slower, which can be explained by noticing that the core must execute an instruction before the next data can be requested and therefore delays are introduced in the data stream. Even while a core might have multiple outstanding load misses (eight on the Tilera architecture), this boundary will be reached soon because none of the data are in the cache yet.

The other three transfer methods take roughly twice as long, namely 46,0 μs instead of 21,3 μs for the largest task size. The overview in Figure 6.3 depicts the actual sequence of operations, based on the code presented in Section 5.2. Note that the timing is not to scale. The lighter squares indicate

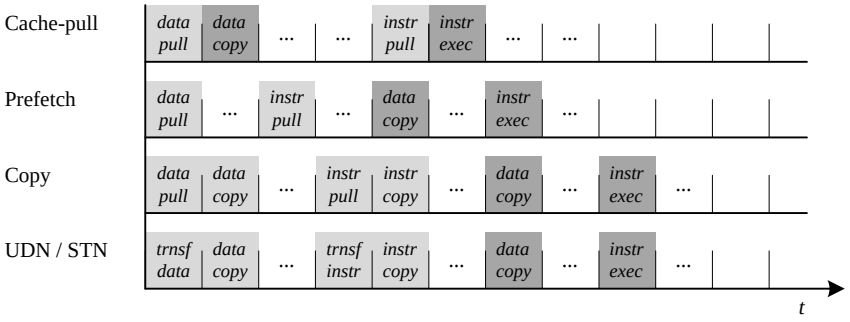

Figure 6.3: The sequence of operations of each transfer method. Light squares represent data transfer operations, dark squares the task execution.

the operations needed for the transfer of data, the darker represent the two steps of task execution. Each operation or pair of operations is repeated a number of times, indicated by three dots. From this figure it becomes clear why the other three methods are slower; in those the transferred data must additionally be stored by the core with *copy* operations. Although the address range to which the data are stored is prefetched beforehand, these copy operations still consume a considerable amount of CPU cycles.

The results of the UDN and STN are very similar. This was to be expected because all networks are physically the same (except the switches and high level flow control) and are always completely reserved for the experiments. If we want to compare the UDN and STN with a shared memory approach, it is more fair to look at the "copy" method because the amount of basic operations is the same. We see that the copy method is slightly slower because the "conservative buffer-preallocation" flow control is slower than the simple ACK messages we used for the UDN and STN.

There are no timing artefacts in the graph and all methods slow down linearly with increasing task and dataset sizes. This is conform the expectations based on NoC theory as presented in Chapter 4, and we conclude that there were no abnormal events during the experiments. The linear factor shows that the transfer methods are scalable in this particular setup. We should note that because all resources are reserved for each experiments, there is no congestion. Thus we cannot see the major disadvantage that all but the STN transfer method have, namely undeterministic behaviour in case of a network overload.

6.3 Transfer Distance

We repeated the experiments from Section 6.2 but now vary the distance between source and destination. Again we take the mean value of a series of

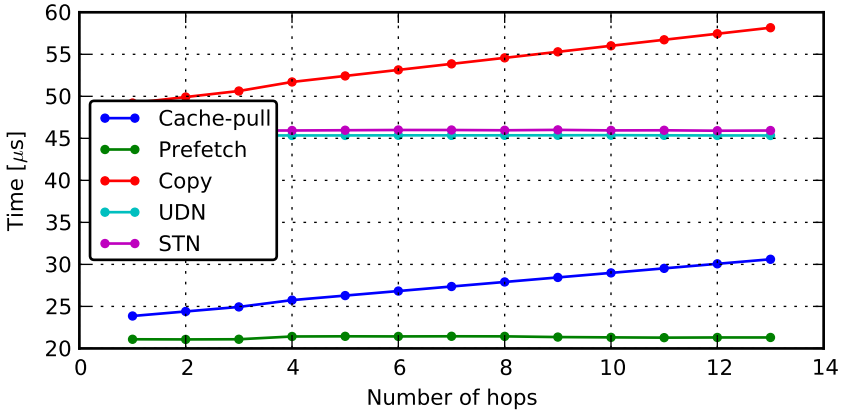

Figure 6.4: The transfer time plotted against the number of hops between source and destination.

10.000 iterations. We plot the measured time against the varying distance, expressed in the number of hops. The results are depicted in Figure 6.4. This graph shows the transfer of a task with a code and dataset size of 8000 bytes. We also performed the experiment with other task sizes but do not show those here, the results are very similar except for linear shifts in the measured time.

We see that the transfer time of the cache-pull method increases linearly with a growing number of hops, the total increase is 25% over 12 hops. This is a relatively small increase due to the 1-cycle per-hop latency of our test platform. The effect of adding one hop to the path is that the *latency* to get a data element increases with some constant. The timing details of each transfer method are depicted in Figure 6.5, in this example we see the load and execute stages of two instructions. As soon as uncached data are accessed with the cache-pull method, the cache system is instructed to load those data. The core must wait for the data to arrive before it can complete the instruction and continue with the next. The latency on communication is thus the limiting factor, and the delay required to communicate over an additional hop is added to *every* load operation. We see that the increase in distance has a significant impact on the transfer time.

The transfer time does increase for the prefetching method. We can understand this by noticing that all transfer operations appear after another, and that the core does not stall in between. While the latency of each load operation increases, a core does not have to wait on the answer and the different transfer operations can be *pipelined*. This can be seen in Figure

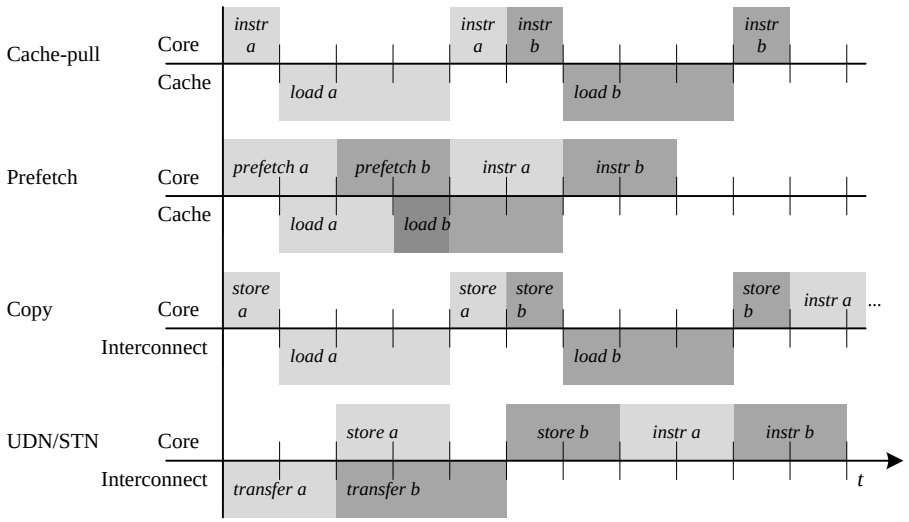

Figure 6.5: Timing details of the different transfer methods.

6.5 because the load operations overlap, which is only possible because the prefetch instructions are not dependent on the arrival of data. Thus, the delay needed to travel an additional hop is added to the transfer time only *once*. The transfer time over thirteen hops is so small that it does not show up in the graph.

Copying code and data explicitly still relies on the cache subsystem, but now the core waits on the requested data again because it must store those to the local L2-cache as can be seen in Figure 6.5. This means the core must wait for the data to arrive between every instruction, in this respect the situation is similar to the cache-pull method except that the instructions must additionally be executed. Therefore the transfer time of this method also increases, namely with 18% over 12 hops.

In the STN and UDN all data is sent in one block, and there is no need to wait for acknowledgements because we know that the complete path is reserved. The timing behaviour is again depicted in Figure 6.5. Unlike the explicit copy method, there are no data request operations in the destination CPU but rather the data is "pushed" by the source after negotiation of the start of the migration. Thus the situation is similar to the prefetching method, and the increased communication latency is hidden. The latency is very low in general because the NoC is clocked at the same speed as the CPUs. We see that pipelining that was presented in Chapter 2 is inherent to some transfer methods, and brings great advantages.

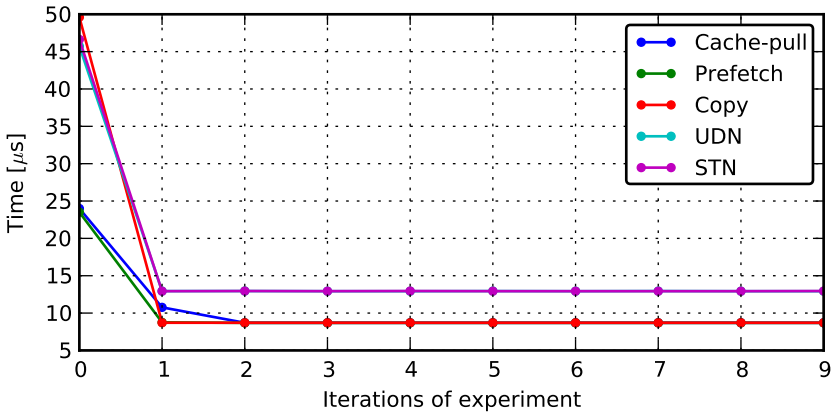

Figure 6.6: The transfer time and subsequent execution time for ten additional iterations of the experiment.

6.4 Execution Time

Next we investigated the effect that a migration has on the execution time of a task. The setup is the same as before, but we now migrate the task only once after which it is executed from the local cache for the remaining experiments. Thus the very first measurement includes the migration, and all following measurements only show the tasks execution time. The task and code size is again 8000 bytes. The experiments are again repeated for 10.000 iterations, the first ten of which are depicted in Figure 6.6. We see that the tasks execution time does not change after the second iteration, regardless which transfer method is used.

We already saw that the transfer time of the cache-pull and prefetch transfer methods are much shorter. In the graph we see one peculiar timing artefact: the execution time need *two* iterations to settle with the cache-pull method. This violates the requirement which states that the execution of a task must be the same after a migration as it was before. With the other transfer methods the migration has no effect on the execution time after the first iteration (which includes the migration), which is conform the requirement. The execution time of the methods based on shared memory are shorter than the UDN and STN methods because the source code of those is slightly different.

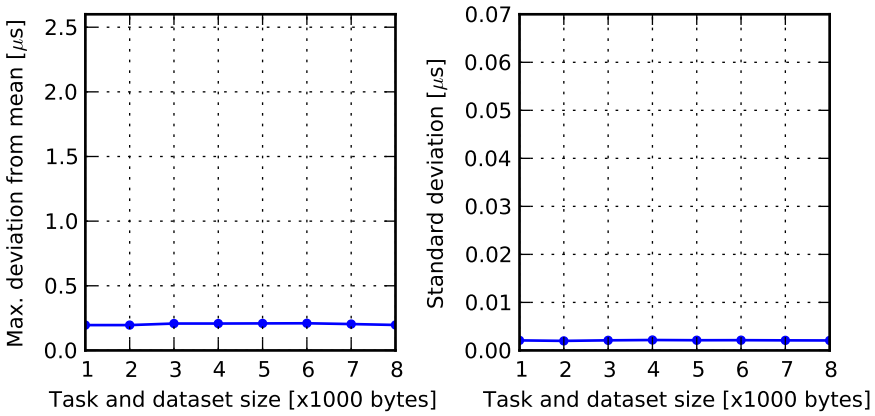

Figure 6.7: Maximum deviation from the mean and standard deviation using cache-pull.

6.5 Variations in Transfer Time

In this section we look at the variations in transfer time between multiple executions of the same experiment. It is essential to evaluate and understand these variations in order to assess whether a transfer is deterministic. We re-use the results of the first series of experiments. From each series we determine the value that deviates the most from the mean value and depict this absolute difference in a graph. Furthermore we plot the standard deviation.

We depicted the variations on the transfer time of the cache-pull transfer method in Figure 6.7. We see that the absolute deviation from the mean is rather constant. This small constant variation of the transfer time comes from the cache behaviour. Between measurements with different code sizes the cache is flushed and some code of the framework must be reloaded in the first iteration, which therefore always takes slightly longer. The standard deviation is constantly 2 ns and $< 0{,}1\%$ of the total transfer time, which is probably adequate for most systems.

Figure 6.8 depicts the variations on task transfers with prefetching. The maximum deviation from the mean is again constant but slightly higher, namely around 0,4 μs. This increase can be explained by looking at the prefetching in more detail. Many prefetch requests immediately follow each other, unlike the cache-pull method where an instruction is executed between two requests. Data requests that are issued so quickly after another lead to "bursty" behaviour of the cache system because of buffering on the

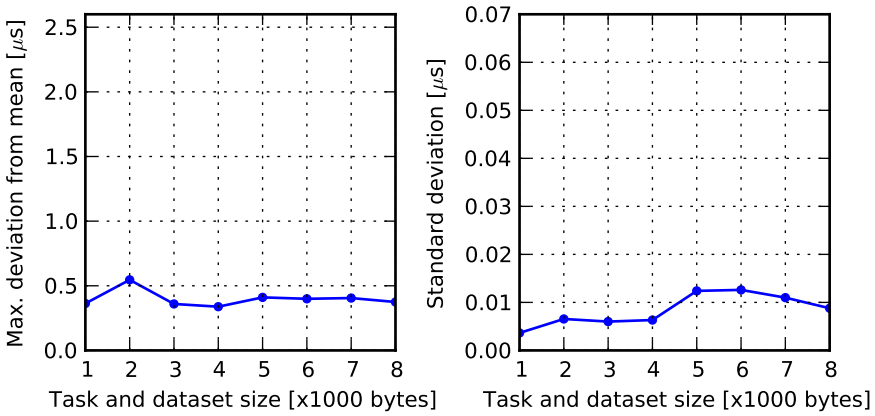

Figure 6.8: Maximum deviation from the mean and standard deviation using prefetching.

tile and in the NoC [49]. The standard deviation reaches a maximum of 11 ns and is thus also higher, but still $< \frac{1}{2}\%$ of the total transfer time. This higher standard deviation indicates that the dispersion also increased.

In Figure 6.9 the variations on the explicit copy method are depicted. We would expect results similar to the cache-pull method depicted in Figure 6.7 but interestingly, they are quite different. The maximal deviation from the mean as well as the standard deviation are much higher and show a non-linear dependence on the task size. This leads to the observation that the behaviour of the cache subsystem is unpredictable if data is not prefetched. As noted, this is inherent to all cache-coherent systems. It shows the trade-off between ease of programming enabled by automated data management on one hand, and increased unpredictability on the other.

The results of task migration using the UDN and STN transfer methods are depicted in Figure 6.10 and 6.11 respectively. The variation on the transfer time is again rather constant for both and around 0,4 μs, similar to the cache prefetch method. The standard deviation of the UDN reaches a maximum of 16 ns which is also comparable to that of cache prefetching. The standard deviation of the STN however ranges between 18 and 70 ns and is therefore considerably higher, although still $< \frac{1}{2}\%$ of the total transfer time. We can explain this because there is no buffering at the receiving side in this network, so the sender is blocked when the receiver does not process the packets immediately and the buffers of the switches in the path are filled. This again results in bursty behaviour, it however has no effect on the maximum deviation as long as the receiver processes the packets regularly.

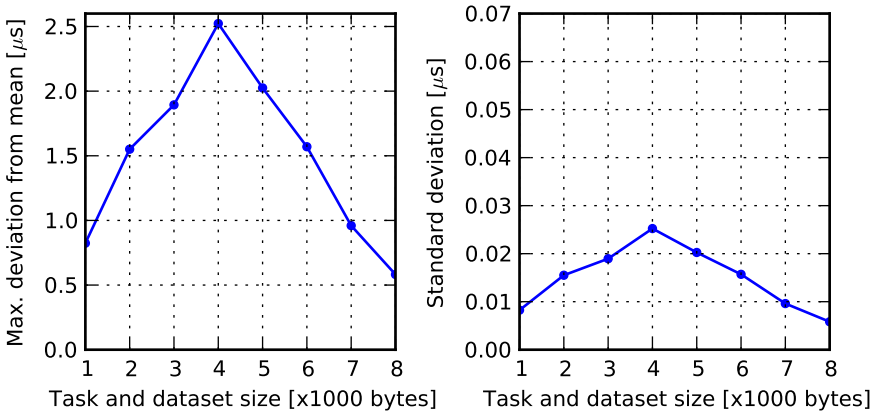

Figure 6.9: Maximum deviation from the mean and standard deviation, using explicit copy.

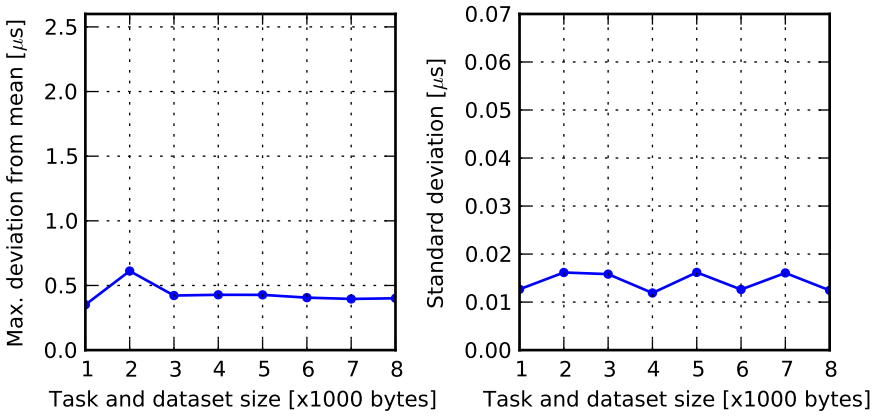

Figure 6.10: Maximum deviation from the mean and standard deviation using the UDN.

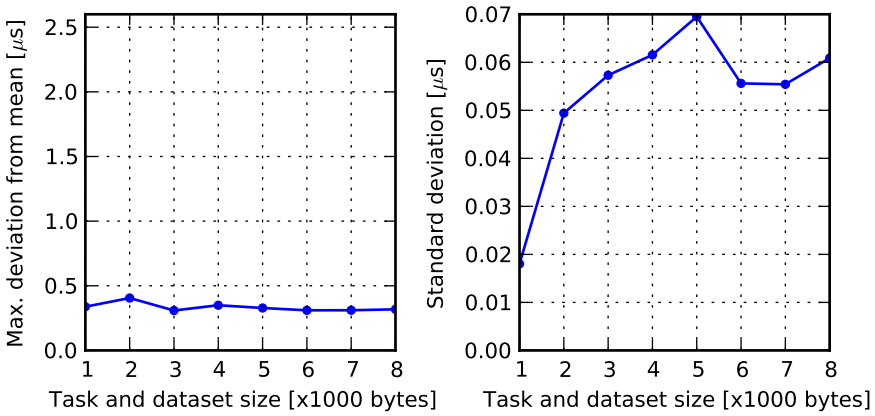

Figure 6.11: Maximum deviation from the mean and standard deviation using the STN.

6.6 Summary and Analysis

At the beginning of this chapter we showed that the standard deviation of the time measurements that we perform on our test platform is $< 0{,}2$ ns, and that the absolute error of our measurements is under 1% for measurements > 170 ns. We then presented four experiments to evaluate the five different transfer methods. In the first experiment we looked at the transfer time of a task migration and found that it grows linearly with an increasing task size. There is no congestion in the network because the NoC is completely reserved for each migration, which is confirmed because there are no timing artefacts. The direct use of the cache subsystem is roughly twice as fast as the others, namely 21,3 μs versus 46,0 μs for the largest task size. This is because in the other methods the core must store the data in the local cache.

In the second experiment we varied the distance between source and destination. The transfer times of the cache-pull and explicit copy methods grow with 25% and 18% respectively when the distance is increased with 12 hops, but remain constant for all others. This is because in those two the core must wait for each data word to arrive, and thus experiences the increased latency on each word. In the other methods the transfers are pipelined so the increased latency is only experienced once. We conclude that the pipelining inherent to some transfer methods brings a huge advantage.

In the third experiment we measured the time of one task transfer followed by multiple executions. The first iteration includes the transfer time and one execution, after which the instructions and data are available from the local

L1 caches and the execution time should be constant. We see this expected behaviour for all but one of the transfer methods; in the regular cache-pull approach it takes two iterations before the execution time is constant. This is an unexpected anomaly because once a task is executed, all data should lie in the L1 caches. While we examined the source code very thoroughly, we cannot explain this phenomenon. From the theory presented in previous chapters however we know that cache-pull is not suitable for deterministic task transfer anyway, it was merely included for comparison.

Finally we re-use the results of the first experiments and determine the variation between successive iterations of a transfer. This indicates how "deterministic" a transfer method is:

- the maximum deviation from the mean indicates how overly negative the worst case transfer time will be on the average;
- the standard deviation indicates how dispersed the results are.

We see that the maximum deviation from the mean of the prefetching, UDN and STN approaches are all constant and around 0,4 μs. This shows that a relatively tight upper bound on the transfer time can be determined. The graph of the explicit copy method shows a strong nonlinear dependence on the task size. The standard deviation of all methods is under $\frac{1}{2}\%$ of the total transfer time. That of the STN approach is however significantly higher than that of the others, which is caused by a lack of buffering at the end points.

The experiments show that four of the tested transfer methods are basically suitable for deterministic task transfer, although prefetching is clearly preferred over the explicit copy method. As noted, we did not test the isolation from other traffic so the experiments do not show the limitations in that respect. We however extracted these limitations from the theory in earlier chapters:

- the prefetching and explicit copying of data relies on the shared cache system which uses three networks with buffer-preallocation, comprehensive timing analysis of this hardware is needed to be able to determine a bound on the latency;
- the UDN has no isolation between traffic flows and therefore no congestion and deadlock avoidance, this could be implemented by flow control in software such as circuit switching combined with the hardwall mechanism;
- when the STN is used the switches must be reconfigured whenever a route changes, these changes must be communicated to all involved switches via another transfer method.

Although each method has certain benefits, none of them is usable "as it is". The least effort for achieving deterministic task transfer will be the use of the STN, perhaps with use of the UDN for control traffic.

On first sight it seems that cache prefetching is the most promising method because it is much faster than the programmer accessible networks (UDN and STN). This is true, but the following must be taken into account. Shared memories supported by shared caches can be based on snooping or directory protocols, the latter requires more implementation effort but has a better scalability [25]. The timing analysis of large shared caches using either of these techniques is however difficult if not impossible [50]. An exception is the caching of read-only data such as instruction memory, which is very useful for task migration [29]. Private caches where each core obtains its data from the main memory or scratchpad memories seem a better approach because their timing behaviour can be analysed. These can be supported by message passing or by communication through shared memory regions that are explicitly defined for the transfer of data. Timing analysis of such a limited number of shared regions is feasible, especially if those contain data objects that are periodically updated by only one actor. They might then be modelled as read-only memory, which greatly reduces the complexity of the timing analysis. In conclusion, we can probably not take much advantage from large shared caches in safety critical real-time systems. An exception are small shared memory regions which are explicitly prefetched before use, preferably with read-only memory. These can be used for the migration of tasks and can be combined with other on-chip communication methods.

Chapter 7

Conclusions and Recommendations

In Section 7.1 we draw the conclusions from our literature study presented in Chapters 1 and 2 and from the concepts we presented in Chapters 3 and 4. Furthermore we recapitulate the results of our experiments described in Chapter 5 and 6. Then we combine these results and draw the overall conclusions. In Section 7.2 we give several recommendations and present ideas for future research.

7.1 Conclusions

7.1.1 Theory

In this thesis we investigated how multi-core processors can be applied in safety-critical real-time systems, especially in avionics. This is motivated by the trend to increase the performance of individual processor boards, as well as to use them more efficiently by consolidating functionality. Multi-cores have the potential to achieve both goals but we must deal with the stringent requirements that apply in safety-critical domains. A concept suitable to deal with this is partitioning, which achieves fault-containment through isolation.

In Chapter 2 we extracted two main challenges:

- dealing with the unpredictability that comes from the concurrent access of shared resources, especially the on-chip interconnect;
- optimizing the hardware usage of multi-core processors without compromising on the determinism offered by static mapping and scheduling.

We presented a basic interconnect model and explained that unpredictability is caused by the concurrent access to shared resources. Networks-on-Chip are scalable interconnects with shared links and buffers. To offer guarantees on Quality-of-Service (e.g. latency and bandwidth), contention for shared resources must be avoided or at least bounded. Usually static mapping and scheduling are applied in safety-critical systems because the correctness of such schemes can be guaranteed, but the efficient use of hardware is hindered because resources must be reserved for the worst case scenario.

In Chapter 3 we used a formal model to propose a combination of concepts to address these challenges. Firstly we extended software partitioning to include inter-task communication so that traffic can be mapped onto the interconnect. Furthermore we extended the concepts of spatial and temporal partition to the hardware. Software partitions (tasks and connections) can be thus be mapped and scheduled onto the hardware partitions (cores and links). Such resource reservation provides a high level solution to the problems with shared resources. To address the second challenge we proposed *mode-based* mapping and scheduling, which allows to switch between multiple precomputed schemes during runtime to optimize hardware usage. Such mode switching requires task migration, we focus on this with our second objective which is to determine the prerequisites for deterministic task migration over a Network-on-Chip.

Task migration enables load balancing, traffic reduction, power and temperature management and the flexible assignment of redundant resources. To switch between multiple static modes and control the changes in a NoC we proposed the use of *transient modes* in which the migrations take place. We deduced the requirements for deterministic task migration and concluded that the *transfer* of tasks is the main new challenge that comes with task migration in multi-cores. Mapping of traffic onto the interconnect is the high level solution to this. For our third objective we focused on this problem in more detail and investigated how it can be made sure that low-level transfers are deterministic by providing guarantees on QoS. We presented several solutions to avoid and bound contention by combining flow-control techniques and buffering strategies. An approach that is very suitable for mapping traffic onto a NoC is circuit switching, possibly multiplexed in time.

7.1.2 Experiments

We performed a number of task migration experiments on our test platform, which features 64 computational cores interconnected by a Network-on-Chip. The goal of the experiments was to investigate which transfer

methods can be used for deterministic task migration. The shared cache system can theoretically transfer data deterministically if prefetching is used, but timing analysis is complex because it must cover the NoC and the shared cache system. Both the UDN and STN are programmer accessible and implement message passing. The former does not feature hardware flow control so this must be implemented in software for it to be deterministic. The latter implements hardware circuit switching and communication is therefore deterministic but not very efficient. Timing analysis of communication via the programmer accessible networks is however much simpler than that of shared caches. Because the programmer must communicate explicitly, code analysis is much simpler. Furthermore only the NoC hardware is involved, for which communication times can be determined quite easily. In each of our experiments the instruction and data memory of a task with varying sizes are migrated and the transfer time is measured.

From the first experiment we found that the transfer time of a task grows linearly with an increasing task size. This means all transfer methods are scalable, which is what we expected. Secondly we found that use of the shared cache system is twice as fast as using the UDN or STN (21,3 μs versus 46,0 μs for the largest task size), but from the theory we know that the cache-pull approach is not deterministic. Therefore prefetching is the only fast method that is suitable for real-time systems. In Chapter 6 however we already noted that there are limitations: the number and size of shared memory regions should be limited and preferably contain read-only data. All other methods are significantly slower because the core must explicitly store data in its local cache. When the distance between source and destination is increased with 12 hops, the transfer times of the cache-pull and explicit copy methods grow with 25 % and 18 % respectively because transfers are not pipelined. We furthermore showed that tasks are executed from local cache after a migration. In the cache-pull approach an inexplicable timing anomaly showed up during this experiment, but the other methods work as expected.

Finally we investigated the variation in transfer times. The variation of the explicit copy method shows a nonlinear dependence on the task size due to the unpredictability inherent to coherent shared caches. This method uses the same memory networks as the prefetch method, so the latter is clearly preferred on our test platform. The prefetch, UDN and STN methods all have a constant maximum deviation from the mean transfer time of roughly 0,4 μs. This shows these methods are deterministic because we can determine an upper bound on the transfer time independent of the task size. The standard deviation of all methods is under $\frac{1}{2}$% of the total transfer time, although that of STN is significantly higher than that of the others due to the lack of buffering at the end points.

We conclude that prefetching from the shared cache and message passing via the programmer accessible networks are suitable transfer methods for deterministic task migration. We see that the coherent shared cache that provides ease of programming for non-real-time applications is complemented by multiple mechanisms that offer deterministic communication for real-time performance. From the theory however we know that none of the transfer methods is usable "as it is", although deterministic communication over the STN can be achieved with little effort. Prefetching is much faster than message passing, but we must keep in mind that it is very hard if not impossible to perform timing analysis of large shared cache architectures. Private caches or scratchpads are more suitable for real-time systems, which can be supported by message passing and explicit data transfer through a limited amount of shared memory regions.

7.1.3 Overall Conclusions

In this thesis we studied a combination of two complex and demanding fields, namely the application of modern *multi-core* processors in *safety-critical real-time systems*, especially avionics. Our contribution is threefold:

- we conducted a literature study and extracted the main challenges, namely to address the predictability of on-chip interconnects and the efficient deployment of software;
- we proposed temporal and spatial partitioning of hardware and software in combination with mode-based mapping and scheduling to address these challenges, and captured this in a model;
- we focused on task migration which is needed for mode-based mapping and requires to address the architectural novelties of multi-cores, and we performed experiments in which we evaluated several transfer methods on our test platform.

These contributions correspond to the three objectives that we set. From the experiments we concluded that four transfer methods on our test platform can transfer data deterministically. Each of these however requires additional work to avoid or bound contention and thus enable guarantees om Quality-of-Service. We showed the complexity of combining resource reservation with lower-level concepts such as routing, buffering and flow control to achieve deterministic communication.

We concluded that it is very hard to analyze the timing behaviour of large coherent shared caches. It is however feasible to use a limited amount of shared memory regions, especially when using read-only data. Private caching strategies and scratchpad memories are also suitable for real-time

systems because timing analysis is simplified. The trade-off between the limited use of shared data objects and private caches or scratchpads is not trivial, as the latter are difficult to use because the programmer must explicitly manage data transfers. A combination of techniques allows designers to select tailored solutions for the different challenges within one system.

The problem with shared resources however persists in all methods, therefore the resources for data transfers must be reserved at design time to ensure an upper bound on data access times. Deterministic inter-core communication can be achieved over reserved connections on programmer accessible networks or through scratchpads or small memory regions that are explicitly shared. If combined with data prefetching, such shared regions are suited for deterministic task migration. On our platform this can be combined with usage of the STN for streaming data and of the UDN when complemented with software flow control. The latter can transport control traffic for the former. Such use of physically different networks is desirable because it is the strongest form of isolation. We see that the combination of multiple mechanisms for explicit data management offer a range of possibilities for deterministic communication. This even includes limited use of the shared coherent cache, which is generally not considered suitable for real-time systems.

Our experiments show that it is possible to guarantee aspects of Quality-of-Service for four out of five methods, but that prefetching has preference over the explicit copy method which uses the same networks. We did however not use sophisticated mechanisms for resource reservation, and did not test to what degree the traffic is isolated. In other words, we could not show under which circumstances the used transfer methods are *not* deterministic. Such experiments require a significant enhancement of the software framework. We did however extract the limitations of each method from the theory. The tasks that we migrated in our experiments are furthermore not a realistic representation of embedded real-time software. This does not affect the validity of our experiments as we studied fundamental behaviour that does not depend on the software functionality. The results of our experiments show that deterministic task migration is possible, and thus confirm the feasibility of mode-based mapping. During this process we showed that the mapping of traffic at design time solves the problem of dealing with a shared interconnect, and we presented several practical solutions for deterministic communication. Overall our experiments confirm the feasibility of the concepts we proposed to deal with the challenges, although there is still much potential for further research.

We believe that our research is very relevant as the interest of manufacturers of embedded systems in multi-cores is growing tremendously. It might

still take several years before multi-cores are applied in avionics because the aerospace industry is very conservative. Our study is valuable input for this lengthy process as it elaborates on the fundamental problems and possible solutions. A short position paper based on this thesis was published in the proceedings of the 3^{rd} International Workshop on Multicore Software Engineering (IWMSE) [26]. Our work is also relevant for real-time systems in other domains such as automotive, medical and automation. Although the constraints in those are not as strict as in avionics, the same fundamental concepts can be used to match the domain-specific requirements.

7.2 Recommendations

In answering our first objective we uncovered many possibilities for further research. Firstly, there are many possible extensions of the software partitions in our model, such as:

- a more complex timing model, including e.g. the arrival time and task period;
- inclusion of additional spatial constraints;
- more complex communication requirements (profiles), including e.g. the bandwidth and latency.

This last item is crucial because the communication requirements of a task must be determined to be able to map the traffic. Such communication profiling is comparable with WCET analysis and is certainly not trivial. Secondly, the hardware partitions in our model can be extended with for example:

- multi-hop connections;
- inclusion of different transfer methods;
- flow control and buffering mechanisms;
- (worst case) timing information;
- additional (external) resources.

Thirdly the concept of mode-based mapping and scheduling can be explored further.

The use of migration to exploit data locality and flexible redundancy are research areas with much space left to explore. We investigated only one of the requirements that we extracted to answer our second objective, namely the deterministic transfer of tasks. The others however also have potential

for further research. Firstly, migrations must be triggered somehow. When mode based mapping and scheduling are used, all cores must be notified of the switch. Such a mechanism is not scalable if centralized. Potential solutions for this problem include the use of self-organizing networks. Secondly, the WCET of tasks on multi-cores strongly depends on time required for communication, which in turn depends on the location of the core to which a task is mapped. The conservative approach is to consider the worst case scenario, but there is much to win if only the feasible mappings are considered for the timing analysis. Such an iterative optimization of parameters offers interesting opportunities for research, for instance using machine learning. Furthermore the concept of transient modes is still an open subject as we did not apply it to specific transfer methods.

To address our third objective we presented a number of deterministic transfer methods for Networks-on-Chip. These are a research domain of their own where routing, flow control and buffering strategies must be tuned while accounting for the network architecture. Furthermore the timing analysis of network traffic and shared caches is in its infancy and contains many interesting challenges. There is also much potential for expanding our experimental setup. Alternative migration mechanisms and other platforms with different transfer methods can be evaluated building on the framework.

When the actual implementation of multi-core processors in a safety-critical design is considered, a number of more practical extensions of the experiment should be evaluated. Firstly, real-time software with realistic timing constraints should replace the software-under-test. Secondly the transfer methods should be improved as suggested at the end of Chapter 6. The framework should be extended so that multiple nodes can communicate simultaneously in order to evaluate the degree of isolation.

Bibliography

[1] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Embedded Syst.*, 2008:1–15, 2008.

[2] M. Al Faruque, T. Ebi, and J. Henkel. Run-time adaptive on-chip communication scheme. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2007*, pages 26 –31, nov. 2007.

[3] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179 – 190, 04-07 2006.

[4] ARINC. ARINC Specification 653P1-2: Avionics Application Software Standard Interface Part 1 - Required Services. Technical report, Aeronautical Radio Inc., Maryland, USA, Dec. 2005.

[5] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[6] D. Barcelos, E. W. Brião, and F. R. Wagner. A hybrid memory organization to enhance task migration and dynamic task allocation in NoC-based MPSoCs. In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 282–287, New York, NY, USA, 2007. ACM.

[7] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70 –78, Jan 2002.

[8] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE International Real-Time Systems Symposium*, pages 149 –160, 3-6 2007.

[9] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study.

In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 15–20, 3001 Leuven, Belgium, 2006. European Design and Automation Association.

[10] T. Bjerregaard and J. Sparso. Scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *11th IEEE International Symposium on Asynchronous Circuits and Systems, 2005. ASYNC 2005.*, pages 34 – 43, March 2005.

[11] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. 50(2-3):105–128, 2004.

[12] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[13] C. Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 1st edition, 2009.

[14] E. W. Brião, D. Barcelos, F. Wronski, and F. R. Wagner. Impact of task migration in NoC-based MPSoCs for soft real-time applications. In *VLSI-SoC*, pages 296–299, 2007.

[15] E. Carvalho, N. Calazans, and F. Moraes. Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs. pages 34 –40, May 2007.

[16] M. Cirinei, E. Bini, G. Lipari, and A. Ferrari. A flexible scheme for scheduling fault-tolerant real-time tasks on multiprocessors. In *IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.*, pages 1 –8, March 2007.

[17] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, pages 684–689, 2001.

[18] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[19] H.-J. Goltz and N. Pieth. A tool for generating partition schedules of multiprocessor systems. In *Proceedings of the 23rd Workshop on (Constraint) Logic Programming (WLP 2009), Potsdam, Germany, September 15-16*, 2009.

[20] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005.

[21] A. Gryc. Zeitpartitionierung am beispiel freisprechsystem. *ElektronikPraxis*, pages 28–31, May 2010.

[22] A. Hansson, M. Coenen, and K. Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 954–959, San Jose, CA, USA, 2007. EDA Consortium.

[23] A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.

[24] J. Henkel, W. Wolf, and S. Chakradhar. On-chip networks: a scalable, communication-centric embedded system design paradigm. In *17th International Conference on VLSI Design, 2004*, pages 845–851, 2004.

[25] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fourth Edition*. Morgan Kaufmann, 2007.

[26] R. Hilbrich and J. R. van Kampenhout. Dynamic reconfiguration in NoC-based MPSoCs in the avionics domain. In *IWMSE '10: Proceedings of the 3rd International Workshop on Multicore Software Engineering*, pages 56–57, New York, NY, USA, 2010. ACM.

[27] H. Jiang and V. Chaudhary. Compile/run-time support for thread migration. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*, pages 58 –66, 2002.

[28] Y.-H. Lee, D. Kim, M. Younis, and J. Zhou. Partition scheduling in APEX runtime environment for embedded avionics software. In *Fifth International Conference on Real-Time Computing Systems and Applications*, pages 103 –109, Oct 1998.

[29] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *30th IEEE Real-Time Systems Symposium, 2009. RTSS 2009.*, pages 57 –67, 2009.

[30] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM*, 32(3):241–299, 2000.

[31] A. Molnos, A. Milutinovic, D. She, and K. Goossens. Composable processor virtualization for embedded systems. In *Proc. Workshop on Computer Architecture and Operating System Co-Design (CAOS)*, Lecture Notes in Computer Science (LNCS). Springer, Jan. 2010.

[32] O. Moreira, J. J.-D. Mol, and M. Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *SAC '07: Proceedings*

of the 2007 ACM symposium on Applied computing, pages 1557–1564, New York, NY, USA, 2007. ACM.

[33] F. Nemati, J. Kraft, and T. Nolte. Towards migrating legacy real-time systems to multi-core platforms. In *IEEE International Conference on Emerging Technologies and Factory Automation, 2008. ETFA 2008.*, pages 717–720, Sept. 2008.

[34] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest. Low cost task migration initiation in a heterogeneous MP-SoC. pages 252 – 253 Vol. 1, March 2005.

[35] Y. Paindaveine and D. S. Milojíčić. Process vs. task migration. volume 1, pages 636 –645 vol.1, jan. 1996.

[36] M. Pastrnak, P. de With, C. van Meerbergen, and K. Goossens. Mixed adaptation and fixed-reservation QoS for improving picture quality and resource usage of multimedia (NoC) chips. In *IEEE Tenth International Symposium on Consumer Electronics, 2006. ISCE '06*, pages 1 –6, 0-0 2006.

[37] R. Pop and S. Kumar. A survey of techniques for mapping and scheduling applications to Network on Chip systems. Technical Report ISSN 1404 0018, Embedded Systems Group, Department of Electronics and Computer Engineering, Jönköping University, 2004.

[38] P. Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1.E.5–1 –1.E.5–10, Oct. 2008.

[39] J. W. Ramsey. Integrated modular avionics: Less is more. *Avionics Magazine*, February 2007.

[40] J. Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.

[41] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 80–89, New York, NY, USA, 2009. ACM.

[42] S. Schliecker, M. Negrean, and R. Ernst. Response time analysis on multicore ECUs with shared resources. *IEEE Transactions on Industrial Informatics*, 5(4):402 –413, nov. 2009.

[43] H. Shen and F. Petrot. Novel task migration framework on configurable heterogeneous mpsoc platforms. In *Asia and South Pacific Design Automation Conference, 2009. ASP-DAC 2009*, pages 733 –738, jan. 2009.

[44] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. pages 389 – 398, 2002.

[45] P. Smith and N. C. Hutchinson. Heterogeneous process migration: The Tui system. Technical report, Vancouver, BC, Canada, Canada, 1997.

[46] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *International Conference on Dependable Systems and Networks, 2003*, pages 625 – 632, 22-25 2003.

[47] Tilera Corporation. TILE*Pro*64$^{\text{TM}}$ processor product brief, 2009.

[48] C. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*, pages 2.A.1–1–2.A.1–10, Oct. 2007.

[49] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, Sept.-Oct. 2007.

[50] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, july 2009.

Acronyms

API Application Programming Interface.

BCET Best Case Execution Time.

CDN Coherence Dynamic Network.

DAL Design Assurance Level.

IDN I/O Dynamic Network.

ILP Instruction Level Parallelism.

IMA Integrated Modular Avionics.

IP Intellectual Property.

ISA Instruction Set Architecture.

IWMSE International Workshop on Multicore Software Engineering.

LRU Least Recently Used.

MDN Memory Dynamic Network.

NI Network Interface.

NoC Network-on-Chip.

OS Operating System.

OSI Open Systems Interconnect.

QoS Quality of Service.

RTOS Real-Time Operating System.

SAF Store-And-Forward.

SoC System-on-Chip.

STN Static Network.

TDM Time-Division Multiplexing.

TDN Tile Dynamic Network.

TLB Translation Look-aside Buffer.

UDN User Dynamic Network.

VCT Virtual Cut-Through.

VLIW Very Large Instruction Word.

WCET Worst Case Execution Time.

Appendix A

Source Code

This appendix refers to the digital copy of the source code that accompanies this thesis, which consists of the following files:

```
/bme_migration/
        bme_frame.c
        bme_frame.h
        bme_mapping.c
        bme_mapping.h
        bme_measure.c
        bme_measure.h
        bme_tasks.c
        bme_tasks.h
        bme_worker.c
        bme_worker.h
        copyright.txt
        linux_client.c
        Makefile
        msg.h
        pci.hvc
```

The hierarchy of the source files is depicted in Figure A.1. In addition to the description of the test setup in Chapter 5 we will now provide some practical instructions on how to use this code. Firstly, the code was only tested with Tilera MDE version 2.1.0-rc.94454 and is *not* compatible with MDE version 3.x or newer. Furthermore the code was only tested on a Tilera TILEncore^{TM}card with a TILE*Pro*64^{TM}processor. The code should

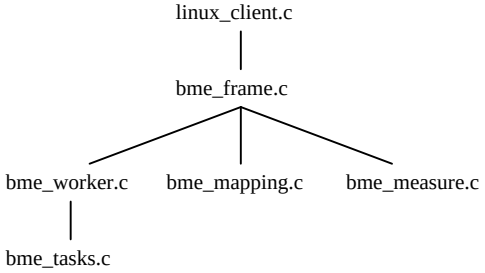

Figure A.1: The source file hierarchy.

be compiled, uploaded and run with the following commands:

```
make all
make run_pci > filename.txt
```

The raw results are printed to the terminal so it is useful to print those to a file for processing later on. To perform the different experiments some of the source files must be manipulated. Firstly, uncommenting the definition of *_TEST_* on line 12 in file *bme_frame.h* performs the experiments for only 100 iterations instead of 10.000, and prints only a summary of the results to the terminal. This is useful to perform a quick check to see if the code functions. Uncommenting the definition of *_SINGULAR_* in the same file on line 15 performs any experiment only with a code and dataset size of 8000 bytes. This is useful to save time for experiments where only those results are of interest. In the files *bme_worker.c* and *bme_mapping.c* many lines of code are preceded by one or more integer numbers, and most of those are commented out. The description of these numbers is listed in Table A.1. To perform a specific experiment, *uncomment* all lines preceded by that number and *comment out* all lines preceded by any other number. The code is currently configured to execute experiment 34. To recreate a figure the concerning experiments must be re-run and the results processed. Figure 6.5 for example requires the results of experiments 34, 36, 38, 40 and 60. Note that experiments 1 to 33 and 46 to 49 were used in the research process but are not related to any of the experiments described in this thesis.

Number	Description
0	Timer test
1	Pull and execute instructions from own L2
2	Pull and execute instructions from L1-i
3	Pull and execute instructions from main memory
4	Pull instructions once from main memory, then execute from L1-i
5	Prefetch and execute instructions from main memory
6	Prefetch instructions once from main memory, then execute from L1-i
7	Load data from own L2
8	Load data from own L1-d
9	Load data from main memory
10	Load data once from main memory, then from L1-d
11	Prefetch data from main memory into L2
12	Prefetch data from main memory into L2, then load from there

13	Prefetch data from main memory into L2 once, then load from L1-d
20	Pull and execute instructions from another L2
21	Pull and execute instructions from another L2 once, then execute from own L1-i
22	Prefetch and execute instructions from another L2
23	Prefetch and execute instructions from another L2 once, then execute from own L1-i
24	Copy and execute instructions from another L2
25	Copy and execute instructions from another L2 once, then execute from own L1-i
26	Copy and execute instructions from another L2 via the UDN
27	Copy and execute instructions from another L2 via the UDN once, then execute from own L1-i
28	Copy data from another L2 via the UDN
29	Copy data from another L2 via the UDN once, then load from L1-d
30	Prefetch data from another L2
31	Prefetch data from another L2 once, then load from own L1-d
32	Copy data from another L2
33	Copy data from another L2 once, then execute from own L1-d
34	Pull and load/execute data and instructions from another L2
35	Pull and load/execute data and instructions from another L2 once, then use L1 caches
36	Prefetch and load/execute data and instructions from another L2
37	Prefetch and load/execute data and instructions from another L2 once, then use L1 caches
38	Copy and load/execute data and instructions from another L2 once
39	Copy and load/execute data and instructions from another L2 once, then use L1 caches
40	Copy and load/execute data and instructions from another L2 via the UDN once
41	Copy and load/execute data and instructions from another L2 via the UDN once, then use L1 caches

46	Copy and execute instructions from another L2 via the STN
47	Copy and execute instructions from another L2 via the STN once, then execute from own L1-i
48	Copy data from another L2 via the STN
49	Copy data from another L2 via the STN once, then load from L1-d
60	Copy and load/execute data and instructions from another L2 via the STN once
61	Copy and load/execute data and instructions from another L2 via the STN once, then use L1 caches

Table A.1: Description of the experiments that belong to the numbers in the source code.