Support for Programming Models in Network-on-Chip-based Many-core Systems

Rasmussen, Morten Sleth

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Rasmussen, M. S. (2010). *Support for Programming Models in Network-on-Chip-based Many-core Systems*. Technical University of Denmark. IMM-PHD-2010-235

Support for Programming Models in Network-on-Chip-based Many-core Systems

Morten Sleth Rasmussen

Kongens Lyngby 2010
IMM-PHD-2010-235

Technical University of Denmark
DTU Informatics
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Abstract

This thesis addresses aspects of support for programming models in Network-on-Chip-based many-core architectures. The main focus is to consider architectural support for a plethora of programming models in a single system. The thesis has three main parts. The first part considers parallelization and scalability in an image processing application with the aim of providing insight into parallel programming issues. The second part proposes and presents the tile-based Clupea many-core architecture, which has the objective of providing configurable support for programming models to allow different programming models to be supported by a single architecture. The architecture features a specialized network interface processor which allows extensive configurability of the memory system. Based on this architecture, a detailed implementation of the cache coherent shared memory programming model is presented. The third part considers modeling and evaluation of the Clupea architecture configured for support for cache coherent shared memory. An analytical model and the MC_sim simulator, which provides detailed cycle-accurate simulation of many-core architectures, have been developed for the evaluation of the Clupea architecture. The evaluation shows that configurability causes a moderate increase of the application execution time. Considering the improved flexibility, this impact is considered acceptable as the architecture can potentially exploit application-specific optimizations and offers a valuable platform for comparing programming models.

Resume

Denne afhandling omhandler aspekter relateret til understøttelse af programmeringsmodeller i mange-kernede arkitekturer baseret op intra-chip netværk. Hovedfokus er overvejelser omkring arkitekturunderstøttelse for et stort antal programmeringsmodeller i et enkelt system. Afhandlingen har tre dele. Den første del omhandler parallelisering af en billedbehandlingsapplikation med formålet at opnå indsigt i udfordringer relateret til parallelprogrammering. Den anden del præsenterer den blok-baserede Clupea arkitektur, som giver mulighed for konfigurerbar understøttelse for programmeringsmodeller med henblik på understøttelse af flere programmeringsmodeller i en enkelt arkitektur. Arkitekturen er baseret på specialiserede netværkinterfaceprocessorer som tillader stor konfigurerbarhed i hukommelsessystemet. Baseret på denne arkitektur bliver en detaljeret implementation af programmeringsmodellen cache coherent delt hukommelse beskrevet. Den tredje del omhandler modellering og evaluering af Clupea arkitekturen konfigureret med understøttelse for cache coherent delt hukommelse. Til evalueringen af Clupea arkitekturen er der blevet udviklet en analytisk model og MC_sim-simulatoren, som muliggør detaljeret simulation af mange-kernede arkitekturer. Evalueringen viser konfigurerbarhed medfører moderate forøgelser i applikationskøretid. Set i forhold den øgede fleksibilitet anses dette som acceptabelt da arkitekturen potentielt kan udnytte applikationsspecifikke optimeringer og udgør en vigtig referenceplatform til sammenligning af programmeringsmodeller.

Preface

This thesis was prepared at DTU Informatics, at the Technical University of Denmark in partial fullfillment of the requirements for acquiring the Ph.D.-degree. The Ph.D.-project was supervised by Professor Jens Sparsø, Assistant Professor Sven Karlsson and Professor Jan Madsen.

Kgs. Lyngby, July 3rd, 2010.

Morten Sleth Rasmussen

Acknowledgements

Many people have supported me during this journey. I am grateful to them all.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In recent years, a wide range of advanced embedded systems have emerged. Personal computing is moving away from traditional computers and transitioning to portable computing devices with constant Internet access. Examples of these are smart phones, portable gaming devices and tablet computers. While embedded systems have traditionally been designed as application-specific energy-efficient integrated circuits, they are now significantly more versatile and are designed as scaled down general computer systems.

1.1 Evolution of Digital System Design

Application-specific integrated circuits and general-purpose processors have existed as separate areas of digital system design for decades. Application-specific systems are generally designed to implement a particular application efficiently and minimize its implementation costs. Mobile phones and digital cameras are examples of such systems. In contrast, general-purpose computer systems are designed with generality and often high performance in mind. However, in recent years, a trend of increasing focus on parallel processing in application-specific systems has caused the two areas to converge. Fig. 1.1 illustrates the evolution of the two types of digital systems.

Application-specific integrated circuits, ASICs, have over time evolved from small gate-level optimized circuits to complex embedded hardware platforms. Each technology generation has enabled increasingly complex systems and called for new design methodologies to reduce the system design costs. As a response

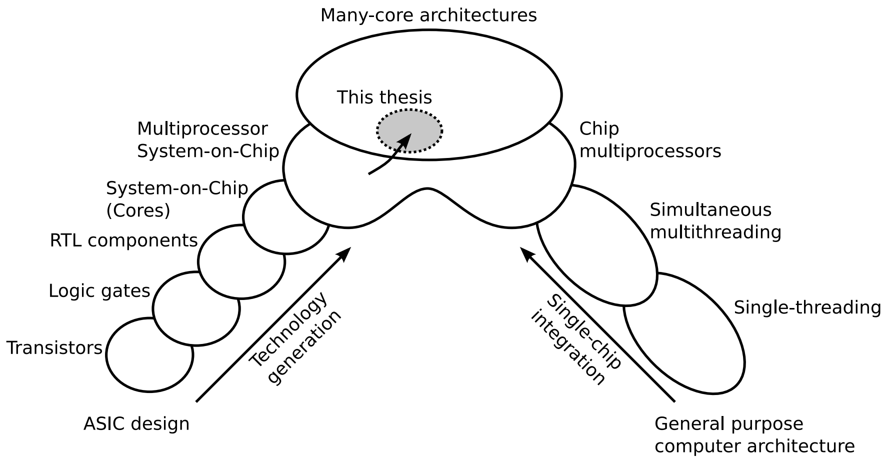


Figure 1.1: Evolution of general purpose computer systems and ASIC/SoC systems.

to this, the basic building blocks of ASICs have increased in size and complexity from being single transistors, through gates and register transfer level components, to pre-designed cores, which may be entire processors. This evolution is illustrated in the left side of Fig. 1.1. In the current era of System-on-Chip (SoC) design, it is possible to compose an entire system using pre-designed cores and integrate it on a single chip. It is therefore a natural next step in the evolution to design multiprocessor SoCs (MPSoCs) by simply adding more processor cores. In this way, the processing demands of future applications can be met without dramatically increasing the design effort and development costs. Furthermore, increasing chip production costs are expected to move the focus of SoC design from application-specific designs to more general SoC designs that are targeted application domains. The emerging trend of MPSoCs means many new challenges related to parallel computing, but also that application-specific system design is converging with the current trends in general-purpose computer system design.

For decades, general-purpose computer systems have been exploiting parallelism in the pursuit of high performance. Multiprocessor systems have been built by connecting processors through a system interconnect. Examples of such systems are the MIT Alewife [6], Stanford DASH [63] and FLASH [59] multiprocessors. Meanwhile, the technology evolution has provided increasing transistor densities that allow increasingly complex processor designs to be integrated into a single chip. As a consequence of this and diminishing returns from processor pipeline improvements, processor designs have evolved from being single-threaded single-core processors to become simultaneous multithreading processors as illustrated in the right side of Fig. 1.1. In recent years, this evolution has continued into chip multiprocessors with multiple general-purpose processor cores and caches integrated in a single chip. Chip multiprocessors are today the standard in general-purpose systems and the number of on-chip processor cores is expected to increase rapidly in future systems. This will eventually lead to many-core chip multiprocessors and cause the boundary between general-purpose computers and MPSoCs to blur.

The current trends indicate that SoC design is heading towards MPSoCs with a high number of processor cores in the future that may go beyond a thousand cores [49]. However, while it is conceptually trivial to imagine many-core SoC systems, it remains largely unanswered how these systems should be programmed. Parallel programming has been studied for decades for high performance general-purpose computer systems, but it is still an ongoing research area. Parallel programming is considered one of the biggest challenges in parallel computing [8] and is a major factor that slows down widespread adoption of parallel systems. It is therefore an obvious move to use existing research to enable the evolution of SoCs into many-core architectures. This raises the interesting question of how existing multiprocessor technologies can be applied to many-core SoCs and what needs to be reconsidered to make these technologies fit into the context of SoC design.

1.2 Multiprocessor System-on-Chip Design Challenges

To lower the design effort, MPSoCs are typically designed using pre-designed cores connected by a shared on-chip interconnect. When the number of cores increases, bus interconnects become bottlenecks in the systems. As a consequence of this, more sophisticated Network-on-Chip (NoC) interconnects have been proposed [25, 10]. Conceptually, NoCs resemble the interconnects found in general-purpose multiprocessor systems. However, due to the different implementation technologies, the design constraints are different. The interconnect of general-purpose multiprocessors typically dedicates an entire chip to handling the interconnect interfacing, while the transistor budget is shared between

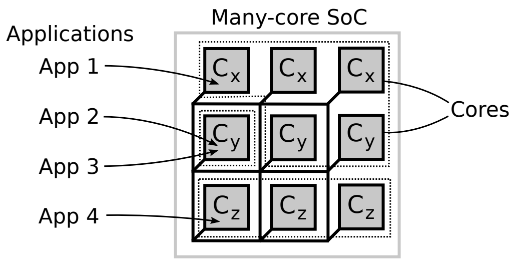


Figure 1.2: Heterogeneous SoC platform with x, y, and z-type cores executing multiple applications simultaneously.

the NoC and the cores in a NoC-based MPSoC. The NoC implementation cost should therefore be kept at a low level to reserve more transistors to implement processing cores.

Furthermore, MPSoCs are typically heterogeneous systems consisting of different types of cores as illustrated in Fig. 1.2 and thus it can not be assumed that the NoC interface can be integrated directly into all cores. In contrast, traditional general-purpose multiprocessors are homogeneous systems consisting of a number of identical processor chips.

Another important difference between the general-purpose multiprocessors and MPSoCs is the applications executing on the system. General-purpose multiprocessor systems are usually used for executing a single application that requires maximum processing performance. On the other hand, MPSoCs may run several applications concurrently, which have different processing requirements and use different resources in the system as shown in Fig. 1.2.

Due to the above aspects it can not be assumed that the hardware support for parallel programming in general-purpose multiprocessor systems can be directly applied to MPSoCs. The hardware/software trade-off must be reconsidered to take the on-chip constraints into account. Furthermore, the lack of an established parallel programming abstraction, i.e. a common programming model, for MPSoCs and the fact that it is unlikely that a single solution for programming future many-core systems will be found means that the hardware support for programming should be flexible. However, one might expect a flexible approach to have a large negative impact on performance compared to a fixed hardware implementation.

The problem statement of this thesis is to determine the general hardware support that is needed for flexible programming of NoC-based many-core systems and estimate the performance impact of this hardware support. The major challenges related to identifying this hardware support are:

- **Flexibility:** Ensure that the hardware support can be used to support a wide range of embedded applications and allow SoCs to be composed of cores of different types.
- **Scalability:** The hardware support must take scalability into account as future many-core SoCs are expected to have hundreds or thousands of cores.
- **Constraints:** The complexity of the hardware support should match the constraints of SoC design. Minimizing the resources used for hardware support enables more cores to be integrated into the SoC.

Addressing these challenges in future many-core architectures is crucial to enable the transition of MPSoC design into the many-core era. The aim of this thesis is to develop a many-core SoC architecture with general hardware support for programming abstractions. The work uses existing programming models as the starting point to propose a many-core architecture with support for multiple programming models. Hardware/software trade-off considerations have a central role in the architecture, which provides increased flexibility through software programability and configurability.

1.3 Contributions

The work leading to this thesis has three main contributions. A more detailed description of the individual contribution is in Sec. 3.4. The contributions are as follows:

- A thorough case study on parallelism and scalability in an image processing application, which reveals the practical parallel programming issues faced by the programmer. The study shows that lack of control over workload distribution on multiprocessor systems can lead to poor cache performance and that these effects are even more pronounced on systems with non-uniform memory access latency. In spite of this and limited available parallelism, reasonable speed-ups are achieved. This work has previously been described in [89, 90].

- The tile-based Clupea many-core architecture, which provides highly flexible generic hardware support for on-chip communication and memory management that can support a wide range of programming models. Implementation of a number of programming models are outlined and a detailed implementation of support for cache coherent shared memory programming is presented and evaluated. The evaluation shows modest execution time increases for a range of benchmark applications when compared to a fixed hardware implementation of similar programming model support. Considering the high flexibility of the architecture which means a high potential for optimizations, this architecture is an interesting alternative to other architectures with fixed hardware support. The Clupea architectural concepts has previously been presented in [87].
- A set of many-core modeling tools, consisting of an analytical model for early design space exploration and the MC_sim trace-driven cycle-accurate many-core simulator. The analytical model provides early performance estimates for the Clupea architecture based on a small set of model parameters. The analytical model has previously been described in [88]. MC_sim provides a configurable component-based simulator targeted at detailed simulation of the Clupea cache coherent shared memory implementation.

Although this thesis takes SoC as the starting point, the work can to a great extent be applied to general-purpose many-core systems due to the previously discussed convergence of the research areas.

The following peer-reviewed papers have contributed to this thesis:

1. Morten S. Rasmussen, Matthias B. Stuart, and Sven Karlsson, "Parallelism and Scalability in an Image Processing Application", *International Workshop on OpenMP, IWOMP*, pp. 158-169, 2008.
2. Morten S. Rasmussen, Matthias B. Stuart, and Sven Karlsson, "Parallelism and Scalability in an Image Processing Application", *International Journal of Parallel Programming*, 37(3), pp. 306-323, 2009.
3. Morten S. Rasmussen, Sven Karlsson, and Jens Sparsø, "Performance Analysis of a Hardware/Software-based Cache Coherence Protocol in Shared Memory MPSoCs", *Workshop on Programming Models for Emerging Architecture, PMEA*, 2009.
4. Morten S. Rasmussen, Sven Karlsson, and Jens Sparsø, "Adaptable Support for Programming Models in Many-core Architectures", *Workshop on New Directions in Computer Architecture, NDCA*, 2009.

1.4 Thesis Outline

The following briefly outlines the structure of the thesis.

- **Chapter 2** is an introduction to the basics of parallel computer architecture, parallel programming models and System-on-Chip design. The chapter should be considered as a condensed summary rather than a complete reference. The chapter also introduces the definitions of the terms used throughout the thesis.
- **Chapter 3** provides an overview of related work in the areas of multiprocessor architecture, System-on-Chip design and Network-on-Chip design. The last section of this chapter gives a more detailed description of the thesis contributions and their relations.
- **Chapter 4** presents a case study on parallelism and scalability in an image processing application. Parallelization strategies and their implementation are discussed and evaluated.
- **Chapter 5** describes the proposed Clupea many-core architecture. The hardware architecture concepts are presented. The chapter also describes the implementation of cache coherent shared memory on the architecture.
- **Chapter 6** presents the many-core architecture modeling tools: An analytical model for early design-space exploration and the MC_sim cycle accurate many-core simulator. This section also presents modeling results of the cache coherent shared memory implementation on the Clupea architecture.
- **Chapter 7** discusses future research directions related to the Clupea architecture.
- **Chapter 8** concludes the work presented in this thesis.

Readers knowledgeable about parallel computer architecture and parallel programming models may choose to browse through the background material covered in Chap. 2 to familiarize themselves with the terminology used in the thesis.

Chapter 2

Programming Many-core Systems

Understanding many-core system design requires fundamental knowledge about parallel computer architecture and System-on-Chip design. This chapter provides an overall introduction to these areas. However, it is not intended as a complete reference on the subjects. Rather, it is a summary for the experienced reader, who possesses a basic understanding of computer architecture. For a more thorough introduction to parallel computer architecture and many-core systems, the reader is referred to textbooks [41, 22, 30, 105]. Additionally, this chapter introduces the terminology used in this thesis.

The chapter is structured as follows. The first two sections give introductions to parallel computer architecture and programming from a general-purpose computing perspective. Sec. 2.3 gives an introduction to System-on-Chip architecture with focus on interconnects and programming.

2.1 Parallel Computer Architecture

Parallelism can be found at many levels in computer architecture and ranges from fine-grained instruction level parallelism to coarse-grained parallelism at application level.

- **Data parallelism:** Single instruction multiple data architectures exploit parallelism through performing operations on multiple data elements in parallel. The overall idea of this architecture is to use multiple functional units in parallel to reduce the processing time. Data parallelism is often applied in processors with instruction set extensions for media processing. Processor architectures with this type of parallelism are also known as vector processors.

- **Instruction parallelism:** Instruction level parallelism exploits the parallelism found in the stream of instructions executed by the processor. Instruction level parallelism can be exploited by pipelining, where multiple instructions are in flight simultaneously. Further parallelism can be exploited by issuing and executing multiple instructions in a single clock cycle. This approach is limited by the data dependencies of the instructions in the instruction stream. The latter approach are is found in super-scalar and very long instruction word processors.

- **Task parallelism:** Task parallelism exists at the application level. An application may be decomposable into a number of tasks, which can be performed in parallel using multiple processor cores. Task parallelism is limited by the data dependencies in the particular application. Thread level parallelism is a special case of task parallelism, where all tasks reside in the same memory address space.

Data and instruction parallelism are applied within the processor by exploiting the available parallelism in the instruction stream. Task parallelism, on the other hand, exploits more coarse-grained parallelism by executing multiple instruction streams in parallel using a number of processors in a multiprocessor system. Since the focus of this thesis is many-core architectures, only task parallelism and multiprocessor architectures are considered in this thesis. However, this does not prevent exploitation of data and instruction parallelism in the cores.

Before going further into multiprocessor architectures, a clear definition of these is necessary. A *multiprocessor system* consists of a set of two or more processors interconnected by a system *interconnect*. The processors may be residing in separate chips, *multichip multiprocessors*, or be processor cores integrated in a single chip, *chip multiprocessors*. Thus, the system interconnect may be on-chip, off-chip or both. Many-core architectures are a subset of multiprocessor architectures, which implements multiprocessing using a large number of on-chip processor cores and an on-chip interconnect.

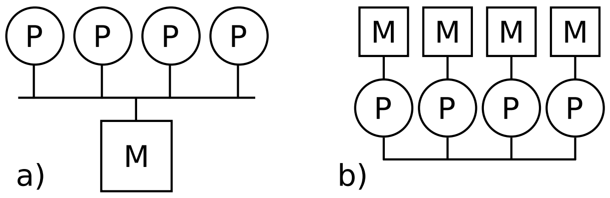


Figure 2.1: Multiprocessor logical memory organization: a) Shared memory, and b) distributed memory. “P” represents processors and “M” is memory.

2.1.1 Memory Organization

Based on the logical memory organization, multiprocessor architectures can be roughly categorized into two different types: Shared memory and distributed memory architectures.

- **Shared memory architectures:** All memory is mapped into a global memory address space, which is accessible by all processors in the system. Data written to memory is eventually visible to all processors in the system. Illustrated in Fig. 2.1a.
- **Distributed memory architectures:** Each processor has its own private memory, which can not be addressed by other processors. A processor can only access its own memory. Access to data held in memory belonging to another processor must be explicitly requested and copied into the local private memory. Illustrated in Fig. 2.1b.

The implementation of the two architecture may be different from these logical organizations. Similar to uni-processor systems, caches are used to improve memory performance and reduce memory contention. Fig. 2.2 illustrates memory hierarchies found in multiprocessor systems. Bus-based memory systems, Fig. 2.2a, can only support a limited number of processors due to bus contention. Shared memory can be distributed among the processors, Fig. 2.2b, to allow fast access to a part of the memory while the remaining main memory must be accessed through the interconnect. Chip multiprocessors often share the last level of caches as shown in Fig. 2.2c.

The simultaneous access to memory by multiple processors in shared memory architectures leads to two issues: Memory consistency and cache coherence. These complicate cache management significantly. The two following subsections will describe these issues.

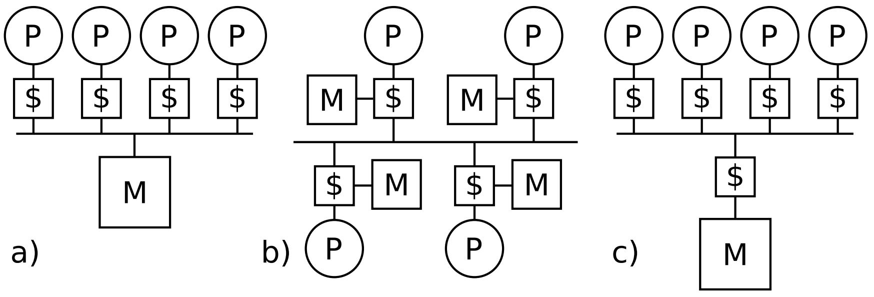


Figure 2.2: Physical multiprocessor memory organizations: a) Bus-based shared memory, b) distributed shared memory, and c) shared memory chip multiprocessor with shared cache. “P” represents processors, “M” is memory and “$” is a cache.

2.1.2 Memory Consistency

In most uni-processor systems, load operations always return the value of the last write to a memory location. In shared memory architectures where multiple processors are accessing memory in parallel the “last” write is no longer clearly defined. Buffering and interconnect latencies may cause writes from different processors to be re-ordered.

The *memory consistency model* of a shared memory multiprocessor defines the programmer’s view of the ordering of memory operations, such as read and write, and synchronization. This includes operations on the same or different memory locations performed by one or more processors. The consistency model is the formal specification of the behavior of the memory system that enables the programmer to write correct programs.

Several consistency models have been proposed. These range from strict to more relaxed consistency models, depending of the provided guarantees. The most commonly assumed consistency model is sequential consistency. Sequential consistency was formally described by Lamport [61] and defined as follows: “The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”. Sequential consistency is considered a strong consistency model since all operations must appear in order and that operations must be visible to all processors.

For a more thorough introduction to consistency models and alternative consistency models the reader is referred to literature [22, 4, 66].

2.1.3 Cache Coherence

In a shared memory system with private caches, multiple copies of a cache line can potentially exist in different caches and main memory as illustrated in Fig. 2.3. When the memory location is written by a processor, a notice about this event must be propagated to all processors with a copy to ensure that reads by other processors will return the value of the last write. How strict this notification is done, depends on the consistency model. For sequential consistency, all write events must be propagated to all processors holding a copy of the affected cache line. Thus, there are four sources of cache misses in multiprocessor systems [22]: i) *Capacity misses* caused by the size of the cache, ii) *conflict misses* caused by the cache organization, iii) *cold misses* caused by first access to a memory location, and iv) *coherence misses* due to cache line sharing.

The *cache coherence protocol* describes how cache coherence is ensured in a given multiprocessor system. The cache controller must be coherence aware and each cache line is typically extended with additional status bits that represents the state of the cache line.

Examples of states found in common cache coherence protocols are: Modified, Shared and Invalid. A cache line in the modified state is an exclusive copy with write permission. Shared cache lines are read-only and multiple copies may exist in other caches. The invalid state is used to flag unused cache lines. Read and write operations to a cache line may require the cache line to switch state as illustrated in the state diagram in Fig. 2.4. State transitions may require several operations to invalidate cache line copies. Extra transition states are often required to keep track of these operations and avoid race conditions. Cache coherence protocols can have more states than the three mentioned here and the exact definition of the states vary from protocol to protocol.

Cache coherence protocols can be roughly categorized as *snooping protocols* or *directory protocols*.

Snooping Protocols

Shared memory architectures with interconnects capable of broadcasting, such as shared busses, can implement cache coherence by exploiting the global knowledge of all memory operations. All cache controllers are able to "snoop" the memory operations performed by all other processors on the bus. When it detects that one of its cache lines has been modified by another processor it either: i) updates the cache line with the data written on the bus, or ii) invali-

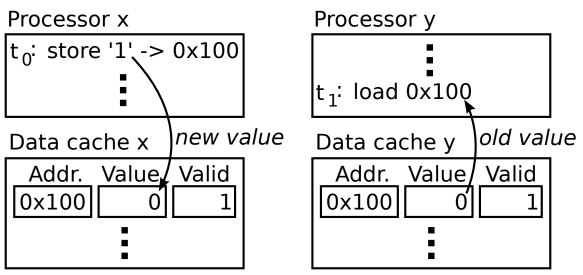


Figure 2.3: Example of the cache coherence issue under sequential consistency: Both processors caches hold a copy of the same cache line. Processor "x" updates the contents of the cache line. Subsequently, Processor "y" has no knowledge about the new contents of the cache line since it already has a valid copy in its cache. A cache coherence protocol is required to prevent this situation.

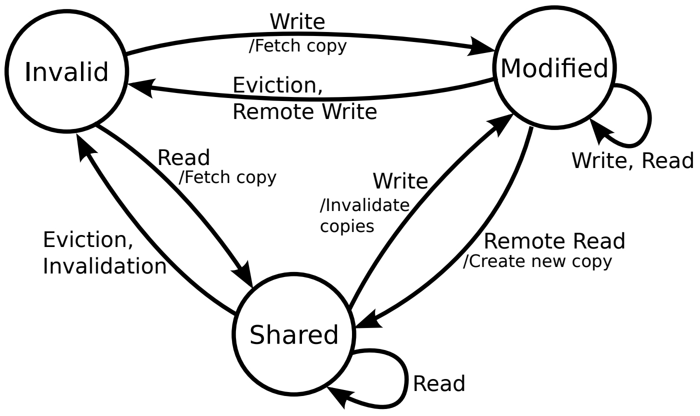


Figure 2.4: Cache coherence protocol example. The principles of the MSI protocol. Transition states are omitted for simplicity.

dates the cache line to force a cache miss that will cause an updated copy to be fetched on the next access. Protocols based on the former approach are known as write-update protocols and the latter approach is known as write-invalidate protocols.

Directory Protocols

Directory protocols are based on look-ups in a common directory rather than broadcasting of memory operations to keep track of shared cache lines. The *directory* maintains information about the location and state of all copies of cache lines in the system. When a processor needs to perform an operation that can affect consistency, the cache controller must consult the directory first. Fig. 2.5a illustrates a write operation. Before a processor can write to a specific cache line, the cache controller must first request exclusive access to the cache line from the directory. The directory ensures exclusive access by sending invalidations to all other caches that hold a copy of the cache line. Similarly, the processor must consult the directory when it attempts to read from a cache line that is not currently held in its cache. The directory ensures that the most recent version of the cache line is forwarded to the processor as illustrated in Fig. 2.5b.

Directory protocols differ in how much information that is maintained about shared cache lines. The directory information may be distributed across several directories depending on the system architecture. Further details on cache coherence is not covered here, but can be found in literature [22, 66].

2.2 Programming Parallel Systems

Programming parallel systems is known to be a complex task. To ease the programming, programmers must use a higher level of abstraction than the bare system implementation. This abstraction is offered by the parallel programming model, which is a conceptualization of the system that the programmer can use for coding applications.

The *parallel programming model* is the top layer of system abstraction and is defined by the underlying compilers, libraries, operating system and programming language. The layers of abstraction are illustrated in Fig. 2.6. Parallel programming models are often implemented as libraries for existing sequential programming languages, language extensions or completely new languages.

The programming model does not necessarily reflect the underlying hardware

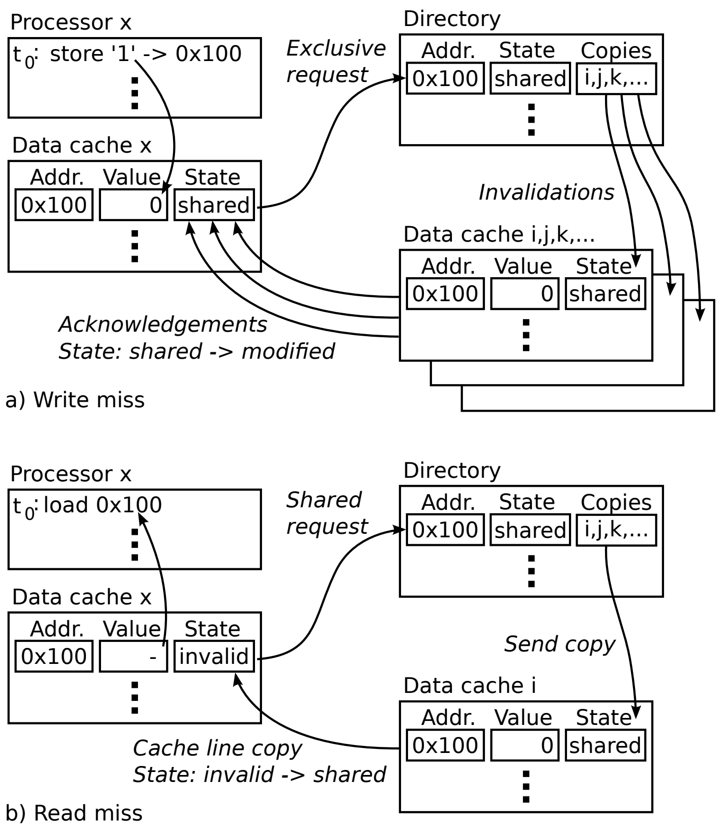


Figure 2.5: Directory-based cache coherence protocol examples: a) Write miss caused by insufficient permissions. Write permission is obtained through the directory, which invalidates all other copies of the cache line. b) Read miss. A cache line copy is obtained through the directory from another cache. The copy could be fetched from main memory as well if the main memory contents are up to date.

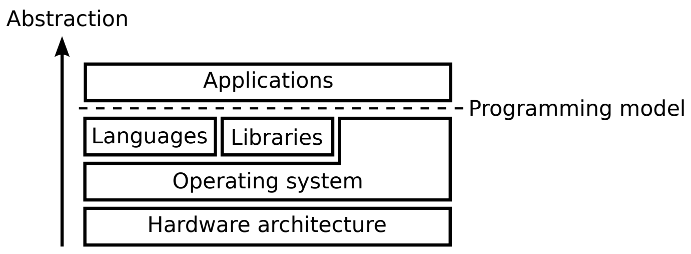


Figure 2.6: Programming model abstraction layers.

architecture. Any programming model can be implemented using compilers and libraries. However, in general, a simple mapping that takes advantage of the programming model support offered by the hardware primitives and functionality of the lower layers of abstraction in Fig. 2.6 leads to a simpler programming model implementation and thus better performance. The ultimate goal of the programming model is to find the highest level of abstraction that still expresses enough parallelism to enable an efficient mapping of applications to the hardware architecture.

A common high-level abstraction of parallel applications is to consider them as a set of tasks. Each task constitutes a part of the application that is executed sequentially on a processor core. Tasks can be executed in parallel, as long as their inter-task data dependencies are obeyed. Programming models have different terminologies and definitions of tasks and how they can interact. Thus, for consistency, *tasks* will be used as a common reference in this thesis for program entities that can be executed in parallel subject to satisfaction of the dependency constraints.

Parallel programming models can be roughly categorized into *shared memory* and *message passing* models. However, since many recent programming models are using concepts from both paradigms, the boundary is becoming rather unclear. The following subsections will describe the basic concepts of the two paradigms.

2.2.1 Message Passing Model

In the message passing model, applications consist of a set of tasks where each task has it own private address space. Interaction between tasks takes place using messages generated through explicit user-level communication operations. These are mostly variants of send and receive operations. The send operation

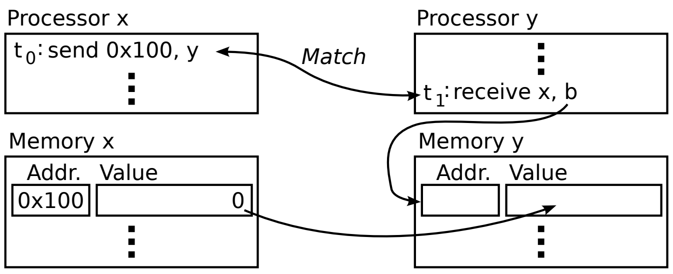


Figure 2.7: The message passing programming model. Data transfer between address spaces of the two processors takes place when a send operation is matched with a receive operation.

specifies a portion of data in the private address space to be sent as a message to a specific receiver. The receive operation specifies a sending task and local receive buffer in the local address space. The message transfer occurs when a receive operation in a task is matched with a send to that particular task as illustrated in Fig. 2.7.

The message passing model can offer blocking and non-blocking variants of the communication operations that can be used to synchronize the execution of tasks. Non-blocking sends can improve performance, since the sender continues execution immediately rather than waiting for the receiver to perform the matching receive operation.

The explicit communication operations force the programmer to identify tasks and their communication, which may be a complex task. Message passing is efficient for communicating large blocks of contiguous data, but may require a large programming effort and cause extra overhead if the communication can not be determined easily. However, the explicit parallelism, lack of shared data, and no implicit data replication leads to good scalability.

2.2.2 Shared Memory Model

The shared memory model has one global address space for all tasks in the application. Interaction between tasks happens implicitly through memory. Any write to a portion of the memory will eventually be visible to read operations by all tasks according to a consistency model, as described in Sec. 2.1.2. The consistency model in the shared memory programming model does not need to match the consistency model assumed by the lower system abstraction layers.

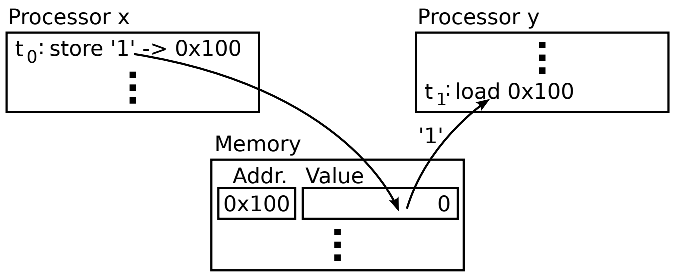


Figure 2.8: The shared memory model. Global address space with implicit communication.

Shared memory has no notion of ownership of data, all task have equal access to all memory as shown in Fig. 2.8. The programmer only needs to identify tasks and communication is handled implicitly.

Task synchronization must be done through explicit user-level synchronization primitives. Common primitives are variants of locks and barriers. A *lock* supports two operations: lock and unlock. The blocking lock operation succeeds when the current state of the lock is "unlocked". The unlock operation releases the lock and allows another task to obtain the lock. The *barrier* primitive allows a number of tasks to rendezvous and continue execution when all participating tasks have reached the barrier.

Compared to the message passing model, the implicit communication of the shared memory model allows a more smooth transition to parallel programming for the programmer. However, the broadcasting nature of writes and the need for complex cache management limit the scalability of the shared memory model.

2.3 System-on-Chip Architecture

Contrary to general-purpose computer architectures, SoC architectures are typically designed for a specific application. Instead of considering the hardware architecture and application programming as two separate tasks, SoC design is based on co-design. Both hardware and software development are parts of a unified design flow. A fundamental step in this flow is to determine which parts of the applications are implemented in software or accelerated by hardware. The trade-offs are individual for each system, depending on the application requirements.

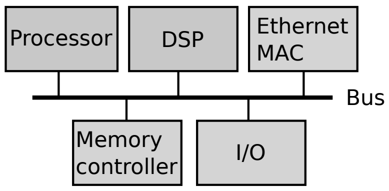


Figure 2.9: SoC design example. Cores interconnected by an on-chip bus.

Given that SoCs are application-specific systems, design costs is a major concern. To minimize the design costs, SoCs typically consist of reusable *cores* connected to an on-chip bus as illustrated in Fig. 2.9. Examples of cores are processors, I/O interfaces, hardware accelerators and memories. A standard bus interface allow cores to be reused across designs.

SoCs are usually based on a common interconnect. Busses do not support concurrent transactions and thus scale poorly with an increasing number of cores. Instead, NoCs have been proposed as a viable alternative for MPSoCs. Furthermore, increasing costs of designing new application-specific chips means that there is a general belief in that future many-core SoCs will be based on flexible platform chips rather than low volume specialized chips. A NoC interconnect can offer both the scalability and flexibility needed for such platforms.

The following two subsection will give a short introduction to NoC and the implications of NoC on SoC programming. For further material on NoC design, the reader is referred to literature [24, 26].

2.3.1 Network-on-Chip

The basic idea of the NoC approach is to replace the bus-based interconnect with a packet-based network consisting of on-chip *routers* and *links* as illustrated in Fig. 2.10. The NoC connects *tiles* consisting of one or more cores that share a single network interface. The network interface of each tile is connected to a NoC router, which is capable of routing packets to other tiles in the system. All inter-tile communication is wrapped into packets in the network interface, sent through the NoC and unwrapped by the network interface at the receiver. A packet consists of header information and payload. The header information is used by the routers and the NoC interfaces to route and identify the packet, while the payload contains the transferred data. Limited width of the NoC links means that packets typically must be transferred as a serialized stream of *phits*.

Many different NoC architectures and topologies have been proposed with different properties. An important factor in the NoC design is the inter-tile communication pattern generated by the application tasks. The NoC must be able to handle the communication pattern without saturating the network. In general, regular NoC topologies, such as mesh and torus networks, offer large bandwidth and flexibility to support different communication patterns. However, excess capacity also means wasting unnecessary resources on the NoC implementation. Thus, a more resource efficient approach is to include the NoC synthesis into the SoC design flow to enable generation of application specific NoC topologies. Application-specific NoCs are often optimized irregular topologies that leaves little or no room for variations in the communication pattern.

NoCs can be categorized as *circuit-* or *packet-switched* depending how communication is managed.

- **Circuit-switched** NoCs resemble the behaviour of point-to-point links by reserving resources for specific connections between tiles. NoC routers support a limited number of connection reservations and thus restrict the number of possible connections. Routing is done on a connection basis rather than for each packet.

- **Packet-switched** NoCs route packets individually using shared resources. No resources are reserved, so the NoC has no limitation on the number of tiles that can communicate. In systems that require guarantees for available bandwidth or latency, logically separate virtual networks can be used to enforce traffic prioritization.

A common issue for all NoC designs is deadlocks. *Deadlocks* can occur if the system can reach a situation where packets block each other in a circular fashion that prevent any of them from proceeding towards their destination. While deadlocks caused by the network itself can be prevented by design, deadlocks can also occur due to the communication of the application itself. A number of solutions to this problem have been proposed [1] that are either based on precise knowledge about the characteristics of the application or end-to-end flow-control. In both cases, deadlocks must be considered in the programming model.

As opposed to the general-purpose parallel systems mentioned in Sec. 2.1, SoCs often consist of a number of different cores in a heterogeneous architecture. The NoC interface must provide the flexibility to allow tiles with different cores

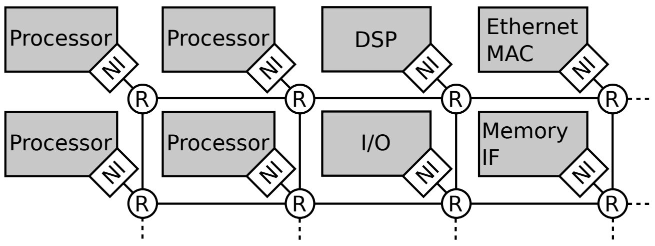


Figure 2.10: NoC-based SoC design example. Tiles connected by a packet-based network through network interfaces, NI, which are attached to the NoC routers, R.

to communicate. NoC interfacing and programming will be described in the next subsection.

2.3.2 Programming System-on-Chip

As previously mentioned, SoC programming is an integrated part of the co-design process. Applications are often represented by task graphs similar to the task-based application representation introduced in Sec. 2.2. A part of the SoC design flow is to map the application tasks to the cores in the SoC hardware platform. When the mapping is determined, each task can be implemented using the programming model provided by its designated cores. This is in contrast to general-purpose multiprocessor systems, which are typically homogeneous architectures used to execute a single parallel application.

Replacing global busses and point-to-point connections with a NoC significantly complicates the SoC design. This complexity must either be included in the programming model or hidden in the lower abstraction layers as illustrated previously in Fig. 2.6. One approach is to let the NoC interface provide a bus abstraction of the NoC interconnect to ensure compatibility with existing SoC co-design flows. The NoC interface is configured to provide virtual busses and point-to-point links that abstract the complexity of the NoC. This approach can also support legacy cores which are designed with bus interfaces. An alternative to this approach is to expose the NoC and let the cores explicitly send and receive messages to specific tiles similar to the message passing programming model. This obviously requires NoC aware cores. Another approach is to integrate the NoC into the in-tile cache controller to provide seamless support for the shared memory programming model.

Chapter 3

Related Work

Many-core SoC architecture is an emerging research area, but it is related to existing research done in the area of parallel general-purpose computers. This chapter has two main purposes: i) To present recent advances in NoC-architecture and programming models for many-core SoCs, and ii) present and compare related work in the area of parallel general-purpose computer systems that potentially can be applied in many-core SoCs. On this basis, the work and contributions of the thesis are defined in more detail.

Multiprocessor systems, SoCs and NoCs are broad research areas. Related work covering these areas can be structured in many different ways. The approach chosen here is to follow the lines of Fig. 1.1 and discuss the trends and previous work on system architecture and programming model support for general-purpose multiprocessor systems and NoC-based SoCs separately. Following this, recent advances in the new overlapping area of many-core architecture research is discussed.

3.1 Multiprocessor Architecture

From a conceptual point of view, general-purpose multiprocessor systems only differ by implementation technology. However, the technology has great influence on the design trade-offs. This has led to two largely distinct generations of multiprocessor system designs: i) Multichip multiprocessor systems, which are composed of discrete processor chips connected by an off-chip interconnection network, and the more recent ii) Chip multiprocessors, which are single-chip systems with multiple on-chip processors and an on-chip interconnect. The following subsections will discuss related work in these areas.

3.1.1 Multichip Multiprocessors

Task parallelism in general-purpose computer architecture gained momentum as the semiconductor technology allowed integration of entire processors in a single chip. Having access to compact and cheap processors it is a natural next step is to combine a number of processors to work in parallel on a single problem to get better performance.

Several large scale multiprocessor systems have basically been constructed using standard processors connected by a custom interconnect [6, 63, 59, 29, 43, 91, 62, 9, 32]. The basic architecture consists of processor nodes, where each processor node contains a processor with private caches, a local portion of main memory and a network interface connected to the memory bus. As a natural extension to this architecture, several of these systems have multiple processors per node connected to the memory bus [63, 62, 9, 32]. The distributed memory architecture is intuitively a good match with the message passing programming model, however, the majority of these systems are designed for cache coherent shared memory due to the conceptually simpler programming model.

Maintaining coherence in large scale multiprocessor systems is a major issue since a simple bus-based snooping cache coherence protocol is infeasible [7, 99, 22]. Instead, these systems must use a more complex directory-based cache coherence protocol.

A number of architectures for directory-based cache coherence has been proposed and implemented. The implementation strategies range from hardware approaches [63, 62] to predominantly software-based solutions [91, 59, 6, 35, 9]. These reflect the general trade-off between fast custom hardware controllers and flexible protocol processor based architectures [73].

In the MIT Alewife multiprocessor [6] the majority of the LimitLESS [17] cache coherence protocol is implemented in a hardware cache controller chip. Corner cases are regarded as rarely occurring and are therefore not implemented in the cache controller. Instead, these are handled in software on the processor, which is designed to have fast context switches to support this approach.

The more recent Piranha [9] is based on interconnecting chip multiprocessors to form systems with a high number of processor cores. Coherence between on-chip cores is maintained through snooping on the on-chip bus. The inter-chip coherence protocol is directory-based and implemented in a more flexible way using highly specialized microcoded protocol engines.

The Stanford FLASH multiprocessor [59] is based on the MAGIC node controller chip. The MAGIC chip acts a combined memory controller and network interface. It consists of specialized hardware interfaces and a programmable protocol processor, which is used to implement the cache coherence protocol. The interconnect interface is accelerated by having hardware support for message preprocessing and hardware buffers. A similar, but slightly more flexible, approach is taken in the Typhoon [91] multiprocessor, which uses a general-purpose processor as protocol processor to support a software implementation of the cache coherence protocol.

Common for these approaches are that the controllers and protocol processors are generally too expensive in terms of hardware to be feasible for on-chip implementation. Their complexity is comparable to the processors that they interconnect. In a many-core architecture, the overhead of the NI should be kept at a minimum to allow more tiles within the chip size limitations.

The implementation of the directory is a major issue related to directory-based cache coherence protocols. The directory needs to store information about all memory locations that are cached in the system. Basically, each cache line must be annotated with a coherence state and a list of caches that currently holds a copy. Depending on the data structure used to hold the information, this may require a substantial amount of memory [66]. This issue is aggravated for many-core systems with on-chip directories, where on-chip memory is a scarce resource. In multichip multiprocessors, the directory is commonly distributed among the nodes and directory information is stored along with the main memory in dedicated directory memory. As the number of cores increases and thereby also the number of caches and potential cache lines copies, the size of the list of sharers grows linearly and becomes excessively expensive. Addressing this issue, a number of approaches have been proposed to reduce the directory storage requirements [78, 66, 2] and improve access latency to the directory data structure [72].

3.1.2 Chip Multiprocessors

The increasing transistor density and diminishing returns from exploiting instruction parallelism in complex processor architectures have called for a new direction in computer architecture. An obvious next step is to target the next level of parallelism and exploit task parallelism by implementing multiple processor cores on a single chip [79] to form a chip multiprocessor, CMP. Today, CMPs are commercially available from all major vendors [54, 48, 3].

The majority of proposed CMP architectures are designed for cache coherent shared memory. Typically, the first levels in the cache hierarchy are private and local to each core and the last-level cache is shared among all cores. The last-level cache is accessed through a bus, a crossbar switch or a more advanced interconnection network. This architecture works well for low number of processor cores, but the last-level cache and shared busses become system bottlenecks as the number of cores increases.

Segmentation of the last-level cache and allowing simultaneous access to segments reduce the bottleneck, but requires a segmented interconnect that leads to non-uniform cache access latency, NUCA. The cache segments are either independent tiles [57, 50, 15] or included in the tiles with the processor cores [110, 16, 39]. The cache line placement is crucial for performance in NUCA architectures. A number of placement and partitioning schemes have been proposed, which attempt to optimize the cache line distribution to achieve better locality [50, 16, 39].

Cache coherence is an issue in CMPs as it is in multichip multiprocessors. Different solutions have been proposed for this.

- **Shared last-level cache:** In architectures where the last-level cache is shared among all processor cores and duplicates the cache lines found in all private caches, cache coherence can be implemented by storing sharing information along the cache lines in the last-level cache [110, 15, 39].

- **Non-shared last-level cache:** Cache organizations where the last-level cache is not globally shared also require coherence among last-level caches. This can be done using a coherence bus [100] or a coherence directory [16, 50].

Few alternatives to the cache coherent shared memory have been proposed for CMPs. The most well known example is the Cell processor architecture [53], which differs from the cache coherent shared memory approach by having private local memories instead of caches for eight of its accelerator cores. Data is transferred between global shared memory and local memories using explicit transfers.

3.2 Network-on-Chip-based Architectures

Early work done on NoC based architectures [25, 10] has been primarily focused on automatic generation of scalable interconnects that can replace the busses and point-to-point links in traditional SoCs and improve design automation [56, 96]. Much of this work consider NoC synthesis flows [51, 11] and NoC optimization methods [34, 76, 75] that can perform application-specific optimizations of the NoC implementation based on an abstract application model represented as a task graph [69].

Meanwhile, the chip production cost has increased rapidly and made customized chip designs an option only feasible for very high volume systems. As a consequence of this, trends are pointing towards embedded platform chips with more general architectures that can be used for a range of applications. In such architectures, NoCs are no longer mere bus replacements, but flexible interconnects that offer the reconfigurability needed to accommodate a range of applications on the same platform. Recent proposals in NoC research follow this trend [68, 37, 98].

3.2.1 Embedded Applications and Programming Models

A fundamental difference between the contexts of general-purpose multiprocessor systems and System-on-Chip is their typical application areas. General-purpose multiprocessor design has been mainly driven by high-performance computing applications, which include scientific applications and commercial server applications. SoC design is driven by embedded applications such as video encoding/decoding, image processing and audio processing, which are all found in any mobile phone today [70]. The algorithms found in many of these applications are difficult to parallelize in a homogeneous way, as it is often possible with large-scale computations. This leads to applications consisting of a heterogeneous set of tasks that needs to be managed in a single programming model.

Furthermore, the processing capabilities of future embedded systems allows multiple applications to execute simultaneously and new applications can be added by the user. This scenario is already valid for the latest generation of smart phone platforms [80].

Dealing with heterogeneous architectures in the programming model is a difficult task. Several frameworks and tool flows have been proposed that address this issue [77, 102, 84, 83, 52, 60].

C-HEAP [77] proposed a streaming oriented programming model and synthesis flow for embedded signal processing where data is communicated as tokens between cores. This abstraction of communication fits well with abstract application models used in design flows for application-specific embedded systems. However, the lack of support for shared data may lead to inefficient parallelization. TTL [102] is a similar task-orient token-based programming model, which relies on token interfaces implemented in hardware or software.

Paulin et al. [84, 83] introduced the MultiFlex approach which targets automated platform generation for multimedia and networking applications. The programming model supports both shared memory programming and Distributed System Object Component programming, which is inspired by remote procedure calls. A system object broker is used to keep track of available cores and distribute tasks among them.

High level programming model approaches [52, 60] completely abstract the hardware architecture and rely on libraries and compilers to determine how tasks communicate. Common Intermediate Code [60] uses channel-based communication by default, but also supports other alternatives such as shared memory.

Attempts to evaluate programming models for embedded systems by comparing streaming and shared memory programming models [85, 65] have lead to inconclusive results. The best choice of programming model is highly application dependent. However, despite this fact, a large fraction of the research in embedded architectures is focusing on streaming programming models due to its explicit expression of parallelism.

3.2.2 Programming Model Implementation

The hardware/software trade-off of the programming model implementation is an important aspect of NoC architecture, i.e. how much hardware programming model support is needed. Implementing the programming model fully in hardware may take up a substantial amount of transistors [64], which could otherwise be used for additional cores. The trade-off involves both the NoC interface and also shared services needed to manage execution of tasks in parallel on a number of cores. As mentioned previously, a lot of work has been done on interconnect interfaces for multiprocessor systems. However, surprisingly little work has been done in this area in the context of NoCs. Most work in NoC and MPSoC design is either considering the network itself or focusing only on high level interface abstractions.

NoC interface implementations can be put into two categories depending on NoC awareness of the cores as proposed in the comparison done by Bhojwani and Mahapatra [12].

Wrapper Interfaces

Wrapper interfaces encapsulate NoC non-aware cores and translate core requests into NoC packets. Examples of this approach are bus and protocol wrappers that allow cores with interfaces such as OCP and AXI to be seamlessly connected through the NoC. This type of NoC interfaces are found in the AEthereal [86] and MANGO [13] NoCs.

The MANGO interfaces [13] are based on the OCP protocol which specifies point-to-point connection between master and slave cores. Each NoC interface presents its core with an OCP complaint interface counter-part. The AEthereal NoC [86] provides additional flexibility by supporting a range of interface wrappers that allow cores with different interfaces to communicate through the NoC using a shared memory programming model. It is common for both of these approaches that they do not consider private caches in the system.

In the more recent approach by Hansson and Goossens [38], the support for heterogeneous architectures is improved by offering a set of compatible hardware wrappers that allow communication between memory mapped cores and simple streaming cores.

Direct hardware implementations of data-flow and token-based programming models for MPSoCs have also been proposed using wrapper interfaces [42, 20]. Here, the NoC interface acts as a core controller, which controls the core by generating requests to the core when new data tokens arrive. When the token has been processed, the NoC interface is responsible for passing token on to the next core.

Wrapper interfaces have also been used to add hardware support for synchronization. Monchiero et al. [74] propose a specialized hardware unit to maintain spin-lock variables coherent in non-coherent shared memory systems. Others have proposed hardware support for task queues [58].

Combined NoC interfaces and cache controllers for shared memory systems [18] similar to those previously described in multiprocessor systems [63, 6] also belong to this category. Here, communication is done implicitly through the shared memory programming model and handled by NoC interface autonomously.

Core Interfaces

The core based network interface is based on a NoC aware core that is capable of handling packetization itself. This type of network interface allows the NoC interfacing to be considered as a hardware/software trade-off. On programmable cores, the packetization can be done partially in software to reduce the interface hardware complexity at the expense of higher latency [12].

DMA-like communication co-processor NoC interfaces [36, 31, 53] can be considered as a combination of the two approaches. The cores are aware of the distributed nature of the system architecture, but non-aware of the interconnect details, which are off-loaded to the communication co-processor. Among interfaces of this type, the memory flow controller found in the Cell processor [53] is a well-known example. Here, data blocks are transferred between distributed memories on request by the processor core.

The programming model can also be supported by special purpose cores. The previously mentioned Distributed System Object Component programming model [83] is based on a hardware implemented broker that distributes service requests to available cores in the system.

3.3 Many-core Architectures

The evolution of SoC architectures into more general many-core platforms and CMPs into to massively parallel many-core architectures have lead to a convergence of the two areas. Tile-based many-core architectures have recently started to emerge [101, 95, 103, 55, 46]. Early many-core architectures were mainly targeted at data-flow applications, such as signal processing, and consist of relatively simple cores.

The MIT Raw processor [101] offers a general tile-based computation fabric interconnected by an on-chip mesh network. The processor cores are tightly integrated with the interconnect through communication FIFOs that are accessible as processor registers. These allow fast passing of single data words between neighboring tiles, which can be exploited in data-flow applications such as video processing.

The TRIPS architecture [94] is a highly configurable data flow oriented architecture that attempts to exploit both instruction and task parallelism. The architecture is organized into four tiles that each contains 16 simple computation nodes. The architecture relies heavily on the ability of the compiler to extract parallelism of the applications. A similar approach is taken in the AsAP architecture [108], which targets low-power digital signal processing applications.

In line with these architecture, Vangal et al. [103] presented an 80-tile processor for high-throughput floating-point applications. The architecture consists of an array of simple floating-point units with small local memories that can communicate through explicit data transfer instructions.

More recent architectures are based on homogeneous set of processor cores similar to those found in CMPs, which are interconnected using a NoC. The shared memory programming model is supported by all of them, however, the hardware support for cache coherence has been reduced or completely removed in the most recent architectures [55, 46].

In the Larabee many-core architecture [95], each tile holds a part of a coherent level two cache and a private local memory. All tiles are interconnected with a ring-based network similar to the one found the Cell processor [53]. Cache coherence is implemented through the level two cache and the ring interconnect. The private local memories are non-coherent and managed using explicit DMA-transfers.

The Rigel architecture [55] uses a global shared address space but has no hardware support for global coherence. The tiles consists of a cluster of eight processor cores, which have access to a shared cluster cache. On-chip global caches are accessible using special global memory operations, that bypass the cluster cache. Global coherence can be implemented through software based on these operations.

Recently, Howard et al. [46] presented a 48-tile many-core architecture targeted at energy efficient server computing. The architecture has moved away from hardware implemented cache coherence, instead the architecture is optimized for message-passing through shared memory. Each tile has a private level two cache that is non-coherent and a hardware message buffer. Communication between cores must be done explicitly by either explicit messages or by flushing the private cache of the sender and forcing the receiver to read the data from main memory.

The addition of the expected heterogeneity of future applications to the already hard problem of programming massively parallel architectures means that programming these systems is currently one of the pressing challenges in digital system design [56, 106, 8, 70, 69]. The variety of the approaches to programming and programming models support found in recent architectures expresses the general uncertainty over how this should be done and indicates that it is unlikely that a single approach can fit all applications.

3.4 This Work

Based on the discussion of related work in the previous sections, it is possible to describe the work that has lead to this thesis in more detail. As previously mentioned in the introduction in Chap. 1, the overall idea of the thesis is to take a holistic approach to programming model support and identify the hardware support needed to aid in future many-core SoCs, which are expected to be massively parallel architectures with hundreds or thousands of cores. Many concepts are already known from research in multichip multiprocessor systems and can be applied to SoC design, but on-chip multiprocessing put them into a new context, so the design trade-offs must be reconsidered. Little work has yet been done on programming model support for many-core SoC architectures despite its importance for future SoC designs.

The hardware/software interface design challenges introduced in Chap. 1 can now be revisited and refined to define the aim of the thesis more precisely.

- **Flexibility:** Future SoC hardware platforms need to be more general to facilitate platform reuse to reduce design costs. Customization is mainly possible through software. Furthermore, no single programming model is likely to fit all applications. Thus the hardware/software interface should support a plethora of programming models and allow application-specific optimizations.

- **Scalability:** Memory hierarchy design and communication are key aspects in many-core architectures. Private caches are crucial to mitigate the communication latency of a NoC-based interconnect. The NoC interface is a central part of the hardware/software trade-off of the programming model support as all inter-tile communication and accesses to non-local memories must be translated into packets transferred across the NoC.

- **Constraints:** The design trade-offs of many-core architectures are different from the architectures proposed for multichip multiprocessors. The processor core and the interconnect interface can be more tightly integrated, but fewer resources can be used for complex interfaces as these resources could be used for more cores. The amount of hardware support for programming models is therefore limited.

The main focus of this thesis is the design trade-offs in NoC interfaces for many-core architectures with respect to programming model support. The aim is to identify generic hardware support that can be used to provide flexible programming model support that can aid programming of future many-core SoCs. The key idea is to integrate the NoC interface and the cache controller into a single programmable network interface processor. The NoC architecture itself is outside the scope of this thesis, however, certain aspects must be considered in the network interface design. In these cases, as few assumptions as possible are made, to make the proposed many-core architecture as NoC architecture independent as possible.

Overall, the thesis work consists of three parts, which are described in Chap. 4, 5 and 6.

- **Case study of parallel programming:** Chap. 4 describes a case study on parallelism and scalability in an image processing application. This work was done to achieve practical experience with parallel programming and to better understand the related challenges. The case study evaluates three parallelization strategies using the OpenMP shared memory programming model using a cache coherent shared memory multichip multiprocessor system. The study reveals programming issues related to cache performance due to lack of memory system awareness in the OpenMP programming model.

- **The Clupea many-core architecture:** Chap. 5 presents the tile-based Clupea many-core architecture. The key concepts of the architecture are allocatable processing tiles and individually configurable support for programming abstractions for each application that execute on the system. The programming model support is based on a specialized programmable NoC interface processor, which integrates NoC interfacing and parts of the cache controller. In contrast to previous multichip multiprocessor interface processors [91, 59, 6], the Clupea approach focuses on the on-chip hardware constraints of single-chip many-core architectures and more flexibility to support a plethora of programming models. Compared to the recently proposed dual-controller distributed shared memory implementation by Chen et al. [18], the Clupea architecture uses a single network interface processor and provides full support for cache coherent shared memory.

  Implementations of a number of programming models are outlined briefly, while a scalable implementation of cache coherent shared memory is discussed in detail. Cache coherent shared memory is a widespread programming model supported by many existing multiprocessors. It therefore represents an important base line programming model to support. The cache coherent shared memory implementation is based on directory-based cache coherence protocol implemented using generic processing tiles as directories which allows the programming model support to be scaled to match the needs of the application.

- **Many-core architecture modeling:** Chap. 6 deals with many-core architecture modeling. An analytical model and the MC_sim cycle-accurate system simulator are presented and used for evaluation of the Clupea architecture configured for cache coherent shared memory support. The analytical model provides early estimates of the performance of the architecture, while MC_sim provides detailed simulation of the architecture based on trace-driven simulation. Results show a modest overhead of the Clupea architecture compared to fixed hardware implementation of similar support for cache coherent shared memory. Directory latency is identified to be the main limitation of the proposed implementation, however, the flexibility of the Clupea architecture allows the number of directories to be scaled to mitigate this limitation.

Chapter 4

Parallelization of an Image Processing Application

Applications for emerging many-core systems aiming at embedded and personal computing are different from the traditional applications for multi-processor systems. Applications for embedded computing are typically working on small data sets, which makes parallelization much more challenging. On top of this, the embedded applications often have to share system resources with other applications and services. Resources are often limited and system response latency is more important than throughput.

This chapter is a case study of a potential application for an embedded many-core system. The study analyzes the parallelization of an image processing application. The aim is to get practical experience with the parallelization challenges introduced with this new class of parallel applications. The chapter is based on two published papers [90, 89] and is structured into the following sections. The first section introduces the case study. This is followed by an overview of the main algorithms of the application in Sec. 4.2. Section 4.3 describes the parallelization strategies. Section 4.4 describes the implementation of the parallelization strategies. Results and a discussion of these are given in Sec. 4.5. Finally, a summary of the case study is provided in Sec. 4.6.

4.1 Introduction

Programming parallel systems is challenging. It remains unclear if the existing parallel programming models from high performance multiprocessor systems are suitable for embedded systems. Thus, there is a need to explore parallel programming models for embedded applications to expose the influence of the constraints of the embedded systems.

So far the programming models for multi-core architectures have been very similar to those for shared memory multiprocessors. OpenMP [81, 82] is one of these shared memory programming models. OpenMP offers a multi-threaded programming model based on a set of compiler directives and library calls. Instead of explicit thread management, OpenMP controls threads, synchronization and work distribution implicitly based on the parallelism exposed in the code through the use of compiler directives. Thus, explicit thread management and lock-based synchronization, which are both complex and error-prone, are largely avoided. One example of the parallel constructs supported by OpenMP is automated parallel execution of for-loop iterations.

Since embedded systems are an emerging area for parallel programming, efficient programming models for these systems have yet to be found. Shared memory has been used for decades in traditional multiprocessor systems and is therefore an obvious starting point for exploring parallelism in applications for embedded systems.

This case study considers an embedded image processing application for object identification using multi-spectral images. Parallelization is studied using OpenMP 2.5 [81]. The contributions of this case study are: i) The analysis of an embedded image processing application; ii) A thorough performance evaluation of the parallel properties of the application using OpenMP.

The major challenges faced when parallelizing the application were to extract enough parallelism from the application and to reduce load imbalance. The experimental results show that, with some tuning, relative speedups in excess of 9 on a 16 CPU system can be reached.

4.2 Application Overview

This case study is focusing on an image processing application developed at DTU [19] and written in Matlab. The application is used for object identification based on multi-spectral imaging and can be used for many different purposes. One example is identifying the species of a *Penicillium* fungus in a

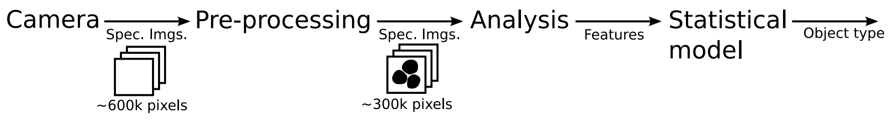


Figure 4.1: Overview of the entire image processing application application.

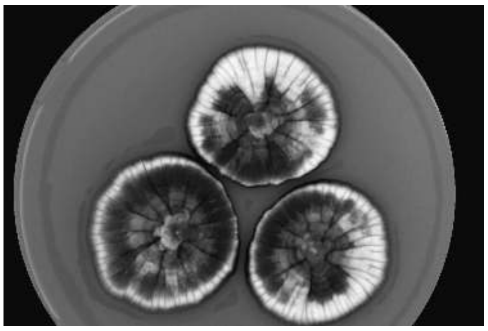


Figure 4.2: Spectral image of fungi colonies.

petri dish from a multi-spectral image [19]. The object identification is based on information extracted from the images in the form of scalar values, called *features*, that each describes some aspect of the input image. Features are grouped into *feature sets*, based on extraction method used for the particular features.

The flowchart in Fig. 4.1 gives an overview of the application. It consist of three major parts: Pre-processing, analysis and a statistical model. The application input is a multi-spectral image of the object that has to be identified. The multi-spectral image is a set of spectral images, where each spectral image shows the object exposed to single colored source of light. Different wave lengths of light reveal different elements in the object. Fig. 4.2 shows an example of a spectral image of fungi colonies.

The pre-processing part involves preparing the raw input image for processing, which means removing unnecessary information in the image and normalizing the image. The analysis part is feature extraction based on arithmetic and morphological operations and scale space analysis. The extracted feature sets are used in the last part, the statistical model, to classify the object using known statistical characteristics of the object types to be identified.

The case study will focus on the pre-processing and analysis based on features from arithmetic operations as these are the most computationally intensive parts of application. Furthermore, the case of identification of fungi is based on features extracted using these operations. The statistical methods for classifying the contents of images are outside the scope of this study and are described elsewhere [19].

The remaining parts of this section will describe the application in more detail.

4.2.1 Pre-processing and Mask Generation

The pre-processing of the multi-spectral input image involves two steps, i) the actual pre-processing and ii) the mask generation.

The pre-processing step produces a noise-filtered normalized image. First, the pixel-wise average intensity across spectral bands in the multi-spectral input image is found. The mean of the resulting single-channel image is found and subtracted from each pixel. Following this, each pixel is then divided by the standard deviation to produce the normalized image. Finally, a 3×3 median filter is used to filter noise. These steps are illustrated in the more detailed overview of the application in Fig. 4.3.

The mask is used to select the interesting parts of the image, thus its generation varies depending on what information is extracted. For the input images used in this study, edge detection is used to find areas of interest in the images. For each pixel in the single-channel image previously constructed by pixel-wise average of the spectral images, the magnitude of the numerical gradient $|(\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y})|$ is calculated where f describes the pixel values as function of coordinates (x, y). The median of the gradient values is found and all pixels whose gradient are greater than or equal to the median are included in the mask. They correspond to interesting areas in the image. The mask can be seen as a bit field where each bit corresponds to a pixel in the image. Each bit indicates if the pixel should be considered or not.

4.2.2 Arithmetic Feature Extraction

The mask is applied to each spectral band in the input multi-spectral image by discarding all pixels *not* in the mask. Five feature sets are extracted from the masked spectral bands of the input image, using five different arithmetic operations. Two operations take a single band at a time, while the other three operate on all pairs of bands. The two single-operand operations are the identity function, which just pass the image data through, and the pixel-wise base-10 logarithm. The other three operations find the pixel-wise difference, product

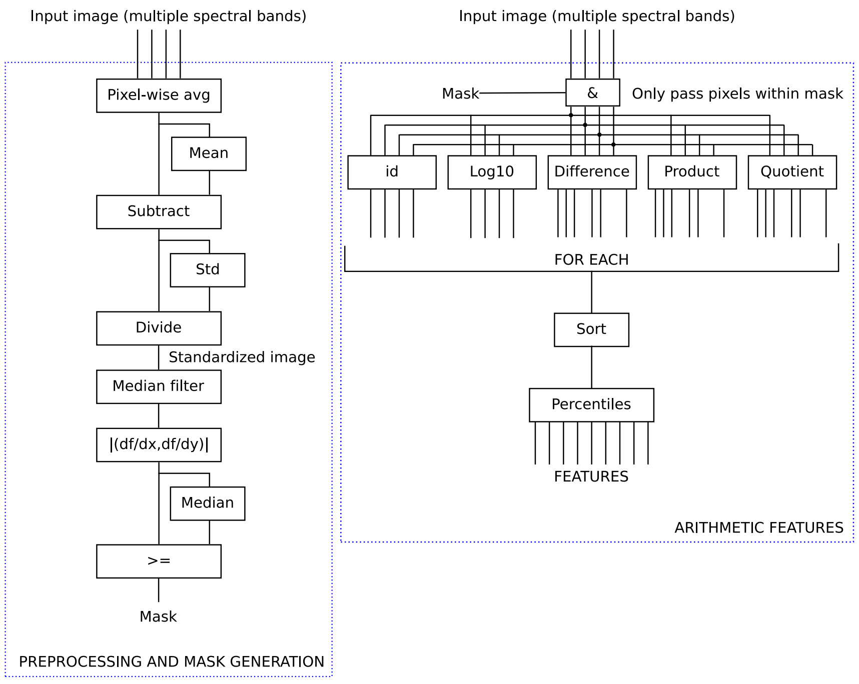


Figure 4.3: Overview of immediately available parallelism in the application.

and quotient of all pairs of spectral images. Each pair is considered only once, e.g. if $I_a - I_b$ is calculated, $I_b - I_a$ is not. If the input image has n spectral bands, the operations produce $2n + 3\frac{n(n-1)}{2}$ data sets.

The features of each feature set are extracted from the data sets produced by the arithmetic operations by finding the 1st, 5th, 10th, 30th, 50th, 70th, 90th, 95th and 99th percentiles of the pixel values. Determining the percentiles requires the data sets to be sorted individually.

4.3 Parallelization Strategies

This section will discuss the parallelization and the OpenMP implementation of the algorithm described in Sec. 4.2. The image processing algorithm differs from traditional high performance computing applications, such as matrix multiplication and physics simulation by having a significantly smaller data set and shorter execution time. Thus, the parallelization overhead can not be neglected.

The algorithm has two main parts as illustrated in Fig. 4.3. The preprocessing and mask generation part is governed by data dependencies, while the arithmetic feature extraction has parallelism immediately available between the feature sets, but also within the individual sets.

Profiling a sequential implementation of the algorithm revealed that 95% of the execution time is spent in feature extraction. Thus, it is the target for parallelization.

To summarize the task parallelism illustrated in Fig. 4.3, five independent feature sets are computed, which each produce n or $n(n-1)/2$ data sets for which the features are extracted by finding certain percentiles in the data sets. This means that the processing required for each feature set differs significantly. The feature extraction within each feature set should, in theory, be possible to split into parallel and equally sized tasks. However, non-uniform memory latencies caused by the target architecture may cause the execution time of each such parallel task to differ. The term *task* is used throughout this study to denote separate pieces of work and should not be confused with the recently introduced OpenMP task construct. Scaling properties are discussed in Sec. 4.3.1 without considering architectural effects which are discussed in Sec. 4.3.2.

4.3.1 Scaling Properties

The running times of the feature sets differ by up to a factor of $(n-1)/2$ leading to load imbalance problems if different feature sets are run in parallel. This study therefore concentrates on extracting parallelism of each individual feature set.

As mentioned earlier, each feature set has n or $n(n-1)/2$ equally sized tasks immediately available, which can be run in parallel. But if n is less than the number of available processors $|P|$, in processor set P, more parallelism must be extracted from these tasks. This is also advantageous to reduce the imbalance slack for the feature sets containing $n(n-1)/2$ tasks, as this may not match a multiple of $|P|$.

Additional parallelism can be extracted by splitting data sets into subsets that can be computed independently and then recombined. This means adding an extra nested level of parallelism. The arithmetic operations of all feature sets have no inter-pixel dependencies, which means that the processing of spectral bands into data sets can be split without creating any subset border synchronization issues. The sorting involved in the percentile calculation can be done on each subset separately followed by a merge of the sorted subsets before the percentiles are found. This allows the arithmetic operations to scale further, but with the overhead of merging the sorted subsets. It should be noted that the execution time of sorting each subset decreases by $d \times log(d)$, where d is the number of pixels in the subset. The execution time of merging the sorted subsets increases proportionally with the number of subsets generated by the data set decomposition. This means that the amount of parallel work decreases and the sequential part increases with an increasing number of subsets. Thus, the gain of increasing parallelism is diminishing. In addition, the parallelization overhead, such as spawning threads and synchronization, may be significant at this level as the subsets are small.

The two levels of parallelism within each feature set, among data sets and among subsets, are denoted as l_0 and l_1 respectively. In the implementation, the parallelism at each level s_0 and s_1, can be adjusted independently, though the parallelism at l_0 is limited. The total number of subsets across all data sets w is given by $w = s_0 \times s_1$ and constitutes the total number of tasks in the application. Subset processing time is defined as the wall clock time spent performing arithmetic operations on the parts of the spectral band data that corresponds to the subset and time spent sorting the subset.

In order to avoid load imbalance, s_0 and s_1 should be determined such that w is equal to or slightly less than a $m \times |P|$, where m is a multiple of the number of available processors $|P|$. If w is slightly larger than $m \times P$, only one or a few processors will be involved in processing the last remaining subsets while the majority of processors are idle, causing a large slack. The slack can be reduced by increasing w. But as mentioned earlier, s_0 is limited by n or $n(n-1)/2$ and s_1 is limited by the merge sort overhead, which causes diminishing parallelization gain. Hence, determining s_0 and s_1 is a trade-off between load imbalance and parallelization overhead.

4.3.2 Non-uniform Memory Latency

The discussion in the previous section holds under the assumption that the execution time of equally sized tasks do not differ. This assumption will not hold for architectures with non-uniform memory latencies. Threads running on processors which have long memory latency will have longer subset processing times than threads with short memory latency.

In this application, all spectral bands of the image are loaded into memory sequentially and then processed in parallel. Assuming a first touch memory placement policy in a hierarchical memory system, all image data will be located in the part of main memory local to the processor loading in the images, e.g. in the local memory on the Uniboard processor board, in a Sun Fire architecture system. A thread running on a processor associated with a different branch of the memory hierarchy, e.g. a processor on a different Uniboard than the one holding the main memory containing the image data, will access all data through the global memory interconnect and therefore have a significantly longer memory latency. This is not easily solved through parallel loading of the spectral images due to the fact that the data set processing requires all combinations of spectral bands. Thus, the effective subset processing time depends on the location of the processor.

Combining this effect with the scaling properties means that even though the total number of subsets w matches the number of available processors, linear speedup can not be obtained. Consider a system with $|P|$ processors, where $P_l \subset P$ is the subset of processors having local memory access to the image data and $P_r \subset P$ is the subset of processors having remote memory access to the image data through global memory interconnect. The execution times of a subset on $p_i \in P_l$ and $p_j \in P_r$ are t_l and t_r respectively, where $t_r > t_l$.

In the case of uniform memory latency, where $P = P_l$ and $w = m \times |P_l|$, the total execution time is given by $T = m \times t_l$, ignoring the parallelization overhead. In the non-uniform case where $P = P_l \cup P_r$, T depends on the task scheduling. Consider the case where w equals the number of processors $|P|$. In this case, every processor will process one subset each. Thus the total execution time is given by $T = max(t_l, t_r) = t_r$, if the parallelization overhead is assumed to be negligible. The processors in P_l finish before the processors in P_r, but the final result is not available until all processors have finished processing their subset. In the case where $w = 2 \times |P_l| + |P_r|$, assuming dynamic scheduling, $T = max(2t_l, t_r)$ as the processors in P_l will finish two subsets. If $2t_l > t_r$ the remote memory access of P_r, will not influence T. This is illustrated in Fig. 4.4. As a consequence of these two cases, resolving load imbalance may not result in the speedup outlined in Sec. 4.3.1. This applies to scaling both the number of processors and subsets, as these are both parameters that influence the load imbalance. Increasing the number of processors, such that $w = 2 \times |P_l| + |P_{r1}|$ becomes $w = |P_l| + |P_{r2}|$, where $|P_{r2}| = |P_{r1}| + |P_l|$, results in $T = t_r$. Thereby the total execution time reduction is only $2 \times t_l - t_r$, and not t_l.

The effect of load imbalance due to non-uniform memory latency also decreases significantly when w becomes much larger than the number of processors. Then again, the amount of parallelism available in the application may be limited and comes at a high cost in terms of parallelization overhead. The optimum solution is a trade-off between parallelization overhead and load im-

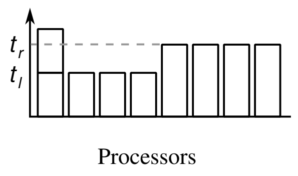


Figure 4.4: Different task execution times caused by non-uniform memory latency.

balance, where load imbalance is caused both by the algorithm itself, but also the architecture of the target execution platform. It should be noted that this is based on dynamic task scheduling. Static task scheduling will perform worse, due to varying execution times among the tasks.

4.4 OpenMP Implementation

The application was originally implemented in Matlab, and then ported to C using standard libraries only and without OpenMP parallelization in mind. All Matlab functions used in application were re-implemented using standard C-libraries to allow verification of the C-implementation by direct comparison to the results obtained using the original Matlab implementation. Subsequently, it was modified to meet the requirements for OpenMP parallelization.

In the sequential algorithm implementation, arithmetic feature extraction is implemented as a loop, where each iteration performs the arithmetic operation on a spectral band or pair of spectral bands, to form a new data set from which features are extracted. Unary arithmetic operators are applied to each individual spectral band in feature sets 1 and 2. These are implemented by a single loop through all the pixels in the spectral band. The feature sets 3, 4 and 5 are based on binary arithmetic operations between two spectral bands. First a list of pairs to be processed is generated. Then all pairs are processed using a loop. Similarly to the unary operations, the binary arithmetic operations are applied pixel by pixel in a single loop. Hence, the feature sets are implemented using two nested loops.

Three different parallel versions of the application have been implemented using OpenMP. One implementation uses nested parallelism, while the two other variants do not make use of nested parallelism.

4.4.1 Nested Implementation

The nested version exploits the two levels of nested parallelism discussed in Sec. 4.3.1. The first level of parallelism, l_0, consists of the aforementioned loop over the data sets, which is already present in the sequential implementation. This loop is parallelized using the OpenMP [81] `for` work sharing construct with dynamic scheduling, which is illustrated as the first thread fork in Fig. 4.5a.

Within each l_0-thread the data set is further split into subsets processed by another loop, which adds an extra loop to the implementation and forms the nested parallelism level l_1. This is illustrated as the second thread fork in Fig. 4.5a. Sorting each individual subset before they are merged as described in Sec. 4.3.1 requires complete control over the subset partitioning, which prevents the use of the existing pixel loop for this purpose. When all nested threads have finished and reached the implicit barrier of the OpenMP work sharing construct, the l_0-thread will continue by merging the subsets and extracting the features.

Since the assignment of the nested threads can not be managed dynamically as proposed in Duran et al. [27] and it is generally not possible to assign the same number of threads to each nested parallel section while having one thread per processor, one thread is created for every subset without considering the total number of threads. Balancing the load optimally may require one thread to process iterations from two different nested loops, which is not possible in OpenMP 2.5. Thus, creating more threads than processors will enable operating system schedulers capable of dynamic thread migration to load balance the processors. However, spawning more threads than processors may also induce a large scheduling overhead in the operating system.

4.4.2 Non-nested Implementation

To avoid relying on the operating system thread load balancing capabilities a non-nested version has been made. To flatten the two levels of parallelism, all $s_1 \times n$ or $s_1 \times n(n-1)/2$ subsets are enumerated and then processed in a single parallelized for loop, as shown in Fig. 4.5b. The number of threads is thereby completely independent of how many subsets the data sets are split into.

However, removing the two level hierarchy from the implementation and allowing subsets to be processed in any order means that it is no longer known when all subsets of each data set has been processed. Thus, merging can only take place when all subsets of all data sets have been processed. This is in contrast to the nested implementation where subsets are merged as soon as all nested threads belonging to a given data set have finished. Hence, merging

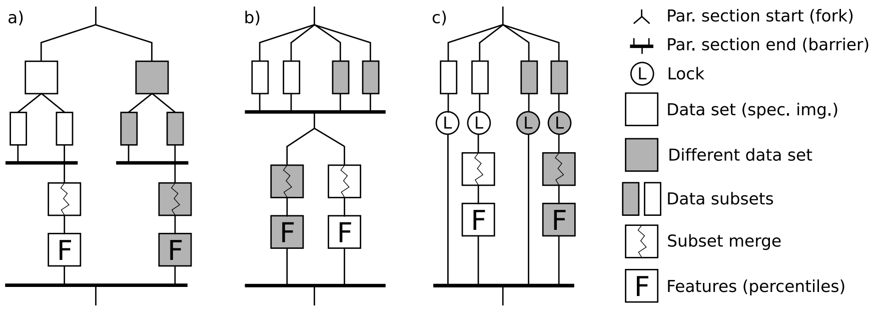


Figure 4.5: Thread utilization in the three OpenMP implementations: a) Nested parallelism, b) non-nested parallelism, c) improved non-nested parallelism using locks. Only two data sets are shown.

and percentile determination must be implemented as a second parallelized loop executed afterwards. This is illustrated by the second thread fork in Fig. 4.5b after all threads in the first thread fork have finished.

This implementation has the advantage of having only one thread per processor, but separating the subset processing and merging in two parallel sections has great influence on the application memory access pattern and thus also the cache performance.

The processed subsets of a given data set can to a great extent be found in the caches of the processors that processed them. But implementing merging as a second loop, Fig. 4.5b, means that there is no guarantee that any of these processors will perform the merging of the data set and exploit that one subset is located in its local cache. In the nested implementation illustrated in Fig.4.5a, however, the l_0-thread will be one of the nested threads and thus cache performance will be better.

Furthermore, by first processing all subsets and then merge them later, some of the subset data may be evicted from the cache due to limited cache capacity before they are merged. Merging the data set as soon as its subsets have finished, improves the probability of finding the subset data in the caches. However, this is very dependent on the cache size.

4.4.3 Improved Non-nested Implementation

To avoid the cache performance disadvantages of the initial non-nested implementation, a second improved non-nested version has been implemented. This implementation merges subsets as soon as all subsets of a data set have been completed and improves locality and thus cache performance.

Subsets are enumerated like in the first non-nested implementation, but instead of merging the data sets in a second loop, it is integrated into the subset processing. The thread finishing the last subset of a data set is responsible for merging all the subsets of the data set before it can process another subset as illustrated in Fig. 4.5c. This is implemented by assigning an OpenMP lock and a counting variable to each data set, illustrated by the “L” in Fig. 4.5c, which keeps track of how many subsets of the data set that have been processed.

A drawback of this version is that the subset processing times are not equal. Though dynamic scheduling is used for the work sharing construct, it can not be expected to counter this effect completely.

A similar implementation could be done using OpenMP tasks, which were introduced in OpenMP 3.0 [82]. Subset processing can be implemented using the new *task* construct. However, the OpenMP 3.0 specification does not support task data dependency notation. Thus the *taskwait* construct must be used to determine when the subset processing has finished and the subsets can be merged, which is similar to using the lock in the proposed implementation. The number of tasks to be processed is known, so the application will not generate tasks dynamically. In all, the net advantage of using OpenMP tasks is estimated to be slightly simpler code.

4.5 Results and Discussion

This section presents results obtained by running the nested and the two non-nested parallelized algorithm implementations using 16 processor cores on the test platform and compares these with the scalability issues discussed in Sec. 4.3.

4.5.1 Test Setup

In the presented results, the algorithm has been used to calculate all arithmetic feature sets of the input images. The input images are ten images, each containing nine spectral bands in a resolution of 777×776 pixels. The light intensity of each pixel is represented by a double precision floating point number.

The test platform used for producing the results in this study is a Sun Fire E6900. The machine has 48 UltraSPARC IV CPUs. Each processor has two cores running at 1200 MHz and has 8 MB L2 cache per core. The machine is running Solaris 10. Compilation has been done using the Sun C compiler version 5.9 patch 124867-01 using these options: `-fast -xarch=sparcvis2 -m32 -xopenmp=parallel -lm`.

The image loading time has been excluded from the measurements by loading all ten images, one by one, into main memory before they are processed. Warm up is done by processing all ten images once. To increase the accuracy of the measurements the presented results are based on the average execution times of ten or twenty consecutive runs of each feature set, where all ten images are processed. The number of runs is determined by the execution time of the particular test case. Using larger input images is not representative for the practical use of the algorithm and will lead to unrealistic results.

The average sequential execution times for feature sets 2 and 3 are 35 s and 127 s respectively, processing all ten multi-spectral images.

4.5.2 Parallel Efficiency

All tests have been limited to a maximum of 16 processor cores. Several parallelization approaches have been tested to investigate how the two levels of parallelism, l_0 and l_1, influence the parallel efficiency. It should be noted that even though all tests have 16 processors available, they may not all be utilized, depending on the number of threads in the particular test case. The nested version creates more than 16 threads in some tests. In order to prevent the threads to use more than 16 processor cores in these cases, a 16 core processor affinity set was specified using the `SUNW_MP_PROCBIND` environment variable for all runs with the nested version. This method may potentially lead to uneven load on the cores, but dynamic task scheduling counters this effect and no negative effects are observed in the results. Even though the main focus of the tests is parallel efficiency, scalability trends can also be extracted from the results of the nested version.

Figs. 4.6 and 4.7 illustrate the speedup obtained in feature sets 2 and 3 for the nested version by increasing the number of threads at l_0 with different data set partitioning at l_1. As mentioned in Sec. 4.4, one l_1-thread is created for each subset. The measurements of feature set 1, 4 and 5 are not significantly different from what can be observed in feature set 2 and 3, thus they are not shown.

Parallelization at l_0 does not impose any parallelization overhead except for thread creation overhead. However, parallelism is limited to nine l_0 threads in feature set 2. Linear relative speedup should be expected, when more threads can be created to utilize more processors. This can be observed in Fig. 4.6 for

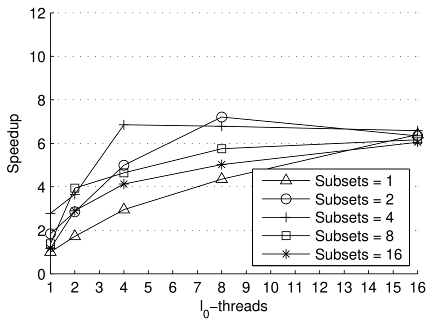


Figure 4.6: Speedups for the nested version of feature set 2 with 16 processors.

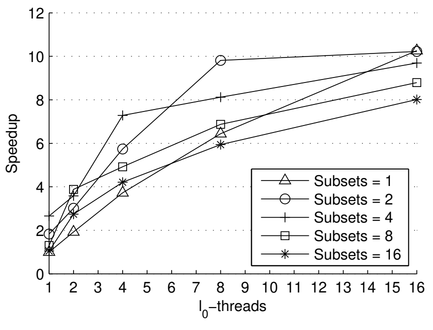


Figure 4.7: Speedups for the nested version of feature set 3 with 16 processors.

one to eight threads with no data set partitioning for feature set 2, which means $w = 9$. As discussed in Sec. 4.3.2, going from eight to 16 threads would double the theoretical speedup since load imbalance is improved. However, a speedup of only 1.5 is obtained, because $t_r > t_l$ meaning that data has to be fetched from a remote Uniboard leading to higher memory latency.

This effect has been confirmed by measuring the execution time of each l_0-thread, when running three and nine threads in parallel without any nested l_1-threads. The Sun Fire E6900 UltraSPARC IV Uniboards have four processors each with two cores, which means that if more than eight threads are used, some of them will be running on different processor boards. Fig. 4.8 and 4.9 show histograms of thread execution time using three and nine threads. It can be seen that using three threads, the histogram has a narrow range, while the histogram of nine threads is spread out. The lower part represents threads running on the board that holds the main memory containing the images, while the upper part is slow threads running on a different board. The ratio between a fast and a slow thread match the speedup obtained going from eight to 16 l_0-threads in Fig. 4.6.

As discussed in Sec. 4.3.1 parallelization at l_1 has sequential overhead. This can be observed in Figs. 4.6 and 4.7 when comparing the speedups of tests with one l_0-thread and increasing the number of l_1 threads. Even though more processors are utilized, the sequential merge eventually outweighs the parallelization speedup. Having more threads than processors also adds thread switching overhead as several threads share a single processor core. It can be observed on both

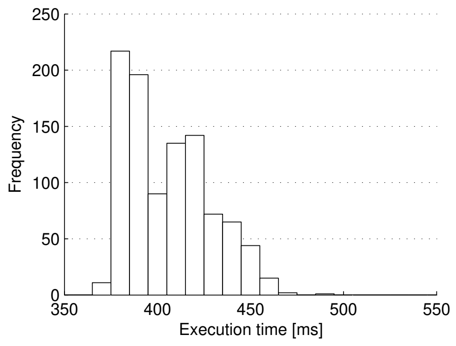


Figure 4.8: Thread execution time histogram when running 3 threads.

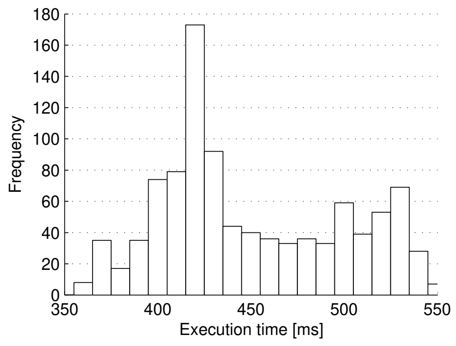


Figure 4.9: Thread execution time histogram when running 9 threads.

Figs. 4.6 and 4.7 that matching $s_0 \times s_1 = |P|$ leads to best results in general.

The effects observed in the results of feature set 2 can also be seen for feature set 3. However, the amount of parallelism available at l_0 is potentially 36 data sets. This leads to better parallel efficiency as less parallelism needs to be extracted at the l_1 level, where the sequential parts are limiting. The efficiency observed in feature set 2 is considered more realistic for real uses of this application, as only a subset of the features is typically needed [19].

The relation between work partitioning and the number of threads is removed in the non-nested versions. The number of threads is completely independent of the subset partitioning. Splitting the data sets creates more tasks that may lead to better work balancing among the threads. In Figs. 4.10 and 4.11, it can be observed that the improved non-nested version performs up to 24% better than the nested version. Comparing speedup of the two implementations, when the number of subsets increases, shows the overhead of having more threads than processors. With few threads, the two implementations have very similar performance, while the improved nested version performs significantly better with many subsets. The graphs representing the nested version in Figs. 4.10 and 4.11 show the best performing thread configuration with the corresponding number of subsets. However, it can be seen that when increasing the number of threads, parallelization overhead counters any speedup gained by increased parallelism. The effect of the parallelization overhead at l_1 can also be observed clearly for the improved non-nested implementation, as it is the cause of the decreasing speedup when having 8 and 16 subsets.

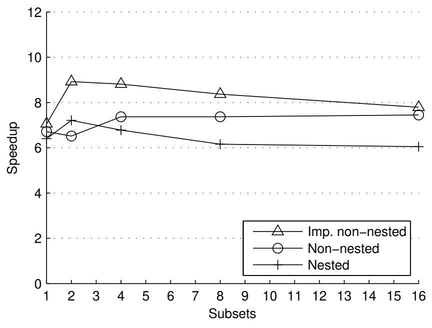


Figure 4.10: Speedups for all implementations of feature set 2 with 16 processors.

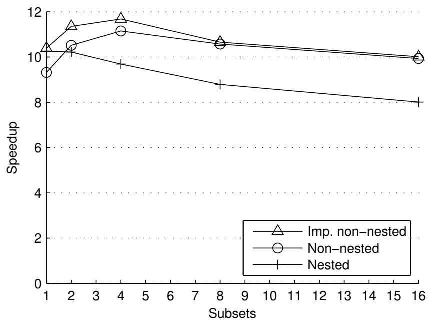


Figure 4.11: Speedups for all implementations of feature set 3 with 16 processors.

The performance of the initial non-nested implementation is difficult to predict due to its issues with regard to cache utilization. At best it can reach the performance of the improved version, but the real performance depends on the thread scheduling done at run-time and the OpenMP library implementation. When having only one or two subsets per data set, the impact of thread scheduling is large, as merging data on a processor without any of the subsets in its cache, will perform significantly worse than if it had a subset present in its cache. This effect will diminish as the number of subsets increases, since the subsets become smaller. The difference of accessing all data in other caches or main memory and having one small subset in the local cache becomes very small. This effect is shown in Figs. 4.10 and 4.11, where the speedup of the initial non-nested implementation approaches the improved one for larger numbers of subsets. The observed speedup of the initial non-nested implementation stresses the importance of considering cache utilization in parallel programming.

4.6 Summary

The case study of the image processing application has analyzed and discussed three different parallelization strategies for the OpenMP shared memory programming model. Each strategy puts an increasing amount of effort into parallelization to optimize for caches and non-uniform memory access latency. The study has revealed some of the challenges that might be encountered when parallelizing embedded application for many-core architectures.

The main challenges encountered are limited data set size and non-uniform memory access latency. Small data sets limit the directly exploitable parallelism due to the parallelization overhead. The non-uniform memory access latency found in many shared memory systems makes it difficult to manage parallel tasks efficiently. The threads of OpenMP are scheduled by the operating system. It is therefore difficult to detect the locations of data and treads in the system. This has been illustrated in this case study by the different execution times experienced by threads with and without local access to image data. Also, the importance of reuse of cached data by different threads has been illustrated by the three parallelization strategies. The operating system does not know the cache contents of the processors when it schedules new threads. It is therefore likely that the operating system makes poor scheduling decisions in this respect. Overall, the study indicates that better control over the location of data and threads can lead to better usage of the memory system in shared memory applications.

Chapter 5

Clupea: A Many-core Architecture with Configurable Support for Programming Models

Parallelization and on-chip communication are fundamental issues in NoC-based MPSoCs that must be addressed by the programming model. The amount of available parallelism and how it can be exploited varies from application to application. It is therefore virtually impossible to find a programming model that is suitable for all applications. The broad selection of programming models for both traditional multiprocessor applications and emerging many-core platforms attests to the complexity of this issue. On top of this is the issue of providing sufficient architectural support for a feasible implementation of the programming model.

This chapter presents the Clupea many-core architecture, which targets future embedded computing platforms with hundreds of cores. The central idea behind the Clupea architecture is to support a plethora of programming models and avoid the situation where the programming model is dictated by the hardware platform architecture. An overview of the architecture has previously been presented in [87].

The chapter is structured into the following subsections: Sec. 5.1 gives an introduction to the central ideas behind the Clupea architecture. Sec. 5.2 describes the system architecture and Sec. 5.3 describes the Clupea network interface, which is the central component of the architecture. Sec. 5.4 describes possible programming model implementations using the Clupea architecture. Sec. 5.5 describes a detailed implementation of the cache coherent shared memory programming model. The evaluation of the architecture is presented in Chap. 6.

5.1 Introduction

Following the current trends in MPSoC and CMP design, future many-core systems are likely to be tile-based architectures interconnected by a NoC, where each tile contains a processing core along with some form of local memory to reduce the impact of the increasing communication latency. Facing on-chip communication latencies of tens or hundreds of cycles in NoC-based interconnects, caching data locally is crucial for performance. While this overall architecture is conceptually trivial to envision, it remains a challenge to determine the programming model support needed in such architecture.

The implementation of a programming model is a hardware/software trade-off as previously illustrated in Fig. 2.6. In general, increasing the hardware support for the programming model leads to less software complexity and better performance. On the other hand, implementing the programming model mainly through software is much more flexible and can be adapted to the application requirements. The Clupea architecture addresses this trade-off while considering how to support a plethora of programming models in a flexible way that can be extended to support a heterogeneous set of processing cores. The architecture is based on the concepts of *configurable support for programming models* and *allocatable processing resources*, which will be introduced in the following subsections.

5.1.1 Configurable Programming Model Support

Hardware implementation of a range of different programming models can be a prohibitively expensive solution in terms of hardware resources and leads to a trade-off between few complex tiles and a larger number of simpler tiles. The aim of the Clupea architecture is to support application-specific programming models and also, to a great extend, support programming models not known at the platform design time. Additionally, to support architectures with a heterogeneous set of processor cores the programming model support can not rely solely on software implementation using the tile processing cores. Taking this into account, the Clupea architecture is based on configurable programming model support. This is offered by a specialized programmable network interface processor, NIP, in each tile. The NIP is tightly integrated with the cache controllers and the processor core. Basic hardware primitives in the NIP allow efficient interfacing to the NoC and management of the local memory resources in the tile. Support for new programming models is implemented by reprogramming the NIP of the tiles. Further details about the NIP architecture are given later in this chapter.

5.1.2 Allocatable Processing Resources

The expected increase in the number of processor cores in future many-core system will lead to a situation where the number of cores is equivalent to the number of tasks in the application. This calls for a new perspective on on-chip processing resource management. When the number of cores is lower than the number of application tasks, tasks must share processing resources. However, the vast number of cores in many-core architectures means that sharing is no longer a fundamental requirement. Tiles with processing cores can be viewed as allocatable resources, similar to memory, which are dedicated exclusively to an application as illustrated in Fig. 5.1. Based on this approach, there is a clear separation between tasks belonging to different applications that might be executing simultaneously. Tiles allocated by two different applications never have access to the same data and never communicate. The operating system can be simplified by giving the application direct control of the tiles when they have been allocated. Application tuning, such as cache optimization, is not impacted by interference between applications or unpredictable scheduling of tasks as it was observed in Chap. 4. However, the vital advantage of this approach is that there is no need to consider interoperability between programming models used by applications executing simultaneously. The programming model used in tiles allocated by one application does not need to consider the programming model used in tiles allocated by other applications. This property is essential for the Clupea architecture.

Tiles can be configured on allocation to support any application-specific programming model requested by the application, e.g., one application may use a message passing programming model while another uses cache coherent shared memory. Furthermore, since tiles allocated by different applications will never communicate directly, separation of applications for improved system security

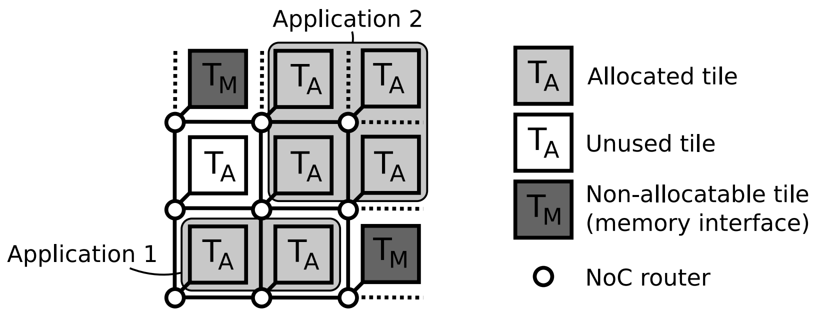


Figure 5.1: Tiles are allocated by applications, which gain exclusive access to their allocated tiles. Applications 1 and 2 use two distinct sets of allocatable tiles. Memory interface tiles are shared.

can naturally be done at the tile level. This thesis focuses on programming model support and the memory system. Therefore, system security aspects will not be considered further.

5.2 System Architecture

The Clupea system architecture basically consists of tiles and a NoC interconnect that interconnects all tiles as illustrated in Fig. 5.2. A tile-oriented operating system is assumed to handle tile allocation. When tiles are allocated by an application, the operating system will configure and initialize the tiles according to the need of the application before they can be used. The application is not allowed to modify the tile configuration itself. The following subsections will describe the tile architecture, the memory system, and the NoC architecture.

5.2.1 Tile Architecture

The system consists of tiles of different types. The main type of tiles is *processor tiles*, which are allocatable tiles used for executing applications. The internal architecture of a basic processor tile is centered around the Clupea NIP as illustrated in Fig. 5.2. The in-tile memories, i.e., caches and scratch-pad memory, are assumed to have a combined size which is equivalent to the private tile memories in recent tile-based architectures [53, 46]. The NIP interfaces all tile components in addition to the NoC as described below. The NIP architecture is discussed in Sec. 5.3.

- **Processor core:** The Clupea architecture has no restrictions on which type of processor core that may be used in a tile. However, the application allocating the tile must use a programming model that is compatible with the capabilities of the core. Specialized processor tiles may have computational accelerators attached or replacing the general processor core. Exploration of processor cores and accelerator architectures are not considered in this thesis. Thus, a general purpose in-order 64-bit processor core with co-processor interface that supports co-processor generated interrupts is assumed.

- **Scratch-pad memory:** The scratch-pad memory, SPM, is a private memory mapped into the address space of the local processor core and can not be referenced directly be other tiles. Both the processor core and the network interface have arbitrated direct access to the SPM.

- **Instruction and data caches:** The instruction and data caches are private caches, which may cache both global memory and the local SPM. The caches are partially managed by dedicated hardware cache controllers, which are capable of handling accesses to addresses in the private SPM and accesses to global addresses that results in cache hits only. Cache misses are handled by the NIP, which is informed about the event in a fashion inspired by previous works of Horowitz et al. [45] and Zeffer et al. [109]. Only cache misses to global addresses require consistency and coherence considerations. Thus by servicing these using the NIP, the memory architecture can be altered by simply reprogramming the NIP while cache hits and SPM caching are managed efficiently by the hardware cache controller. The NIP has full access to read and modify the internals of the cache, such as cache line data, tags and status bits. On a cache miss, the caches block their processor core interface while the NIP updates the cache contents.

5.2.2 Memory System

The memory system is based on the in-tile caches, the SPM, and special *memory interface tiles*, which provide access to off-chip main memory. The number of memory interface tiles is limited by the pin count of the chip. Memory interface tiles can not be allocated, as it can not be assumed that the system has enough

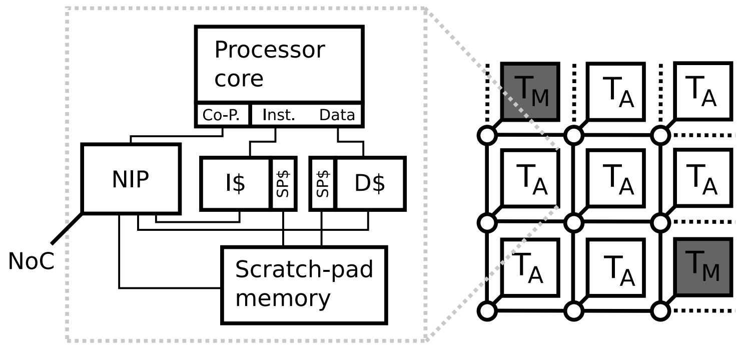


Figure 5.2: Internal tile architecture of processor tile consisting of a processor core with instruction and data caches (I$, D$) and scratch-pad memory (SPM).

memory interface tiles to provide one exclusively for each application. Main memory is shared and mapped into the address space of all tiles, but how it is accessed is determined by the programming model support offered by the NIP configuration. The NIP implements the lower levels of the programming model by transforming memory accesses and messages into packets which are transferred across the NoC as illustrated in Fig. 5.3.

Servicing both global cache misses and all inter-tile communication in the NIP means that the memory system is configurable for each tile independently of the configuration of other tiles. Only the configurations of tiles allocated to the same application must be compatible. The NIP controls where missing cache lines are fetched from and thus implements both the memory consistency model and the cache coherence protocol. For accesses to main memory the NIP must explicitly fetch data from the memory interface tiles and insert it into the local cache or SPM. This may happen implicitly on cache misses or when requested explicitly by the processor core depending on the configuration of the NIP. Through this configurability, the NIP can support a plethora of programming models as shown in Fig. 5.3, including future programming models.

Virtual memory is assumed to be managed internally in the processor core and the caches are physically tagged to avoid managing virtual address translation in the NIP.

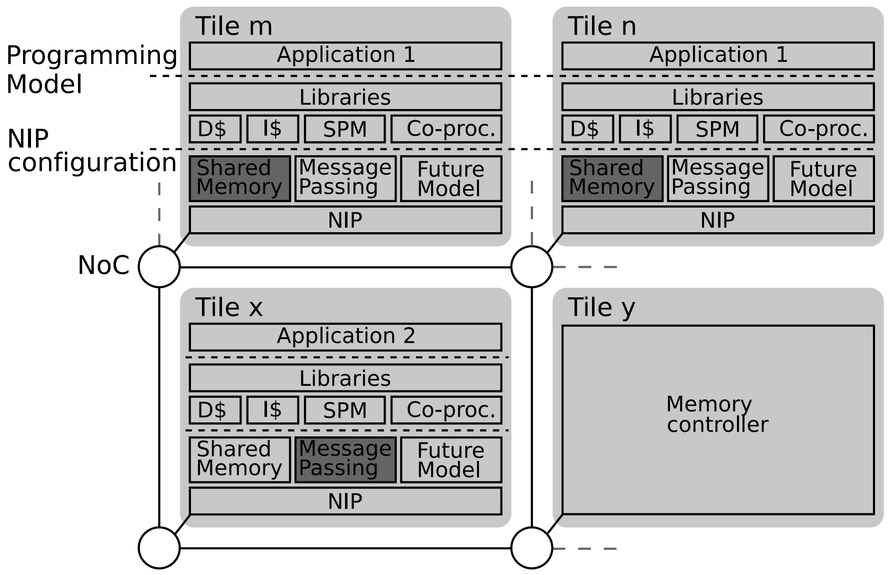


Figure 5.3: NIP programming model configuration. The tiles allocated by Application 1 have their NIPs programmed for shared memory support. At the same time, the tiles allocated by Application 2 are programmed for message passing programming model support through the NIP. Main memory is accessed through non-allocatable memory interface tiles.

5.2.3 NoC Architecture

The NoC interfacing is a central part of the NIP architecture and thus the NoC architecture has great impact on the NIP architecture. Many different NoC architectures have been proposed with different characteristics [23, 14, 33]. The Clupea architecture attempts to be as NoC architecture independent as possible. To reduce the NoC router complexity and implementation costs, only very basic services from the NoC are assumed. The NoC routers are assumed to be simple packet forwarding units without support for resource reservations or virtual channels. The Clupea architecture can be easily adapted to take advantage of NoCs that provide more elaborate services.

The NoC used in the Clupea architecture is a connection-less packet-switched source-routed network that supports two packets sizes, one which can be used for requests and one that can contain an entire cache line as payload. The packets are serialized and transferred as a stream of phits. The topology of the NoC does not influence the Clupea architecture, but for the purpose of illustration, a two dimensional mesh is assumed.

Flow Control

Message dependent deadlocks are major issues in NoCs [1], which can be avoided by careful buffer sizing, using multiple virtual networks or end-to-end flow control. The former two approaches require design time knowledge about the exact communication pattern generated by the applications and are thereby incompatible with the aim of a flexible many-core architecture. Instead, deadlocks must be solved through end-to-end flow control implemented in the NIP.

The flow control scheme used in the Clupea architecture is based on packet acknowledgements and time-outs. Packets may be dropped to reduce network congestion and prevent buffer overflows in the NIPs. This scheme has minimal best case latency. In contrast, credit-based flow control has an inherent issue of distributing credits to potential senders and the size of the NoC buffer for incoming packets must match the number of credits. The packet acknowledgement scheme implies no restrictions on the NoC buffers, however, packet loss must be considered in the communication protocol or handled by the NIP. For instance, this can be done in the Clupea NIP using a *retransmission buffer* in the SPM to store copies of all outgoing packets until they have been acknowledged. Furthermore, an added benefit of this flow control scheme is integrated support for fault-tolerance, where faulty packets are dropped by the NoC.

Packet field	Description
Route	The encoded route to the destination generated by the sending NIP.
Destination	The identity of the destination tile.
Source	The identity of the source tile.
Reply	The identity of a third party tile, which may be the destination of a subsequent reply message
Type	Message type identification.
Address	A reference to a global address.
Sequence number	Packet flow control sequence number.
Data payload	Optional payload data that has the size of a cache line.

Table 5.1: NoC packet layout.

Packet Layout

The packet layout is essential for fast packet processing in the NIP and proper support for the flow control scheme. The packet fields are listed in Tab. 5.1. The NoC routers only make use of the encoded route, the remaining fields are used by the NIP at the destination. Thus, the use of the remaining fields is optional and they may be used for other purposes than the listed descriptions. It is assumed that all caches use the same cache line size.

5.3 Network Interface Processor Architecture

The NIP is the most essential part of the Clupea architecture as it is the key component for configurable support for programming models. The NIP architecture must therefore provide great flexibility while it must allow efficient data transfers between caches, SPM and the NoC at the same time. On top of this, the NIP should have low hardware complexity. Using an extra general purpose processor core as a dedicated communication processor is overly expensive and lacks interfaces to manage the caches and access the NoC. Efficient access to these interfaces is essential for providing a low NIP latency. To address this, the NIP architecture is based on a specialized domain-specific processor pipeline to provide flexibility while maintaining a reasonable hardware implementation cost.

The NIP pipeline supports a limited instruction set, which allows execution of basic control algorithms and is highly optimized for moving data between the NoC, the SPM and the caches. To facilitate these data transfers, the pipeline has integrated interfaces to the caches, SPM, and NoC. Data can be moved directly between these interfaces using special instructions. Based on this approach, programming model support can be configured when the tile is allocated by simply programming the NIP. The overall architecture is illustrated in Fig. 5.4 and will be discussed in more detail in the following subsections.

5.3.1 Processor Pipeline

The basic idea behind the NIP pipeline is to avoid moving data through registers as much as possible and only support the basic arithmetic and logic operations that are needed to perform address manipulation and manage simple data structures. The pipeline supports two word sizes to reduce the hardware costs. The basic word size is 24 bits and allows the entire SPM to be mapped into the address space of the NIP. The arithmetic/logical unit, ALU, in Fig. 5.4 supports a range of operations on basic words. The extended word size is 64 bits and is intended to be used for global addresses. The "E-logic" unit supports only simple shifting, bit masking and logical operations on extended words. The pipeline includes a set of general purpose registers of both word sizes which are labelled "GP regs" in Fig. 5.4. Special instructions allow data to be moved between the basic and extended registers. In addition to these, a number of special purpose registers, "SP regs", are associated with the caches, processor core co-processor interface and the NoC. These will be described later.

The NIP code is stored in a dedicated instruction memory, which is populated by the operating system when the tile is configured on allocation. This ensures that NIP instructions can be fetched in a single cycle and avoids contention for accessing the SPM. The instruction memory is assumed to hold a few thousands of NIP instructions. The NIP pipeline supports four hardware threads with individual program counters and private general purpose registers. The special purpose registers are globally shared along with a small number of general purpose registers that allow data to passed between the threads. The NIP architecture uses special hardware semaphores, called lock variables, for synchronization and thread scheduling. The hardware threads and lock variables will be further discussed later.

The NIP supports four types of instructions: i) Basic arithmetic and logic instructions, ii) control instructions such as branch and jump, iii) load and store operations that can access the SPM, and iv) special instructions that include

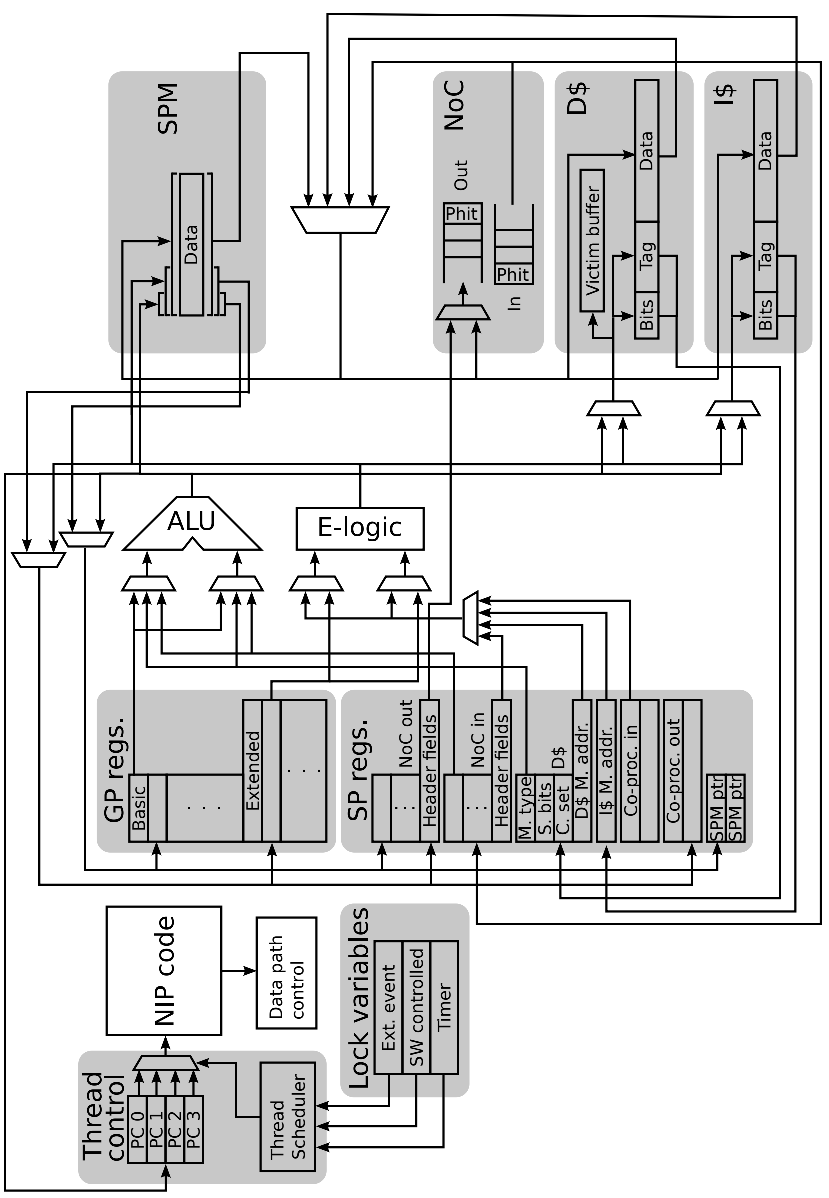


Figure 5.4: NIP architecture overview. Duplicate register files for thread private registers are omitted.

instructions for manipulating caches, data movement, NoC packet header encoding and thread synchronization. The former three types of instructions may operate on general purpose registers and special purpose registers. However, only logical instructions and instructions for moving data to and from basic word registers can operate on extended registers. The SPM allows both basic and extended words to be loaded and stored. The purpose of the specialized instructions will be discussed in the following subsections.

5.3.2 Interfaces

The interfaces between the NIP and the NoC and internal tile components are different by nature and they are therefore integrated into the NIP architecture in different ways. The interfaces are generally controlled through special NIP instructions and special purpose registers.

The NoC interface of the NIP consists of two FIFO buffers for incoming and outgoing packets, which provide direct access to the NoC. The width of the FIFO buffer interface matches the phit width of the NoC. Combined with phit sized access to the NIP interfaces this ensures that data can be moved efficiently between the NoC and the internal tile components. However, this also means that phits can not be loaded directly into NIP registers. Instead, the packet header of incoming packets is explicitly decoded into a set of special purpose registers using a single header decode instruction. The NIP architecture has a special purpose register for each of the NoC packet header fields listed previously in Tab. 5.1 to ensure fast packet decoding. Similarly, the header of outgoing packets is constructed by setting up the header fields in special purpose registers and encoding them into a header phit which is inserted in the outgoing NoC FIFO using a single instruction. Packet payload phits are explicitly moved to or from the caches or the SPM to the NoC buffers using special data movement instructions, which will be discussed later in Sec. 5.3.5.

The NoC is assumed to be source routed and based on a *route table* set up in the SPM by the operating system. The packet route is loaded into the NoC buffer by the NIP. There is no hardware support for the end-to-end flow control scheme. Flow control is implemented through NIP code which generates acknowledgement messages and stores copies of all outgoing packets in the SPM until they have been acknowledged by the receiver. This approach allows the buffers to be resized to match the needs of the application and thus potentially saves hardware resources compared to hardware buffers.

The cache interfaces are shared with the processor core, which is blocked on cache misses. The NIP has additional access to manipulate the internal bookkeeping information in the caches to allow it to handle cache misses. On data cache miss, the data cache special purpose registers, shown in Fig. 5.4, are loaded with the cache miss address, the cache miss type, and the status bits and the cache way of the cache line, if it is already present in the cache, i.e., in case of insufficient write permissions. Using special instructions, the NIP can manipulate cache status bits, update tags, control the cache victim buffer and transfer cache lines directly between the caches and the NoC buffers. Full control over the victim buffer means that the cache replacement policy is implemented by the NIP. The instruction cache has a slightly simpler interface as instructions are typically read-only. Thus, the instruction cache only loads the miss address into the NIP special purpose register.

The processor core interface contains two pairs of extended special purpose registers for transferring data between the NIP and the co-processor interface of the processor core. These allow extended word sized data and pointers to be passed directly between the NIP and the processor core. Transfers from the NIP to the processor core cause an interrupt in the processor core to allow the processor core to detect new data in the co-processor interface registers.

5.3.3 Hardware Threads

The NIP has support for four hardware threads as mentioned earlier. The reason for this aggressive approach is the fact that four components may request services from the NIP at any given time.

- **Instruction cache:** Instruction cache misses to instructions stored on global addresses. The NIP must fetch the missing instructions and insert them into the instruction cache.

- **Data cache:** Data cache misses to data stored on global addresses. The NIP must locate the missing cache line or upgrade the cache line permissions to match the memory operation.

- **Processor core:** The local processor core can generate explicit requests to the NIP through its co-processor interface.

- **NoC:** Any incoming packet must be processed by the NIP to determine its message.

Using four hardware thread contexts, large context switching overheads can be avoided by dedicating a thread to handle a particular source or interface. The

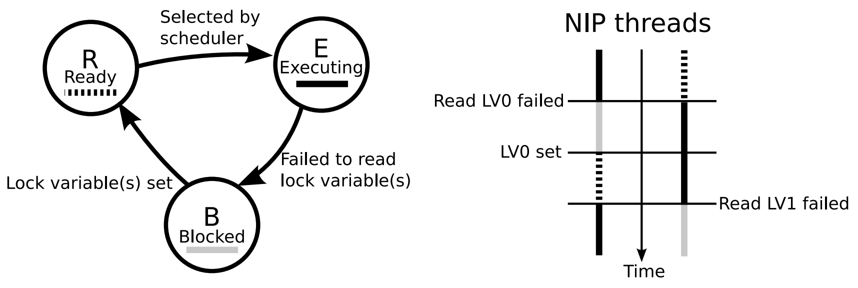


Figure 5.5: NIP thread states (left) and example on execution of two threads (right).

hardware thread context allows the NIP to switch from processing, for instance, a data cache miss to processing an incoming NoC packet in a few cycles.

Each thread has its own program counter and private general purpose register file. Thread scheduling is non-preemptive and done in hardware. Thread context switching incurs only the overhead caused by starting up the NIP pipeline with a new stream of instructions. The thread execution model is based on three states: Executing, blocked or ready as shown in Fig 5.5. Only one thread can be executing at any time and this thread will continue execution until it is blocked by an access to a *lock variable*. These will be described shortly. Blocked threads can not be scheduled for execution until their unblock conditions are met and they enter the ready state. The next thread that is scheduled for execution is selected among the threads in the ready state. The main advantage of this scheduling scheme is the fact that it ensures that threads are never blocked unexpectedly and thus the need for synchronization between threads is reduced to a minimum. For example, there is no need to coordinate accesses to the NoC buffer to avoid interference between threads, since the threads can not be pre-empted.

5.3.4 Lock Variables

External requests to the NIP and internal synchronization between NIP threads are based on lock variables in the NIP architecture. Lock variables are specialized hardware locks that can be set, read and cleared. Each variable is either associated with an external event or controlled by the NIP threads. Examples of external events that are associated with lock variables are cache misses in the data cache and the arrival of a new packet in the incoming NoC packet buffer.

The external event lock variables are set by hardware whenever the associated event occurs. Software controlled lock variables are used for synchronization among the NIP threads and are set using a special NIP instruction.

A NIP thread can read lock variables through polling and blocking instructions, which may specify one or more lock variables to be read in parallel. Blocking access to a lock variable which has not been set will block the NIP thread and put it into the blocked state as illustrated in Fig. 5.5. The thread is unblocked and scheduled for execution when all the accessed lock variables have been set. Thus, the lock variables offer an efficient mechanism for suspending NIP threads until a certain set of events have occurred. For example, a thread responsible for handling data cache misses can block itself until the next data cache miss occurs. Most lock variables are unset by the NIP thread, while a few are unset by hardware, such as the lock variable indicating a full outgoing NoC buffer.

5.3.5 Data Movement

Efficient primitives for moving data between interfaces is essential for the NIP architecture. Since data only needs to be moved rather than processed, there is no need to pass the data through the main data path of the NIP. Instead, the NIP architecture supports direct data movement for this purpose.

Phit-sized data can be transferred directly between the interfaces using special NIP instructions in a single cycle. The supported data sources and destinations are: The NoC buffers, cache lines in the instruction and data caches, and the SPM, which all provide phit sized data access as illustrated in Fig. 5.4. Packet payload phits can be moved directly into cache lines and the NoC packet route header phit can be loaded directly from the route table in the SPM to the outgoing NoC buffer. The source and destination addresses of the SPM handle is controlled through two special purpose pointer registers. Phit-sized data movement is only possible between the NoC buffers and other interfaces to minimize the hardware costs.

5.3.6 Configuration Example

To understand the NIP architecture and programming better, the following example in Alg. 1 and 2 illustrate the pseudo-codes for two NIP threads that are involved in handling data cache misses. The threads use lock variables to wait for a new cache miss and the incoming reply packet from the memory interface tile. Data movement instructions are used for copying the cache line directly into the cache. The first thread, Alg. 1, generates a NoC message to the memory interface tile and the other thread, Alg. 2, receives incoming NoC messages and inserts the missing cache line into the data cache.

Algorithm 1 NIP thread pseudo-code example for handling data cache misses. Write back of victim cache line and retransmission buffering are omitted.

```
{Block thread until data cache miss}
start: wait(Data cache miss lock variable)
{Set NoC packet header fields}
set msg_header_out_destination = memory_tile_id;
set msg_header_out_address = data_cache_miss_address;
set msg_header_out_type = memory_read;
set msg_header_out_reply_id = tile_id;
{Load route into outgoing NoC buffer}
load_phit(memory_tile_id) → NoC_out_buffer;
{Create packet header. The packet has no payload}
noc_header_assemble → NoC_out_buffer;
jump → start
```

5.4 Support for Programming Models

The generic NIP architecture does not provide support for any default programming model and does not enforce any basic programming model. The NIP must be programmed before the tile can be used. This section will briefly discuss how a few example programming models can be supported on the Clupea architecture. A more thorough discussion on implementation of one programming model, cache coherent shared memory, is given Sec. 5.5.

5.4.1 Shared Memory Models

Non-coherent shared memory can be supported directly by the NIP architecture. Cache misses are handled by the NIP using the lock variables associated with the caches. The NIP sends a request packet to the memory interface tile to fetch the missing cache line as it was previously described in Alg. 1 and 2. In this configuration the NIP provides transparent access to main memory through the caches and thus provides a shared memory programming model to the application executing on the allocated processor tiles.

Algorithm 2 NIP thread pseudo-code example for handling incoming NoC messages.

```
{Block thread until a NoC message is received}
start: wait(Incoming NoC buffer not empty lock variable.)
noc_header_disassemble ← NoC_in_buffer;
{Check message type}
if msg_header_in_type == memory_reply then
  {Move packet payload, the cache line, into the data cache phit by phit.}
  move_phit NoC_in_buffer → data_cache_0;
  ⋮
  move_phit NoC_in_buffer → data_cache_n;
else
  {Other message type}
end if
jump → start
```

A simple implementation of cache coherent shared memory is to emulate a snooping cache coherence protocol by sending cache lines requests to all allocated tiles and the memory interface tile. If any of the allocated tiles reply with a cache line it can be assumed to be the most recent copy. Otherwise, the cache line will be provided by the memory interface tile. Due to the pseudo-broadcasting, this approach scales poorly as the number of tiles increases. Other more sophisticated alternatives scale better. One example of these will be described in Sec. 5.5.

Support for address space partitioning and thread private data can also be provided by the NIP. Based on the cache miss address the NIP may perform different actions depending on which memory partition it belongs to. This concept is known from thread private and thread shared variables in OpenMP [81, 82] and partitioned global address space programming models [28, 21]. For instance, memory that is thread private requires no coherence, while global memory that is shared by multiple threads requires coherence. In this case the NIP can handle the two situations differently and thereby optimize cache misses on accesses to private data rather than treating all cache misses as coherent. This model must be supported by the programming language or the compiler.

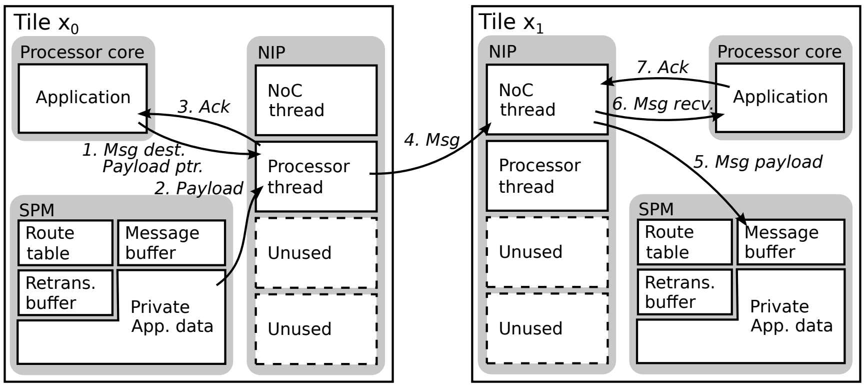


Figure 5.6: Message passing using the NIP as a message passing co-processor. Two hardware threads are used to support message passing. Caches are omitted for clarity. The unused NIP threads may be used to support the caches.

5.4.2 Message Passing Models

Simple message passing between allocated tiles can be supported using the direct processor core/NIP interface, which makes the NIP appear as a message passing co-processor to the processor core. Messages are send by the processor core by passing a pointer to the message payload in either the data cache or the SPM to the NIP through co-processor interface registers along with additional message header information such as destination tile and messages size. Based on this information the NIP constructs and sends the messages. Completion of the message transfer is acknowledged to the processor core by the NIP through the co-processor interface to indicate safe reuse of the payload memory space. These steps are illustrated in Fig. 5.6.

Incoming messages are stored in the SPM by the NIP in a pre-allocated message buffer. The processor core is notified of the message arrival through the co-processor interface, which interrupts the processor core. The processor core acknowledges the messages to the NIP through the co-processor interface to signal safe reuse of the message buffer to the NIP. Message reordering and handling of outstanding asynchronous send and receive operations is implemented by higher level software layers running on the processor core.

Additionally, it is possible to support single word messages where the message payload is passed directly between the NIP and the processor core through the co-processor interface registers. This approach will support very low latency message transfers suitable for fast inter-tile signaling as buffer management is completely avoided. Support for single word messages is also possible combined with a shared memory programming model.

DMA transfers [53] can also be supported by the Clupea architecture. Passing a pointer from the processor core to the NIP through the co-processor interface, the NIP takes care of copying data from main memory to the local SPM by sending one or a series of memory requests to the memory interface tile. Similarly, data transfers from the local SPM to main memory can be handled by the NIP.

Transfers between tiles can also be supported by mapping the SPMs into a global shared memory map. In this case the NIP would determine the destination tile based on the memory mapped SPM address and support the global memory space view.

5.5 Implementing Cache Coherent Shared Memory

To evaluate the capabilities and the configurable support for programming model of the Clupea architecture, this section will discuss implementation of cache coherent shared memory in more detail. This implementation will be used in Chap. 6 as basis for performance modeling and evaluation of the Clupea architecture. Other programming models could likewise be implemented on the Clupea architecture, however, this work will only consider cache coherent shared memory in detail. The shared memory model is used in a wide range of systems and is the foundation of many programming models.

5.5.1 Introduction

The Clupea architecture targets many-core systems and therefore requires a scalable implementation of cache coherent shared memory. Previous large-scale cache coherent shared memory multiprocessors are multichip multiprocessors based on processor nodes with local caches [64, 59, 91, 6, 62], a portion of main memory and a protocol processor that implements a directory-based cache coherence protocol. The Clupea architecture shares the basic architecture with these systems, i.e. distributed processors with private caches and a non-broadcasting interconnect, and thus it is obvious to use those of the implementation concepts known from these systems that can be applied in the Clupea architecture.

Implementation Challenges

The memory system of the Clupea many-core architecture is fundamentally different from the memory systems of previous multichip multiprocessors in two ways:

1. Main memory is off-chip and accessed through memory interface tiles as discussed in Sec. 5.2.2. The number of memory interface tiles is limited by the pin count of the chip and will lead to a high processor tile per memory interface tile ratio.

2. Accessing off-chip main memory is always orders of magnitude slower than accessing on-chip caches and SPMs. This is true for both caches and SPMs in both the local tile and in remote tiles.

Due to these conditions it is crucial for the Clupea cache coherent shared memory implementation to keep any important data on-chip [16]. Off-chip memory should be accessed as little as possible to avoid the latency and minimize the memory contention caused by the high number of processor cores per memory interface.

The previous multichip multiprocessors [59, 91, 6] are based on distributed directories implemented by complex protocol processors that store the directory information in the local portion of main memory. This approach is unsuitable for on-chip architectures since accessing directory information in main memory is too expensive for resolving cache misses that may only require an on-chip cache-to-cache transfer. CMP architectures with shared last-level caches address this issue by maintaining directory information at the last-level cache. However, it is a prerequisite of this approach that the last-level cache must include copies of cache lines of all other caches. Thus, scarce on-chip memory resources are spent on the duplicate cache lines.

The Clupea architecture has no shared last-level caches and only limited on-chip memory resources, so it is not possible to hold all directory information on-chip. Furthermore, the processing capabilities of the Clupea NIP can not match the previous protocol processor chips and it is therefore infeasible to have a directory in every tile. Hence, the main challenge of implementing cache coherent shared memory on the Clupea architecture is to implement coherence directories with the available on-chip resources, i.e. allocatable processor tiles.

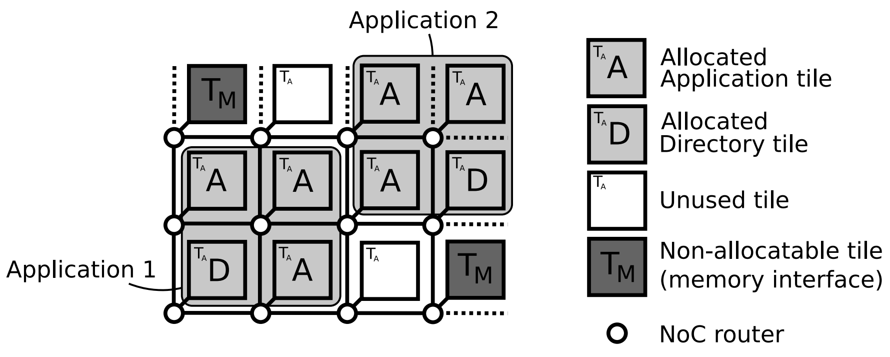


Figure 5.7: Allocated tiles for cache coherent shared memory applications. Each application allocates a number of tiles to become either application or DTs.

Adding dedicated hardware for this purpose will increase hardware costs and conflict with the aim of having flexible programming model support.

Implementation Overview

The proposed cache coherent shared memory implementation for the Clupea architecture is based on a directory-based cache coherence protocol. Using the limited memory and processing resources of each tile to implement a distributed directory in every tile allocated by the application is expensive. Instead, directories are implemented using ordinary allocatable processor tiles, which are dedicated to implement directories. The directories serve as coordination points and hold no cache line copies locally. This means a clear separation between the configurations of processor tiles that execute the application and processor tiles that are used as directories. These will be referred to as *application tiles*, AT, and *directory tiles*, DT, respectively. An application that assumes cache coherent shared memory needs to allocate both ATs and DTs for programming model support as illustrated in Fig. 5.7. The configuration of ATs and DTs is assumed done by the operating system when the application is loaded.

The pressure on the DTs can be adjusted to fit the needs of the application by changing the ATs per DT ratio. Memory intensive applications which generate many cache misses may benefit from having more DTs, while the number of DTs can be reduced for less demanding applications.

The cache coherence protocol is inspired by the write-invalidate MOESI SMP directory protocol implementation in the GEMS [71] multiprocessor simulator.

The protocol has been reimplemented and modified to suit the Clupea architecture. The protocol states have been reduced to Modified, Shared, Owned and Invalid to simplify the protocol. Directories forward cache lines between on-chip caches whenever possible instead of accessing main memory. Furthermore, main memory write acknowledgement has been introduced to avoid race-conditions caused by the fact that the directories are not co-located with memory interfaces.

The cache coherence protocol implementation consists of two main parts: i) the local NIP configuration in the ATs and ii) the configuration of DTs. The overall behavior of these two parts are illustrated by the simplified state diagrams in Fig. 5.8. Transition states, acknowledgement messages and messages related to cache line write-back are omitted for clarity.

Cache lines in the data caches of the ATs contain status bits that determine the cache line state. The cache line state is either modified, shared or invalid: Modified cache lines have read-write permission, shared cache lines are read-only and invalid cache lines do not hold valid data. The cache controller checks the cache line permissions for every cache access and generates a cache miss event in the NIP if the referenced cache line is not present in the cache or if the processor core attempts to write to a shared cache line. Depending on the cache miss type, the NIP sends a request for either a read-only or a writable copy of the cache line to the DT. A shared read-only copy is requested by a "GETS" request and an exclusive read-write copy is requested by a "GETX" request as illustrated in Fig. 5.8a. Cache lines are invalidated either as a part of a cache line replacement or by request from the DT. The DT can request a cache line copy to be send to another AT using a "Fwd_GETS" or "Fwd_GETX" message. "Fwd_GETX" and "Inv" requests from the DT invalidates the cache line.

The DTs keep track of the cache copies using a list of current sharers and four states: Modified, shared, owned and invalid. Cache lines in the modified state reside exclusively in one cache and this cache has read-write permission. Shared cache lines are read-only and may reside in several caches. Cache lines in the owned state are similar to shared cache lines, but have been modified previously. Thus, the last copy of this cache line must be written back to main memory when it is evicted from the AT data cache. Invalid cache lines are not currently cached in any AT in the system. The actions of the directory depend on both the directory state of the cache line and the request as illustrated in Fig. 5.8b.

Cache misses generally involve a series of messages and at least three tiles: The local AT, the DT and a remote tile that may be either another AT or a memory interface tile. For instance, a data cache read miss in an AT causes

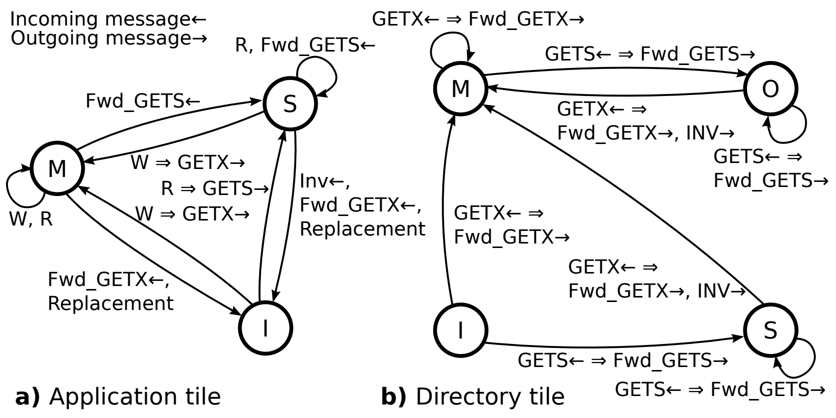


Figure 5.8: Overview of cache line states and messages in the cache coherence protocol: a) The local part of the protocol in the ATs, and b) the directory protocol. Messages and state transitions in the directory related to cache line write-back are omitted.

the NIP to send a "GETS" request to a DT. The DT sends a "Fwd_GETS" request to either a remote AT which holds a copy of the requested cache line or the memory interface tile if no copies exist on-chip. The remote AT, or memory interface tile, replies by sending a cache line copy directly to the original requesting AT. Meanwhile, the NIP in the local AT determines the victim cache line and moves it to the victim buffer. Following this, the NIP sends an eviction request to the DT. Depending on the state of the victim cache line it is either deleted or written back to main memory. The eviction process is not shown in Fig. 5.8.

The cache coherence protocol assumes reliable communication, which is not provided by the NoC. Thus, all messages must be stored in a retransmission buffer and the receiver must acknowledge reception of all messages.

The following two subsections will cover the implementation details of the ATs and DTs.

5.5.2 Application Tile Configuration

The configuration of the NIP in the ATs implements the AT part of the cache coherence protocol in Fig. 5.8a to provide support for the cache coherent shared memory programming for the application executing on the processor cores.

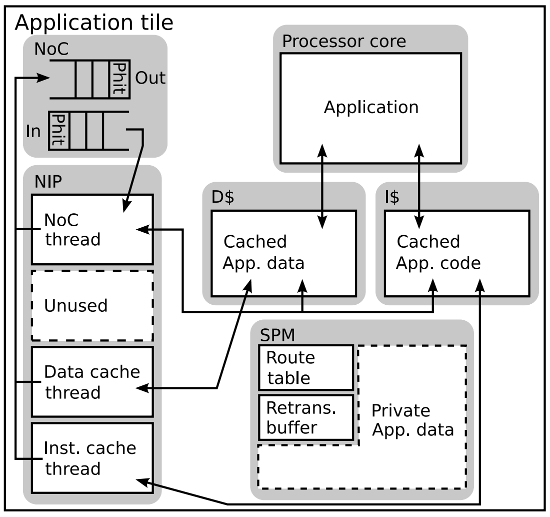


Figure 5.9: Overview of the data structures and NIP threads in the ATs. Arrows indicate data structure access. Accesses to the retransmission buffer and route table are omitted for simplicity.

Cache misses to global addresses are handled by the NIP which request the missing cache line from a DT. Also, the NIP must forward and invalidate cache lines in the local cache on request by the directories. Fig. 5.9 illustrates how this is implemented.

Thread Configuration

The NIP configuration uses only three of the four NIP threads. These manage the instruction cache, data cache and NoC interfaces respectively. The aim of this threading approach is to reduce inter-thread synchronization and thread switching by using one thread to handle all actions related to a cache miss or an incoming message. All threads that send or receive messages from the NoC must also implement the necessary flow control mechanisms. These will be discussed later.

- **Data cache thread:** The purposes of this thread is to send cache line requests to the DTs and to manage cache line replacements. The data cache thread awaits data cache misses using the associated lock variable. Sending a cache line request involves a number of steps:

1. The first step is to determine the cause of the miss, i.e. if it requires a read-only or read-write copy of the cache line.
2. When using multiple DTs, the responsible directory is selected based on the address of the missing cache line.
3. Send a request for a cache line copy to the DT.
4. Move victim cache line to victim buffer if replacement is necessary. Replacement is not necessary if the cache miss is due to a write to a shared state cache line.
5. The final step is to send an eviction request to the responsible DT if a victim cache line has been placed in the victim buffer.

Subsequently, the data cache thread blocks itself while waiting for the next data cache miss. Replies from the DT and the insertion of the missing cache line into the data cache are handled by the NoC thread.

- **NoC thread:** The NoC thread handles all incoming NoC messages. The NoC thread awaits incoming NoC packets using the associated lock variable. The first step of the NoC thread is to decode the message header to determine the message type. The following actions depend on the message type of the received message:

    1. Incoming cache lines that have been requested by the data cache thread or the instruction cache thread are inserted into the caches. The completion of the cache miss is confirmed by sending a confirmation message to the directory. This indicates to the directory that the AT now holds a copy of the cache line.
    2. Cache line copy requests from DTs are handled by sending a reply message containing the cache line to the AT specified in the DT request. "Fwd_GETX" requests require the cache line to be invalidated afterwards.
    3. Invalidation requests from DTs are handled by setting the status bits of the specified data cache line to invalid.

- **Instruction cache thread:** The instruction cache thread is similar to the data cache thread with a few exceptions. Instructions do not require any cache coherence considerations. Instruction cache misses are therefore processed by the instruction cache thread by sending a request for a cache line copy directly to the memory interface tile. Furthermore, eviction requests are not necessary as the instruction cache lines are never modified.

Flow Control and Routing

To ensure that messages can be retransmitted in case of packet loss in the NoC, as discussed in Sec. 5.2.3, all threads must store a copy of all outgoing messages in the retransmission buffer in the SPM. The retransmission buffer is managed as a software controlled circular buffer. All packets are identified by their destination and a sequence number. Separate sequence number counters are maintained in the SPM for all destination tiles. All incoming messages that are received by the NoC thread, except acknowledgement messages, are acknowledged by sending an acknowledgement message to the sender. Incoming acknowledgement messages are handled by the NoC thread, which clears the corresponding entry in the retransmission buffer. Message retransmission is based on time-outs signalled by a hardware timer in the NIP that is accessible as a lock variable. If a message has not been acknowledged before the time-out, it is assumed lost and it will be retransmitted by the NoC thread. Duplicate messages due to false time-outs are detected by the receiver using the message sequence numbers. If the retransmission buffer is full, incoming requests packets are dropped to ensure that acknowledgements can get through and free up retransmission buffer space.

Routes to all tiles involved in executing the application are maintained in the route table in the SPM as illustrated in Fig. 5.9. The route table is indexed by the destination tile identity and the route is copied to the outgoing NoC buffer.

5.5.3 Directory Tile Configuration

The DTs are processor tiles dedicated to supporting the cache coherent shared memory programming model. Hence, all in-tile resources of the DTs, including the processor core, can be used to implement the directory part of the cache coherence protocol. The directory is an inherent point of contention in the system that may lead to queuing of requests in the incoming NoC buffer during contented periods. According to Little's law [67], the size of the queue is proportional to the directory latency. Increasing the directory latency will not only increase the latency of the directory itself, it will also increase the time spent queuing at the directory significantly. Both the queuing time and the latency

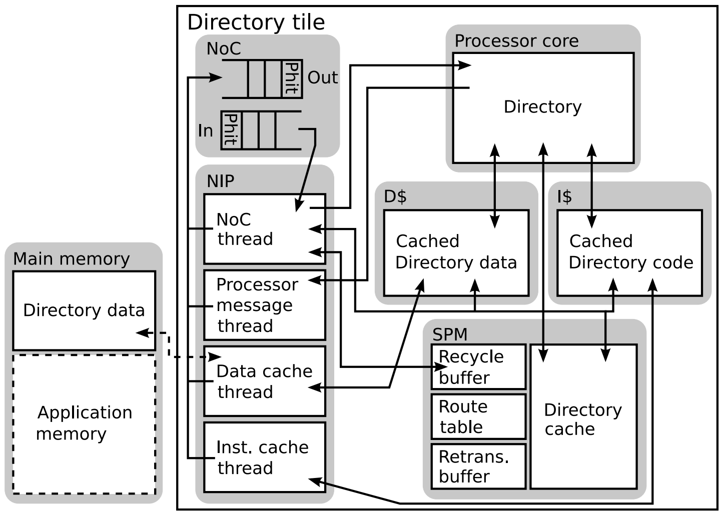


Figure 5.10: Overview of the data structures and NIP threads in the DTs. Arrows indicate data structure access. Accesses to the retransmission buffer and route table are omitted for simplicity.

of the directory itself contribute to the cache miss latency experienced by the ATs. It is therefore important to minimize the directory latency.

To mitigate this issue, the DTs use both the NIP and the local processor core to process directory request. Similar to the LimitLESS cache coherence protocol of the MIT Alewife multiprocessor [17], common case directory requests are handled by the NIP, while corner cases are off-loaded to the processor core.

Directory Data Structures

The Clupea architecture has no dedicated memory for storing directory information. Instead, the directory information, i.e., the state and the list of sharers of all cache lines currently cached by data caches of the ATs, must be stored in main memory and the SPMs of the DTs. To take advantage of the spatial and temporal locality of the directory accesses, the SPM is used to hold a software-managed directory cache, DC. The DC caches the directory data stored in off-chip main memory as illustrated in Fig. 5.10.

The DC is managed by the processor core and the NIP can only access directory information in the DC. Thus, on DC misses, the NIP hands over the request to the processor core, which processes the request and updates the DC. By using the processor core to manage the DC, the data cache of the DT is used to cache the directory data stored in main memory. Hence, the cache line directory data can be found in three locations: i) the DC in the SPM, ii) the data cache in the DT, or iii) the off-chip main memory. Increasing the number of DTs will decrease the number of cache lines handled by each directory and generally decrease the probability of a DC miss. Furthermore, the DC miss rate can be improved by selecting a proper DC block size and associativity [72].

The directory data is stored using a two-level scheme [2]. DC entries are optimized for common case directory accesses and reduced memory usage. Each entry consists of three parts that can be mapped into 64 bits: i) The cache line state. ii) A vector of the IDs of the first five ATs that hold a copy of the cache line, i.e. the sharers. An extra bit is used to indicate if there is more than five sharers. This is similar to the directory entries of the LimitLESS cache coherence protocol [17]. iii) A pointer to any waiting requests in the recycle buffer, which will be described later.

Directory entries in main memory use a bit field to represent sharers to allow the number of sharers to increase beyond five. As the NIP can only access directory information in the DC, requests for cache lines with more than five sharers must also be handled by the processor core.

Incoming requests for cache lines that are currently in a transition state can not be processed immediately. Instead, they are temporarily stored in the *recycle buffer* in the SPM. The size of this buffer is bounded by the number of possible outstanding cache misses in the system. When the cache line leaves the transition state, the request is fetched from the recycle buffer and processed.

Similar to the ATs, the DTs have route information and the retransmission buffer in the SPM.

Thread Configuration

The DT uses all four NIP threads and the processor core to implement the directory. The NIP threads are responsible for the following tasks:

- **NoC thread:** The NoC thread handles all incoming messages and implements the part of the cache coherence protocol that is handled by the NIP. Processing of common case directory requests involve the following steps:

1. First, directory information is read from the DC using the address of the requested cache line.
2. If the cache line is not in a transition state, a cache line request is sent to the first sharer or the memory interface tile, if there are no current sharers. If the cache line is in a transition state, the request message is moved to the recycle buffer and processed later.
3. Invalidation requests are sent to all sharers if an exclusive read-write cache line copy is requested.
4. The directory information in the DC is updated to reflect the new state of the cache line.
5. If a request in the recycle buffer is pending for the cache line, it is processed and removed from the recycle buffer.

As mentioned previously, the NIP can only handle request for cache lines that are shared by five or fewer ATs and that have its directory information available in the DC. In the remaining cases, the request is moved to the SPM and the request is passed on to the processor core using the co-processor interface. The affected DC line is set to a locked state to avoid race-conditions between the NIP and the processor core. The DC line is later unlocked by the processor core, when it has been updated. Additionally, the NoC thread is also responsible for handling incoming data and instruction cache lines that have been requested by the data and instruction cache threads.

- **Processor message thread:** The processor message thread allows the processor core to send request and invalidation messages to ATs. The thread blocks on the lock variable associated with the co-processor interface of the processor core. As a part of handling off-loaded directory requests the processor core forwards cache line requests and sends invalidation messages. These messages are constructed by the message thread on request by the processor core.

- **Data cache thread:** The data cache thread awaits data cache misses caused by the processor core. Only directory information is accessed in main memory by the processor core. Thus, missing cache lines can be requested directly from the memory interface tile.

- **Instruction cache thread:** Similar to the data cache thread, the instruction cache thread awaits instruction cache misses and requests missing cache lines from the memory interface tile.

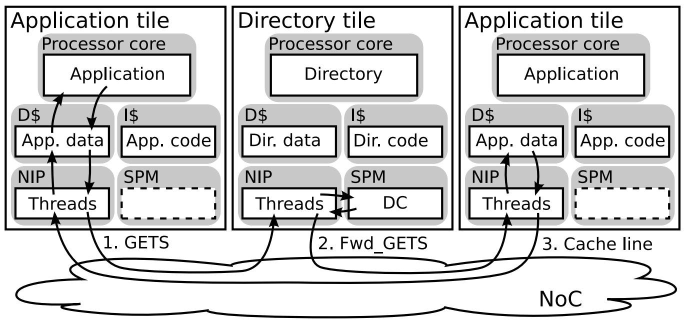


Figure 5.11: Cache read miss example. The missing cache line is fetched from a remote tile through the directory. Acknowledgement messages are omitted.

The processor core executes code that waits for interrupts from the co-processor interface caused by the NIP. Depending on the input, the processor core is required to manage the directory cache or process directory requests that can not be handled by the NIP. Off-loading directory requests to the processor cores increases the directory latency, but simplifies the NIP programming and thus improves the common case latency.

Fig. 5.11 illustrates the messages and tiles involved in a cache read miss. The local NIP sends a request, “GETS”, to the DT. Based on the cache line state in the DC, the DT forwards the request, “Fwd_GETS”, to a remote AT which holds a copy of the cache line. The cache line is sent directly to the local tile by the NIP in the remote tile. The local NIP inserts the cache line into the cache. To complete the cache miss, the NIP sends an acknowledgement message to the directory to confirm that it now holds a copy of the cache line. This message and all flow control acknowledgement messages are omitted in the figure.

5.6 Summary

This chapter described the tile-based Clupea many-core architecture, which targets future many-core platforms for embedded applications. The main focus of the architecture is platform flexibility through configurable support for programming models. The flexibility is achieved through specialized programmable network interface processors in each tile and an overall system view where tiles are considered as allocatable resources similar to memory. Allocated tiles are dedicated to the application which gives the application full access and control over the in-tile resources. The memory system is configured independently for each application by simply reprogramming the network interface processors. Hence, different programming models may co-exist in the system and programming models can be optimized for the individual application.

The Clupea architecture makes no assumptions about inter-tile communication patterns and assumes, but is not limited to, simple routers. NoC packets can be dropped at any time and flow control and retransmission are handled in the network interface processor to reduce the NoC implementation complexity.

In addition to introducing the Clupea architecture, this chapter has also discussed the implementation of one programming model: Cache coherent shared memory. The proposed implementation is based on a directory-based cache coherence protocol that uses only allocatable tiles. These are used both for executing the application and to provide programming model support.

Flexibility through programmability comes at the cost of increased latency compared to fixed hardware approach. On the other hand, the increased flexibility means that the Clupea architecture can easily support optimizations that exploit application-specific properties without modifying the hardware platform. The performance impact of the Clupea architecture will be evaluated in the following chapter which discusses modeling of many-core architectures.

Chapter 6

Modeling and Evaluation

System modeling is an integral part of the design and evaluation of new system architectures. The complexity of computer systems makes it virtually impossible to do design space exploration using prototypes. Modeling the performance of memory system and interconnect of the Clupea architecture therefore constitutes a significant part of the work leading to this thesis. The outcome of this work is two contributions in the form of memory system performance models for tile-based many-core architectures:

- An analytical model for early design space exploration that allows fast evaluation of many-core architectures with little modeling effort.
- The MC_sim simulator, which is a detailed trace-driven simulation model, which captures the behavior of the on-chip memory system at cycle-accurate level of precision.

The aim of both models is to evaluate the Clupea architecture in the cache coherent shared memory configuration against similar programming model support implemented in fixed hardware to determine the performance impact of the Clupea architecture. However, the models can be used to model other programming models or different many-core architectures with some modifications.

The chapter is organized as follows: Sec. 6.1 gives an introduction to the general modeling approach used for both system models. Sec. 6.2 introduces the analytical model and present results obtained for the Clupea architecture. Sec. 6.3 presents the MC_sim many-core memory system simulator. Finally, Sec. 6.4 presents and discusses the evaluation of the Clupea architecture con-

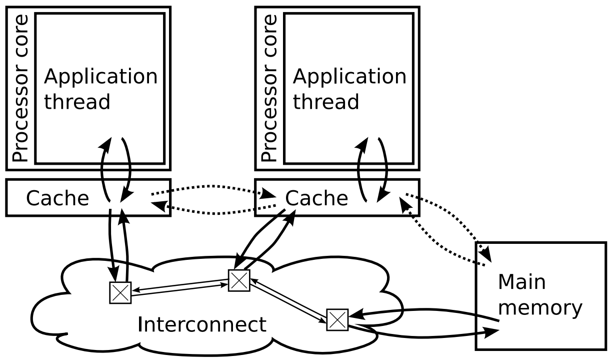


Figure 6.1: Many-core memory system model overview.

figured for cache coherent shared memory as described in Sec. 5.5 based on detailed simulation results obtained using MC_sim.

6.1 Introduction

The overall aim of the analytical model and the simulation model developed in this chapter is to evaluate programming model support for cache coherent shared memory in terms of impact on application execution time. This section gives an overview of the components of the system architecture that need to be modeled and how they interact. The model components are illustrated in Fig. 6.1, which shows a simplified view of the Clupea architecture.

6.1.1 Application Modeling

The overall application modeling approach used for both models is to consider the shared memory applications as a set of threads executing on a number of processor cores. Each thread can be considered as a generator of memory accesses that must be handled by the memory system as indicated by the interaction arrows in Fig. 6.1. Each access occurs at a certain time and accesses a certain memory address for either loading or storing data. Collectively, they generate the *memory access pattern* of the application. The actual data content of the memory is not interesting for the modeling objective of the two models. Only timing and the location and state of the data need to be considered to model the contention in the memory system and the interconnect. The timing of memory references is not static since different memory latencies of different memory system implementations affect the processor core stall time on memory accesses. The timing of memory references can therefore only be related to the completion of the previous memory reference, i.e. the time spent executing between two memory reference instructions.

Instruction caches are not modeled since these do not need to consider cache coherence and they are assumed to have negligible impact on application execution time.

6.1.2 Memory System Modeling

The focus of the memory system modeling is to determine the latency of a memory accesses generated by the application model. The latency depends three factors: i) The cache controller latency, i.e., the NIP latency, in the ATs, ii) the latency and the location of the cache coherence directories, and iii) the location of the cache line and the latency retrieving it, i.e., the AT which holds a cache line copy in its cache or the memory interface tile if the cache line is not present in any of the caches. The location of the cache line is determined by the cache coherence protocol and the previous memory accesses to the cache line. Thus, the model of the memory system must also consider the cache coherence protocol.

6.1.3 Interconnection Network Modeling

The communication between tiles that takes place as a part of the cache coherence protocol uses the NoC interconnect. Thus, the latency of a memory reference also depends on the NoC latency, which is highly non-uniform for most NoC topologies. NoCs typically have lower latency between adjacent tiles than distant tiles. This means that the mapping of application threads to ATs has influence on the communication latencies. The *communication pattern* of the application is the result of the memory access pattern, the application-to-tile-mapping and the memory system. Furthermore, contention between messages within the NoC, e.g. for access to router ports, may also influence the communication latency.

The three models are combined as shown in Fig. 6.1: The memory accesses generated by the application model are input to the memory system model, which generates messages that are transferred by the interconnect model.

6.2 Analytical Modeling

Early evaluation is necessary for design space exploration for many-core architectures due to their complexity. At this stage few design parameters have been decided and thus it is infeasible to build a simulation model. Instead, an analytical model can be used to get initial estimates on the performance of the design. This section presents an analytical model for comparing the average memory access latency and application execution time of different programming model implementations in a many-core system. The section is based on a published paper [88] and first describes the model components and then present results for the Clupea architecture.

6.2.1 Model Overview

Capturing the exact behavior all the three model components introduced in Sec. 6.1 in a unified analytical model is very complicated and may require input data that is not available. Previously proposed models of interconnects and memory systems are based on extensive sets of model parameters [5, 93, 97]. The aim of the analytical model is to focus on the programming model implementation by comparing different implementations of cache coherent shared memory using the same cache coherence protocol. More precisely, the model objective is to evaluate the impact of the processing time of the cache coherence protocol operations on the application execution time. In this scenario, the application and the interconnect models are used with the same parameters. Only timing parameters of the memory system model differ between the shared memory implementations.

Analytical modeling will inevitably introduce some uncertainties in the results. In this model, the uncertainties may influence the outcome of the comparison in favor of one of the implementations. To avoid that the result is in favor of the Clupea architecture, the model takes a pessimistic approach by adding weight to the influence of the implementation differences and favoring faster reference implementations whenever it is possible.

6.2.2 Application Model

The application execution time is affected by the memory access time and thread synchronization. Estimating the synchronization impact requires a complex application model which can not be easily constructed at an early stage in the design process. Thus, the model neglects synchronization to follow the aim of using only a small set of obtainable parameters.

Comparing two systems with the same programming model support means that the only difference is the memory system. The type and amount of the memory accesses generated by the application is the same. Thus, it is sufficient to model only the average memory access time and not consider the number of memory accesses generated by the application. The analytical application model is reduced to modeling the average memory access pattern. The average memory access pattern is modeled by the expression in Eqn. 6.1, where m is the memory access type, c_m is the cost of the memory access type in clock cycles and p_m is its probability. These are discussed in detail later.

$$t_{mem} = \sum_m c_m p_m \tag{6.1}$$

The relative execution time impact of cache coherence protocol implementation can then be determined as the cycles per instruction ratio, CPI, figures of the two implementations.

6.2.3 Memory System Model

Each memory access type in Eqn. 6.1 is associated with a number of events in the memory system. A memory access causes either a cache miss or cache hit. Cache hits are modeled by a constant memory access time. A cache miss leads to a number of protocol events, which typically includes sending messages between tiles. Expanding Eqn. 6.1 using the miss rate per instruction of the application, M_{app}, leads the following equation for comparing the relative application execution time impact of memory system implementation A and B. For simplicity, non-memory instructions are assumed to have a CPI of 1.

$$r_{exec} = \frac{M_{app}(c_{missA} - c_{missB})}{M_{app}c_{missB} + (1 - M_{app})c_{hit}} \tag{6.2}$$

The analytical cache coherence protocol model is based on the approach proposed by Srbljic et al. [97]. The cache coherence protocol is modeled as a set of protocol events, k, which corresponds to fetching cache lines from other caches, writing cache lines to main memory, and cache line state changes. Some of these events involves sending messages across the NoC. Each event is associated with a cost pc_k in terms of latency and a probability pp_k. The average cache miss latency in cycles is then determined by Eqn. 6.3.

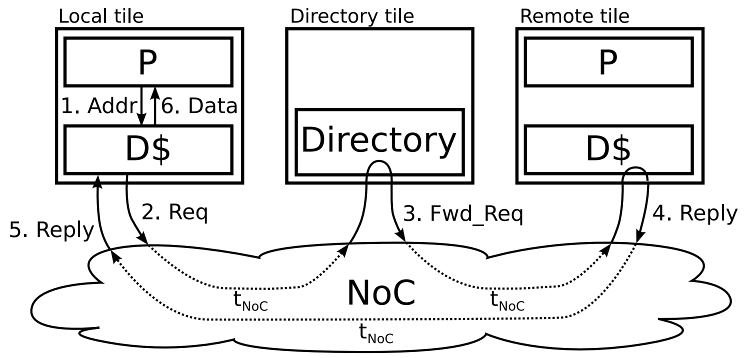


Figure 6.2: Cache miss example for directory-based cache coherence protocol.

$$c_{miss} = \sum_k pc_k pp_k \tag{6.3}$$

pp_k is determined by the memory access pattern of the application. The protocol event latency pc_k can be decomposed into two parts: i) The network latency $t_{NoC}(F)$ of the n_m messages involved in the event, which depends on the communication pattern of the application F, and ii) the protocol latency $t_{protocol}(k)$ of the n_t tiles involved in the event.

$$pc_k = \sum_{n_m} t_{NoC}(F) + \sum_{n_t} t_{protocol}(k) \tag{6.4}$$

Message transfers and protocol latency may be overlapped in time, so only transactions and protocol latency that contribute to the critical path should be included in n_m, n_t and $t_{protocol}(k)$. The NoC latency is discussed in Sec. 6.2.4. Comparing two implementations of the same protocol means that only $t_{protocol}(k)$ is different for the two implementations.

In the directory-based cache coherent shared memory implementation proposed in Sec. 5.5, cache misses always involve three messages, $n_m = 3$, in the critical path as illustrated in Fig. 6.2. Cache line eviction requires eviction permission from the directory, but using a cache line victim buffer this can be removed from the critical path.

The protocol latency, $t_{protocol}(k)$, in Eqn. 6.4 has contributions from each of the three tiles, $n_t = 3$, involved in the cache miss.

- The protocol latency in the local tile is given by $t_{local} = t_{loc-req} + t_{loc-rsp}$. $t_{loc-req}$ represents the latency from cache miss until the request message has been send into the NoC, Fig. 6.2(1-2), and $t_{loc-rsp}$ is the latency from the arrival of the first phit of the reply, Fig. 6.2(5-6), until the cache line is in place in the cache.

- The directory latency, t_{dir}, represents the DT latency from reception of the first phit in the request until the first phit of the forward request is sent into the NoC, Fig. 6.2(3).

- The remote tile latency, t_{rem}, is the latency of the remote tile, from the reception of the forward request until the reply has been send into the NoC, Fig. 6.2(4).

Contributions to $t_{protocol}(k)$ caused by contention are not considered in the model as it is assumed that the cache controllers are only lightly loaded and that the directory requests are load-balanced across several directories to avoid saturation.

Assuming that cache lines can always be found on-chip in other ATs and that $t_{protocol}(k)$ is the same for all misses, Eqn. 6.3 is reduced to one set of protocol events. This assumption complies with the pessimistic modeling approach, since including the latency of off-chip memory reduces the relative influence of the main focus of the model, $t_{protocol}(k)$.

6.2.4 Interconnect Model

The latency, $t_{NoC}(F)$, is determined by a number of factors related to the communication pattern of the application and the NoC topology. The NoC latency can be considered to have two major contributors: i) The unloaded network latency, and ii) the added latency caused by link contention.

The unloaded network latency is determined by: The distance (in terms of link/router "hops") between the source and destination tiles and the latency per hop. This is illustrated in Fig. 6.3. The latency of constructing messages and sending them into the NoC is included in the protocol latency, $t_{protocol}(k)$, hence it is not represented in the interconnect model.

The communication pattern is modeled as a probability matrix F, which describes the probability of communication between all N tiles in the system. F_{sd} is the fraction of messages sent by tile s of all messages in the system that are destined for tile d.

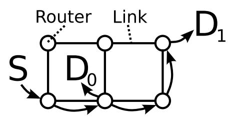


Figure 6.3: Interconnect model overview.

$$\sum_{s}\sum_{d\neq s} F_{sd} = 1, \qquad s,d = 1\dots N \tag{6.5}$$

Using the probabilities F_{sd} and the distances between the communicating tiles d_{sd} it is possible to determine an average communication distance, which allows the average interconnect latency for a single phit to be determined using the hop latency.

$$hops_{avg} = \sum_{s}\sum_{d\neq s} F_{sd} d_{sd}, \qquad s,d = 1\dots N \tag{6.6}$$

$$t_{NoC}(F) = t_{router} + hops_{avg} \times (t_{link} + t_{router}) \tag{6.7}$$

Modeling NoC contention analytically requires detailed knowledge about the temporal communication pattern of the application and a detailed model of the NoC architecture. This information is not available at an early stage in design process. Thus, for the reason of simplicity contention is not modeled in the analytical interconnect model. This means a general underestimation of the network latency. However, this complies with the overall modeling goal of taking a pessimistic approach. Reducing the contribution of t_{NoC} increases the influence of $t_{protocol}(k)$ in Eqn. 6.4, which is the main focus of the model.

6.2.5 Results and Discussion

This section presents analytical modeling results for comparison of the Clupea architecture configured for cache coherent shared memory and a similar hardware-based architecture, where the cache coherence protocol and directories are implemented directly in hardware. The aim is to evaluate the performance impact of providing support for programming models using the programmable Clupea NIP compared to a fixed hardware implementation of similar support for cache coherent shared memory.

Protocol	Description		Clupea	Hardware
$t_{loc-req}$	Miss to message in NoC	[cycles]	20	15
$t_{loc-rsp}$	Reply from to data in cache	[cycles]	25	20
t_{dir}	Intra-tile directory latency	[cycles]	40	30
t_{remote}	Intra-tile remote tile latency	[cycles]	40	35
NoC	Parameter			
t_{link}	Link latency	[cycles]		1
t_{router}	Router latency	[cycles]		4

Table 6.1: Model parameters for protocol processing latencies of the Clupea implementation and the fixed hardware approach.

The model parameters listed in Tab. 6.1 are estimates based on the proposed Clupea NIP architecture and latency estimates for previous cache coherence protocol implementations using programmable protocol engines [40, 35]. The latencies of the hardware approach is based on a minimum message handling latency of 10 cycles and assume a fixed 10 cycle directory processing latency. Cache controller processing times are assumed to be 5 and 10 cycles for processing requests and transferring cache lines respectively. The latencies are slightly pessimistic compared to the average latencies observed for the detailed simulation in Sec. 6.4. The interconnect used in the evaluation is a NoC with a mesh topology with minimal routing. Messages consists of seven phits. The communication pattern, F, is assumed to have uniform distribution.

The cache miss latency of the two implementations are compared using Eqn. 6.3 in Fig. 6.4 to show how the overhead of the protocol processing scales when the number of tiles increases. The Clupea graph represents the relative cache miss latency increase for NIP-based implementation compared to the fixed hardware approach. Fig. 6.4 also includes more pessimistic latency estimates where the total protocol processing latency is increased by 10 to 40 cycles compared to the Clupea estimate in Tab. 6.1. These illustrate how the cache miss latency is affected by the protocol processing latency. From the figure it is clear that the impact of the latency of protocol processing time in Clupea NIP decreases rapidly as the number of cores increases.

Estimating the application execution time impact using Eqn. 6.2 requires the knowledge of the cache miss rate per instruction of the application. Figure 6.5 shows the relative application execution time increase of the Clupea implementation for a range of benchmark applications. The miss rate per instruction for the applications are based on data previously published by Leverich et al. [65]. The miss rates per instruction are obtained by simulation of a MESI cache co-

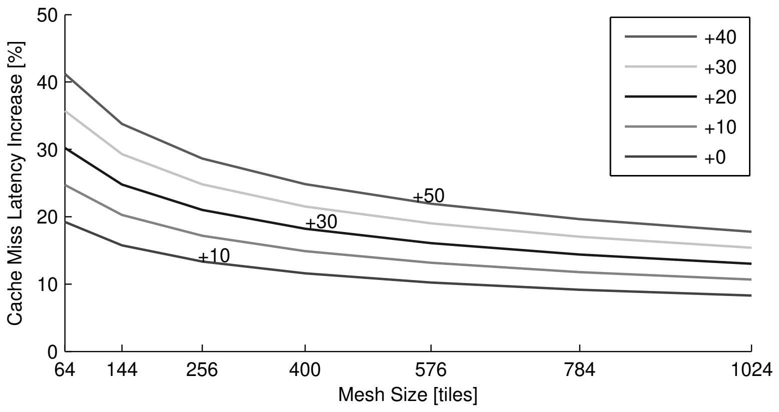


Figure 6.4: Relative increase of the cache miss latency as the number of tiles increases, which in consequence increases the average communication distance.

herence protocol with 32 kilobyte data caches on multiprocessor with 16 tiles. Thus, for this comparison it is assumed that the modeled cache coherence protocol has similar miss rates per instruction. The comparison assumes $CPI = 1$ for all instructions that do not cause cache miss misses. Figure 6.5 is based on an average communication distance of 3 hops, which is approximately the average communication distance in a 16-tile mesh NoC when the communication pattern is uniform.

The analytical model can also be used for early evaluation of modifications of the cache coherence protocol. The following example illustrates the potential of optimizing the programming model implementation using the flexibility of the Clupea NIP. The optimized protocol assumes knowledge about which data that is private and shared in the application. Private data can be fetched directly from the memory without involving the directory. Hence, the latency of one message and the directory are avoided. The optimized protocol adds an extra set of events to Eqn. 6.3 and the probability $pp_{private}$ that determines if the missing data is private. It is assumed that the optimization requires 10 cycles of extra protocol processing in the local tile to determine the type of cache miss. Fig. 6.6 shows the relative cache miss latency of the optimized protocol compared to the hardware implementation with a fixed protocol without the private data optimization. Hence, 0% means equal average cache miss latency for the optimized Clupea approach and the non-optimized hardware approach.

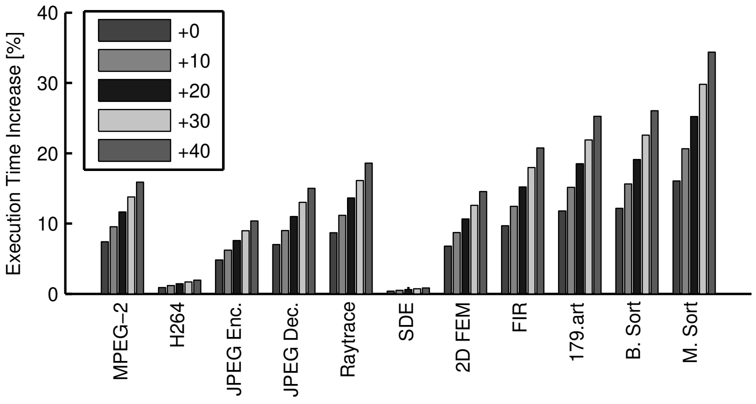


Figure 6.5: Relative increase of the application execution time for benchmark applications.

The average communication distance is 10 hops. The graph clearly shows the potential for optimizing the programming support using the Clupea architecture for applications with explicit knowledge about private and shared data. The performance impact of Clupea implementation is cancelled by the private data optimization as $pp_{private}$ increases. This may even lead to situations where the Clupea architecture has better performance when the ratio of private memory accesses is high.

6.3 MC_sim: A Fast Cycle-accurate Memory System Simulator

The shared memory programming model implies complex interaction between processors and caches in different tiles in many-core architectures. This is further complicated by the packet switched NoC interconnects, which add non-uniform communication latencies and contention. It is virtually impossible to capture this interaction in detail in an analytical model. Thus, a simulation model that captures the behavior of the entire memory system is required to evaluate the support for programming models in the Clupea architecture in more detail.

This section describes the MC_sim simulator. The objective of the MC_sim

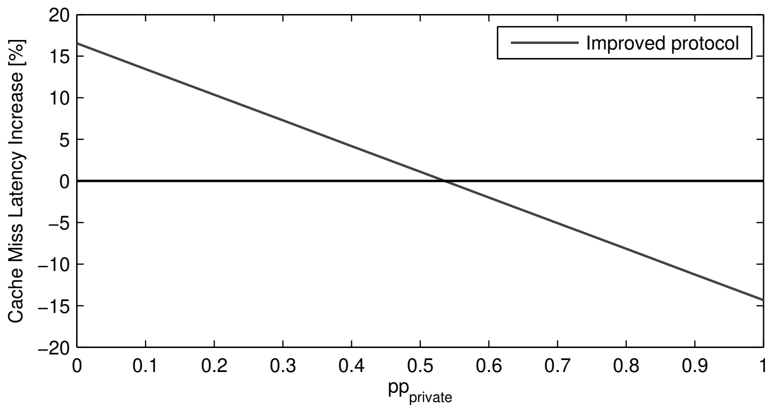


Figure 6.6: Relative cache miss latency of the optimized cache coherence protocol.

simulator is to model the Clupea architecture configured for cache coherent shared memory with a cycle-accurate level of detail while maintaining a reasonable simulation performance. MC_sim is a stand-alone trace-driven simulator implemented in C++ using a component-based design, which allows easy customization. Furthermore, fully flexible configuration of the application, directory and memory interface tiles is supported. Gathering extensive statistical information during the simulations, MC_sim is a valuable tool for studying memory systems and interconnects for many-core architectures. MC_sim is used for detailed evaluation of the Clupea architecture configured for cache coherent shared memory in Sec. 6.4.

6.3.1 Model Overview

The architectures targeted by MC_sim are tile-based many-core architectures, such as the Clupea architecture. The objective is to model the detailed behavior of both the memory system and the NoC interconnect. The simulator is based on a hierarchy of model components, where the basic components are tiles and the NoC interconnect. The tile components include models of processor tiles and tiles that are a part of the memory system. The tiles are decoupled from the NoC interconnect by a message interface. Messages are transferred between tiles by the NoC model, which models interconnect aspects such as topology and network contention. Fig. 6.7 shows an overview of the simulator components. Most components can be configured through parameters to allow easy reconfiguration of the simulated architecture and all components have parametrized timing models. While the description of the MC_sim focuses on modeling the Clupea architecture, MC_sim can be used to model any tile-based many-core architecture through the component-based design. Simulator components can easily be modified or replaced.

Fundamental principles of simulation imply that cycle-accurate simulation of parallel systems is inherently slow. Detailed execution-driven full system simulation using simulators such as Simics [104] with GEMS [71] can easily require several days to simulate even small benchmark applications. MC_sim therefore only focuses on parts of the system that are essential for modeling programming model support and the memory system. By applying trace-driven simulation the simulator complexity is significantly reduced and this allows more effort to be put into a detailed memory system model. Pre-generated traces leads to a loss in precision compared to execution-driven simulation since thread synchronization is not captured. To mitigate this issue, MC_sim uses synchronization emulation during trace replay to ensure realistic thread synchronization. This will be discussed in more detail later.

To reduce the simulator complexity further, MC_sim does not model instruction fetching or instruction caches. Instruction caches require no coherence considerations and are assumed to have negligible impact on the memory system. Further details about generation of application execution traces will be discussed in Sec. 6.3.2.

The memory system model captures the detailed cache behavior and the cache coherence protocol processing timing in a cycle-accurate level of detail. This includes internal contention, occupancy, and a detailed cache coherence protocol implementation of the protocol described in Sec. 5.5. Similarly, the NoC interconnect is also modeled in detail to capture the impact of message contention. The following subsections will describe the MC_sim model components in more detail.

6.3.2 Application Model

The MC_sim application model is based on memory reference traces to capture the behavior of the application and the processor cores. The memory reference trace can be considered as a recording of a particular run of an application. It is a complete log of all memory accesses generated during execution of the application with a particular input. Fig. 6.8 shows a memory reference trace

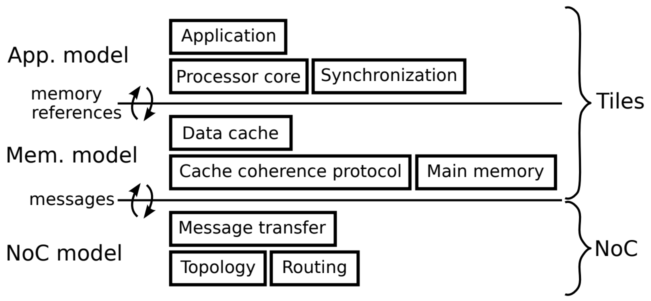
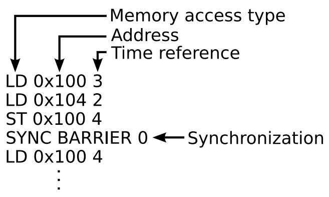


Figure 6.7: MC_sim component overview.



```
LD 0x100 3
LD 0x104 2
ST 0x100 4
SYNC BARRIER 0
LD 0x100 4
   ⋮
```

Figure 6.8: Application memory reference trace example.

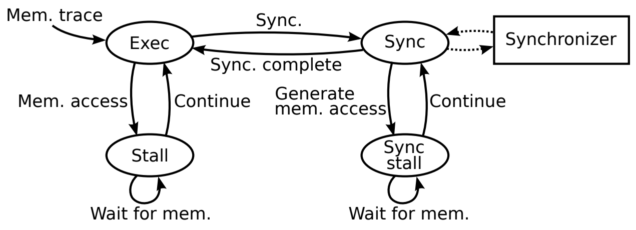


Figure 6.9: Processor core model. During normal execution, the processor core is either executing or blocked due to a data cache miss. The synchronization states are used to model thread synchronization.

example containing four memory references and a synchronization indication. Synchronization is discussed later. Each memory reference is annotated with the memory access type and a time reference. The time reference is the number of instructions executed by the processor core since the previous memory access. Applications executing in parallel on a number of processor cores are represented by a memory trace for each processor core.

Processor Core Model

Based on the memory reference trace each processor core can be modeled by two states as illustrated by the "Exec" and "Stall" states in Fig. 6.9. The processor core is either: i) Executing and waiting for the next memory access to occur, or ii) stalled while the memory system is serving the memory access. The "Sync" and "Sync stall" states are used for emulation of synchronization. Fig. 6.10 illustrates how memory accesses are replayed using the memory reference traces. The processor core is assumed to spend one cycle executing each instruction that does not cause a data cache miss. The model can be extended to handle multiple outstanding memory accesses, however, in the current model the processor cores are assumed to allow only one outstanding memory request.

Modeling Synchronization

Capturing the synchronization between application threads that execute on different processor cores is crucial for the relative timing of memory accesses. Synchronization is typically based on locks and barriers in shared memory applications. Ignoring these synchronization primitives can lead to memory access patterns that can never occur during real executions of the application due to changes in the execution environment of the simulated systems. This issue is known as the "global trace problem" [44]. Thus, the simulator must ensure that the trace replay is synchronized correctly to prevent threads from advancing past a synchronization point before all threads are synchronized.

Additionally, synchronization leads to periodic memory accesses to the data structure of the synchronization primitive by each thread to determine when the synchronization has completed. The number of memory accesses depends on how long each thread is waiting for the synchronization to complete. The progress of the threads is affected by the memory system which means that the number of synchronization memory accesses varies depending on the simulated memory system. Thus, instead of using a pre-determined synchronization memory access pattern the simulator must construct a new pattern of memory references that is realistic for the synchronization primitive on the simulated system. Any memory reference related to the synchronization primitives must be omitted from the trace. To identify accesses to synchronization primitives, MC_sim uses explicit notation in the memory reference traces to uniquely identify encountered locks or barriers as shown in Fig. 6.8. MC_sim emulates the memory references of locks by implementing test-test and set-based spin-locks with exponential back-off in the application model. Barriers are emulated on basis on lock emulation.

Synchronization is modeled by the "Sync" and "Sync_stall" states in Fig. 6.9. When a lock or barrier is encountered in the trace, the processor core model switches to the synchronization states. In these states, the processor core model generates memory references related to the synchronization operation until it succeeds, i.e., the lock is obtained or the barrier is passed. MC_sim does not include data in the memory model, hence synchronization must be supported by a global synchronizer component, which coordinates all processor core models. The synchronizer determines the success of the synchronization operation as illustrated by the simplified barrier synchronization example in Fig. 6.10. For every memory access related to the barrier, the processor accesses the synchronizer to check if the synchronization has completed. AT "x" fails to pass the barrier in the first attempt and must retry until AT "y" has reached the barrier. This example only shows access to one memory location associated with the barrier, the implemented barrier emulation consist of a series of memory accesses.

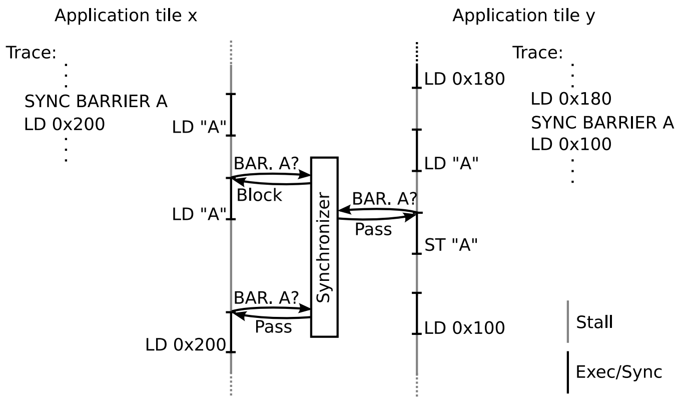


Figure 6.10: Simplified barrier synchronization example with two threads. Timing references are omitted for simplicity.

Tracing Methodologies

Memory traces can be obtained by logging the memory references during execution of the application on an execution platform. MC_sim does not include an instruction set simulator, thus traces must be obtained from a different source. Traces from any execution platform can be used as long as they are compatible with the trace format used by MC_sim.

The memory reference trace captures both the application and the internal processor core architecture. The trace should therefore be obtained using an execution platform with processor cores similar to the simulated system. The two following methods can be used to obtain memory traces using physical and simulated platforms.

- **Binary instrumentation:** Binary instrumentation tools, such as Pin [47], allow every instruction of each application thread to be instrumented to generate memory traces on a physical execution platform. This method will generate a separate trace for each application thread and not for each processor core as it is assumed by the MC_sim model. Hence, this method can only be used if the number of threads is equal to the number of processor cores in the simulated system. Though, any number of threads can be traced regardless of the number of cores on the execution platform. Binary instrumentation traces only virtual memory address accesses of the application itself and does not include operating system calls.

- **Execution-driven simulation:** Execution-driven simulators such as Simics [104] can be used to obtain traces from simulated execution platforms. The memory references are logged directly from the memory interfaces of the simulated processor cores. These traces contain physical addresses of all memory accesses generated by the application and the operating system. This method requires an execution platform with the same number of cores as the simulated system. Synchronization is captured by instrumenting the application code to notify the execution platform simulator when a synchronization operation is encountered. The results that will be presented in Sec. 6.4 are all based on traces obtained using this method with the Simics [104] simulator and GEMS [71] memory system model.

It is common for all tracing approaches that the memory reference trace only represents one possible path of execution in the application. Applications with dynamic thread management and load balancing may generate different traces depending on the outcome of synchronization. Thus, trace-driven simulation of this type of application may not always give an accurate picture of the simulated system.

6.3.3 Memory System Model

The memory system model includes all components of the memory system including caches, the cache controller, the NoC interface and memory interface tiles as shown in Fig. 6.7. The memory system model is partitioned into local models for each tile that communicate using the NoC model as illustrated in Fig. 6.11.

- **Application tiles:** In the ATs, the local memory system model is responsible for modeling the local data cache, the scratch-pad memory and the cache controller which includes the NoC interface. Inputs are memory accesses from the application model and incoming messages from the NoC model. The functional cache controller model fully implements the cache coherence protocol proposed for the Clupea architecture.

- **Directory tiles:** The DT model accepts inputs in the form of directory requests from the NoC model. The model implements the directory part of the cache coherence protocol including the directory data structure, the DC and off-loading to the processor core.

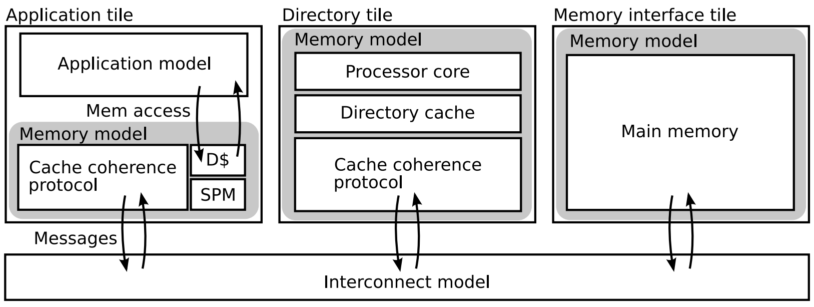


Figure 6.11: Memory model overview.

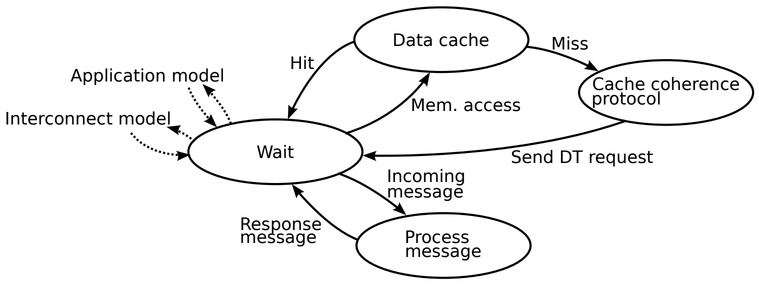


Figure 6.12: Local memory system model in each tile.

- **Memory interface tiles:** The memory interface tile is driven by incoming messages from the NoC model and models the latency of accessing off-chip main memory.

Fig. 6.12 illustrates the general states of the local memory system model of the ATs. Memory accesses from the application model are passed to the cache model to determine if a cache miss has occurred. In case of a cache hit, the application model continues executing the memory reference trace. If a cache miss occurs, the application model blocks execution of the local processor core trace and the cache controller model takes over to service the cache miss and send a request to the directory. Incoming messages from the NoC are handled by the cache controller model according to the cache coherence protocol. This includes forwarding cache lines and unblocking the trace execution when a missing cache line has been received.

The models of the directory and memory interface tiles follow that of the application, but have no interaction with the application model. The functional models of the memory system are combined with a detailed cycle-accurate timing model that, in contrast to other models [71], determines the latency of all memory system operations individually based on the actions involved in the operation. The timing model is based on a set of parameters, which specifies the cycle latency of all basic actions in the memory system model. Occupancy is modeled by blocking the memory model from processing new inputs for duration of the action specified by its parameters. Latency is modeled by delaying messages and cache updates for the duration of their associated latencies. Correct ordering of events in all model components is ensured by using global simulation time. All model components are invoked once every clock cycle to ensure consistent time progress in all parts of the simulator.

6.3.4 Interconnect Model

The interconnect model is modeling message transfers between tiles. Each tile is associated with a NoC port, which enables the tile to send and receive messages. Tiles are addressed by a unique tile identification number, which is used by the interconnect model to deliver messages.

Internally the model consists of a set of router models, which are configured to resemble the modeled NoC topology. Each router is modeled as a set of queues and a routing algorithm. The NoC model does not model message dropping as this is assumed to happen only in extreme cases. Instead, the NoC model avoids deadlock by having unbounded router buffers [1]. Message acknowledgement and retransmission buffers are still included in the model, however, retransmission never occurs. The mapping of tiles to NoC ports is fully configurable.

As discussed earlier, the interconnect latency has two main contributions: The unloaded latency and the latency caused by contention. Both of these contributions are captured by the simulation model, though, the NoC model does not model latency related to retransmission. The router latency and throughput are specified through model parameters. Contention is detected by the router model and introduces additional router delay to the blocked message.

6.3.5 Simulator Implementation

MC_sim has been implemented as a stand-alone simulator in C++ with simplicity in mind to allow easy modifications. Most model parameters, including the timing model, are run-time configurable through parameters and configuration files. Data and directory caches are fully configurable with regard to size, cache line size and associativity. Many-core architecture of arbitrary configurations are automatically configured for a specified number of application, directory and memory interface tiles. Memory reference traces have been obtained using Simics and are compressed to reduce disk space utilization. MC_sim offers extensive statistical data collection for each individual tile in the system.

Simulation speed depends on the system configuration. Typical simulation speeds for a system configuration with 8 ATs, 4 DTs and a single memory interface tile is approximately 300K cycles/s on a recent high-end x86-processor. For a configuration with 64 ATs, 32 DTs and a single memory interface tile, the simulation speed is reduced to approximately 50K cycles/s.

6.4 Results and Discussion

This section presents modeling results for the Clupea architecture using MC_sim. The evaluation of the Clupea architecture is based on the cache coherent shared memory implementation described in Sec. 5.5. The main purpose of the evaluation is to determine the execution time impact using the Clupea architecture compared to a fixed hardware cache controller approach, HWC, that implements similar support for cache coherent shared memory.

6.4.1 System Configuration

The evaluation considers systems with 8 to 64 ATs with 2 to 8 ATs per DT and a single memory interface tile for both the Clupea architecture and the HWC approach. It should be noted the AT/DT ratio does not imply any association between the ATs and the DT. Any AT can potentially access any DT. The basic system configuration listed in Tab. 6.2 is used for all experiments unless otherwise stated. The ATs and DTs are assumed to have 256 kilobyte local memory which is organized into 64 kilobyte data and instruction caches, and a 128 kilobyte SPM. As mentioned earlier, the instruction cache is not modeled. Assuming a DC entry size of 64 bits the maximum DC size is 8192 entries. The default DC organization is two way set associative, where each set consists of eight entries. This organization is further discussed later. Directory requests are load-balanced using a memory address bit mask to ensure that cache line requests to a particular cache line always are handled by the same directory. This distribution method allows DTs to operate independently. The default address bit mask is placed above the cache line byte index bits and the DC index bits so that the directory load balancing is independent of the DC organization to ensure a more fair comparison between different DC organizations. This means that the directory load balancing is based on 256 kilobyte address space segments, i.e., all addresses in a 256 kilobyte contiguous memory address space map to the same DT. A more fine-grained load distribution is discussed later.

6.4.2 Timing Models

As mentioned earlier, the evaluation considers two main system architectures: The Clupea architecture and the HWC approach.

- **Clupea:** The Clupea timing model represents the latencies of the NIP. The timing model is based on the latency estimates listed in Tab. 6.3. The latency estimates are based on MC_sim cache coherence protocol code and the proposed NIP architecture that has been illustrated earlier in Fig. 5.4. Every basic block of the protocol code has been associated with latencies derived from NIP pseudo-code implementing the same functionality. Each of these latencies are included in the timing model to provide a detailed picture of all system bottlenecks.

  The NoC packet layout is assumed to have routing information in the first phit and the remaining header fields are encoded in the second phit. This allows all header fields to be decoded in a single cycle. Moving messages to the recycle buffer and processor core offloading require most of the header fields to be stored in the SPM and therefore takes several cycles. Similarly, updating buffers managed in the SPM requires pointers to be updated and also involves checking for buffer overflow. Directory cache access requires several cycles to load and check the directory cache tag against the request address. The AT and DT latencies are determine by MC_sim by associating the latencies listed in Tab. 6.3 with the operations in the MC_sim simulation model.

- **HWC:** The HWC timing model is assumed to implement the same functionality as the NIP in the cache coherent shared memory configuration in the ATs, but it implements the cache controller and the cache coherence protocol directly in hardware. The latency of most common operations is therefore reduced. The HWC timing model is based on the Clupea model where latencies are reduced to reflect potential hardware optimizations. The DTs in the HWC approach are assumed to have a hardware controller

System Configuration	
ATs	8-64
ATs per DT	2-8
Memory interface tiles	1
NoC topology	2D mesh
Tile mapping	Dir. tiles in center
Cache coherence protocol	MOSI
Application and Directory tiles	
Processor core	In-order execution
Processor core instruction latency	1 cycle/inst.
Data cache size	64 kilobyte
Data cache associativity	4-way
Data cache line size	64 bytes
Data cache replacement policy	pseudo-LRU
Data cache eviction buffer size	1 cache line
Data cache hit latency	1 cycle
Scratch-pad memory (SPM) size	128 kilobyte
Scratch-pad memory access latency	1 cycle
Directory Configuration	
Directory cache (DC) entries	8192
Directory cache entry size	64 bits
Directory cache associativity	2-way
Directory cache block size	8 entries
Directory cache entry tile IDs	5 (10 bits each)
Directory load balancing	256 kilobyte segments
Memory Interface tile	
Outstanding accesses	8
NoC	
Link (phit) bit width	128 bits
Link throughput	1 phit/cycle
Request message size	2 phits
Payload message size	6 phits

Table 6.2: Clupea architecture configuration.

which implements the directory controller instead of the NIP, however, it still uses a processor core for offloading of complex directory operations. The DTs have dedicated hardware DCs. The DTs can not be reconfigured, thus the number of DTs is fixed at design time. The latencies of the HWC listed in Tab. 6.3 are comparable to previous detailed studies on implementations of cache coherent shared memory multichip multiprocessors [72, 73]. The cache miss latency of the DCs are assumed equal for the two approaches. Later it will be clear that this has very small effect as the directory miss rate is very low for most applications.

In addition to the realistic HWC main reference model, a selection of other variants will also be included in the comparison to provide a more complete picture. These are: i) An utopian single-cycle model where the total processing time of all AT and DT requests are reduced to one recycle, i.e. the response is always available the cycle after the request has been received. Thus the effective cache miss latency is reduced to a few cycles. A true single-cycle cache miss latency is not technically possible in the current MC_sim implementation. The data caches are enlarged to 8 Mb and configured to have 32-way associativity to emulate infinite fully associative caches. This reduces the number of cache misses to minimum. The mesh interconnect is replaced with a single-cycle crossbar. This model serves as the lower bound on the benchmark execution times. ii) A variant of the utopian model with data caches of the default size. Compared to the other utopian model, this will show the effect of the data cache size and associativity. iii) An aggressive implementation similar to the second model, but with the default mesh interconnect. This reference shows the lower bound on the execution time that can be achieved by minimizing the AT and DT latencies. iv) A low-power edition of the HWC implementation, where all AT and DT controller latencies are 50% longer to model the extended gate-delay of a low-power implementation.

The NoC latencies are based on a recent CMP implementation [46] and previously proposed NoC architectures [92]. Tiles are assumed to be separate clock domains and thus messages must pass through synchronization hardware when they enter and exit the NoC.

6.4.3 Benchmark Applications

The evaluation is based on traces of eight benchmark applications from the SPLASH2 [107] benchmark suite. These parallel benchmark applications have an execution time that makes them suitable for detailed simulation. More com-

NIP/Cache controller operations	**Clupea**	**HWC**
Pipeline start-up	2	1
Logic/arithmetic	1	1-0
Conditions	1-2	1-2
Directory cache access (hit)	6-8	2
Message seq. no. check	3	2
Send message acknowledgement	8	5
Send packet header	3	2
Send packet payload	4	4
Recycle message	13	4
Data handle forwarding	1	-
SPM access	1	-
Update retransmission buffer	5	4
Offload to processor core	12	10
Directory processor		
Offloaded directory update	20	
Send invalidation message	20	
Directory cache miss	100	
NoC		
NoC router	4	
NoC link	1	
Clock domain synchronization	2	
Memory Interface		
Main memory latency	200	

Table 6.3: Simulation model latencies for Clupea architecture and hardware controller reference in processor core cycles.

Benchmark	**Input**	**Instructions**	**Mem. access**
Barnes	16384 particles	315×10^6	95×10^6
Cholesky	tk29.0	643×10^6	153×10^6
FFT	2^{20} data points	94×10^6	25×10^6
LU (contiguous)	512x512 matrix	168×10^6	45×10^6
Ocean (contiguous)	258x258 grid	164×10^6	49×10^6
Radix	2^{23} keys	291×10^6	115×10^6
Volrend	Head (256x256x126 voxels)	1.7×10^9	427×10^6
Water (spatial)	512 molecules	94×10^6	35×10^6

Table 6.4: Benchmark applications, benchmark input and trace information in terms of average instructions and average memory accesses per processor core for 16-core traces.

plex benchmark applications require weeks of simulation time for a single run. Since a detailed study requires hundreds of simulation runs, more complex benchmarks are not feasible options. The benchmark applications and their inputs are listed in Tab. 6.4. The single-threaded benchmark execution times for these inputs range from 1.1s to 4.2s on Sun Fire 6900 with 1200 MHz Ultra-SPARC IV processors. The memory reference traces of the applications have been recorded using Simics [104] with GEMS [71] configured to emulate a cache coherent shared memory SPARC architecture multiprocessor system. Locks and barriers of the benchmark applications were replaced with custom implementations that allowed synchronization to be detected and replicated in MC_sim as discussed in Sec. 6.3.2. Only the parallel sections of the benchmarks have been traced. The benchmark applications have been used unmodified and thereby do not use the SPM of the ATs.

6.4.4 Relative Execution Time

Fig. 6.13 shows the benchmark execution time of the Clupea architecture normalized to the HWC approach execution time for systems with 8 to 64 ATs and AT/DT ratios of 2, 4 and 8. Results for 1, 2 and 4 ATs could not be produced due to technical issues related to recording traces. Note that the normalization considers only the execution time increase caused by the different latencies of the two approaches, i.e., simulations of the same memory reference trace using the same number of ATs and DTs, but different timing models. Hence, a relative execution time of 1 means that the Clupea architecture benchmark execution time is equal to the execution time on the HWC model. The average relative execution times for all eight benchmarks with 8, 16, 32, and 64 ATs and 4 ATs per DT are 1.31, 1.48, 1.46, and 1.56 respectively. The individual impacts on the execution times range from 1.12 to 1.81 for these configurations and show that the execution time impact of using the Clupea NIP to provide support for programming models varies significantly with the memory reference pattern of the benchmark applications. The main reason for this relatively large overhead is not the Clupea architecture itself, but the result of queuing at the DTs due to contention. This is discussed in more detail later. Increasing the number of DTs to distribute the directory load across more DTs reduces the relative execution time. For instance, for the LU con. benchmark the relative execution time is reduced to 1.07 by reducing the AT/DT ratio to 2. However, as it will be discussed later, increasing the number of directories does not work equally well for all benchmarks.

The relative execution time generally increases slightly as the number ATs grows even when the AT/DT ratio is kept constant. The cause of this is increasing imbalance between the loads on the DTs as the number of directories increases, which will be discussed later.

The FFT and Radix benchmarks show interesting behavior in Fig. 6.13 by showing little improvement going from 8 to 4 ATs per DT with 16 ATs while a significant improvement can be observed when the ratio is reduce to 2. The reasons for this are different for the two benchmarks. For the FFT benchmark the drop is a result of little improvement in the HWC execution time by reducing the AT/DT ratio to 2 while the Clupea approach experiences significant performance gain. Thus the normalized improvement is significantly better. The reason for the larger improvement for the Clupea architecture is a major reduction in the directory contention that leads to a large reduction in queuing time at the DTs. Later it can be seen that this is very high for FFT in Fig. 6.20. The Radix benchmark has by far the largest DC miss rate as it is shown later in Fig. 6.27 and thus benefits from reduced DC miss rates when the number of DTs increases. Increasing the number of DTs both increases the total directory processing capacity, but also the total DC size as the request are distributed across more DTs and thus reduces the amount of DC capacity and conflict misses.

Another interesting observation is the slightly increasing relative execution time for the Cholesky benchmark with 32 allocated ATs. Cholesky causes an excessive load on the most contended DT for all directory configurations due to poor directory load balancing. Hence, the execution time for the Clupea approach is dictated by the directory latency of the highly contended directory. The HWC experiences similar contention due to the DT load imbalance and thus the relative execution time remains largely unchanged. The cause of the

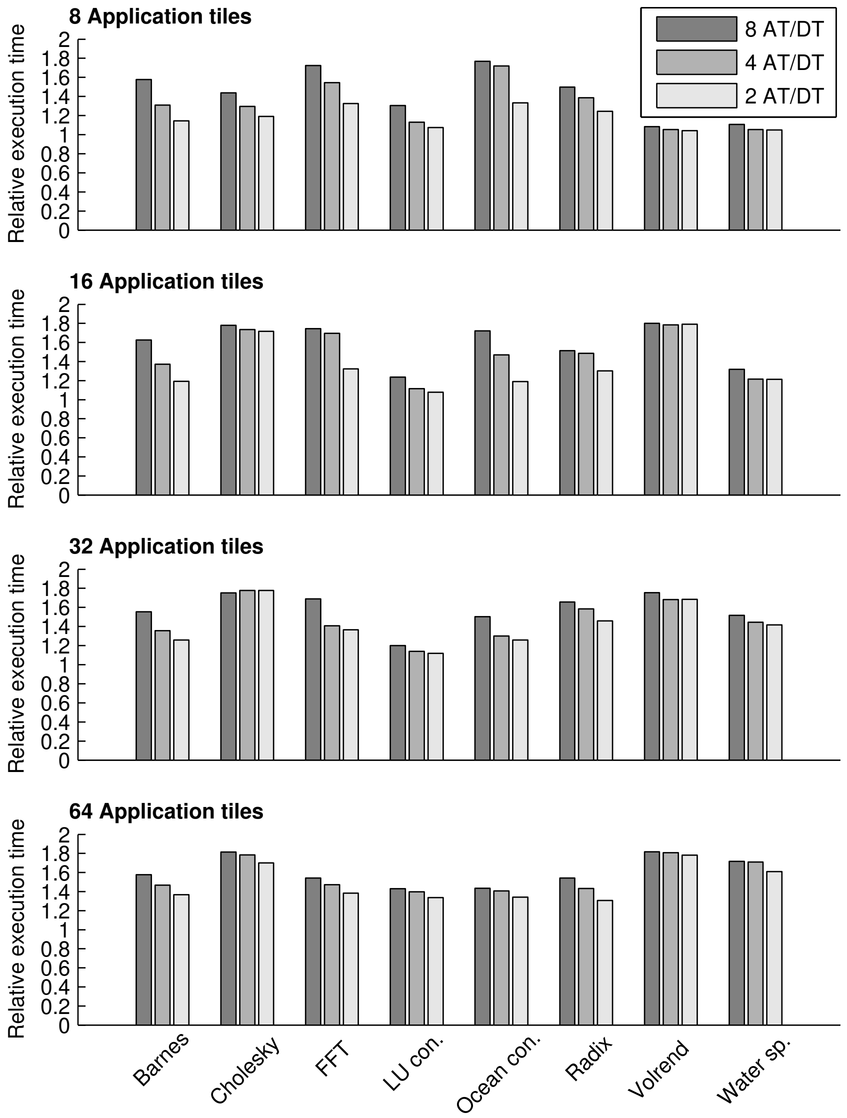


Figure 6.13: Relative execution time for eight SPLASH2 benchmark application using the Clupea architecture for 8, 16, 32, and 64 ATs with AT/DT ratios of 8, 4, and 2.

of the slight increase from having 8 ATs per DT to 4 ATs per DT is attributed to a different memory reference interleaving due to the different latencies of the two approaches.

The simulations generally show the same trends regardless of the number of ATs. Thus, simulations with 16 allocated ATs are used in the remainder of this chapter as example for more thorough discussions. In this configuration, the parallelization overheads due to the relatively small input data sets are not dominant and the number of DTs is still large enough to study the effects of directory load balancing. Fig. 6.15 and Fig. 6.14 show the absolute execution times of the benchmark applications for the Clupea and HWC models for 8 to 64 ATs normalized to the execution of the 8 AT configuration. Several benchmarks have increasing execution times with larger numbers of ATs due to the parallelization overhead in the benchmarks. Thus, the limit of feasible parallelization has been reached and further parallelization makes no practical sense. However, for 16 ATs the execution times are generally improved compared to executions with 8 ATs for both the Clupea architecture and the HWC approach. Only Cholesky and Volrend show consistently increasing execution times. However, this is the case for both timing models. Thus, the Cholesky and Volrend benchmarks generally show the performance of the two programming model implementation when the memory system is saturated. For the remaining benchmarks, the memory system is not saturated for 16 ATs and thus these benchmarks provide a more realistic picture. Comparing the the two figures shows that higher latencies of the Clupea architecture causes a slight increase in the parallelization overhead. Thus, optimal amount of parallelization is slightly lower for the Clupea architecture.

Overall, the Clupea architecture shows a trend of slightly increasing relative execution times compared to the HWC approach when the number of AT increases. The most significant increases are observed for the Cholesky and Volrend, which saturate the directories. However, this is the case for both the Clupea architecture and the HWC approach, thus the relative execution time is largely constant regardless of the number of ATs for these two benchmarks. The directory load is discussed in detail later. For the remaining benchmarks, the relative execution time only increases slightly as the number of ATs increases. This indicates that the proposed Clupea cache coherent shared memory configuration scales well for an increasing number of ATs for most applications. It has no inherent limitations and has been shown to support cache coherent shared memory in many-core systems with 64 ATs. Obviously, the Clupea architecture can not compete on performance when compared to a direct hardware implementation of the same support for cache coherent shared memory. However, it

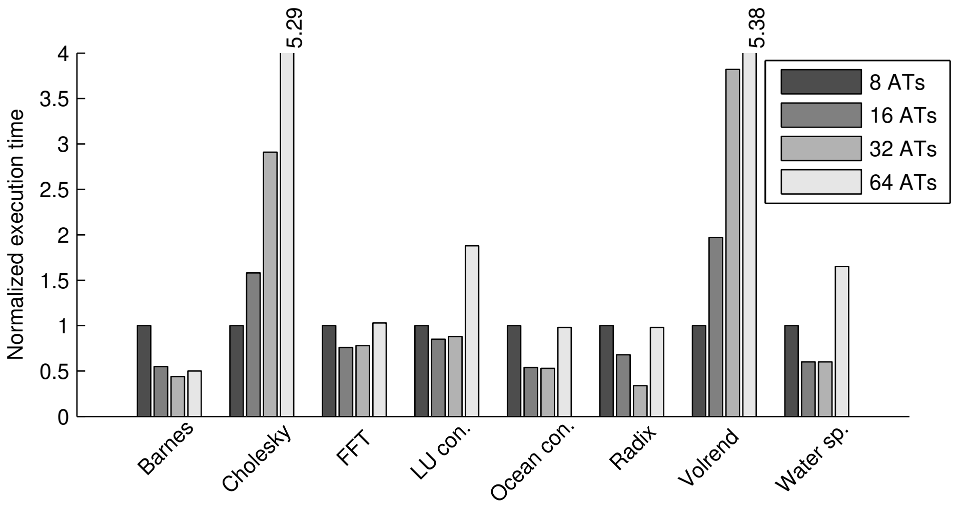


Figure 6.14: Benchmark execution times for 8 to 64 ATs and a AT/DT ratio of 4 using the HWC timing model normalized to the 8 AT execution time.

should be noted that the flexibility of the Clupea architecture allows the DTs to be reconfigured for any purpose in other configurations and that the NIP can be used to do application-specific optimizations of the programming model support. Application-specific tuning of the DC is discussed later. More extensive application-specific optimizations require support from the operating system, compilers and the run-time. Hence, this will not be considered further in this thesis.

6.4.5 In-tile Latency

To provide a more complete picture of influence of the AT, DT and NoC latencies, Fig. 6.16 shows the relative execution times for the Clupea and HWC implementations in addition to the four reference timing models mentioned in Sec. 6.4.1 for 16 ATs and 4 DTs. The execution times are normalized to the HWC model with default data cache size. As one would expect, the execution time increases caused by increasing in-tile latencies are more pronounced for benchmark applications with high data cache miss rates. Cholesky and Volrend are both examples of this. The data cache miss rates for the benchmark applications are shown later in Fig. 6.30. The execution time of the utopian model

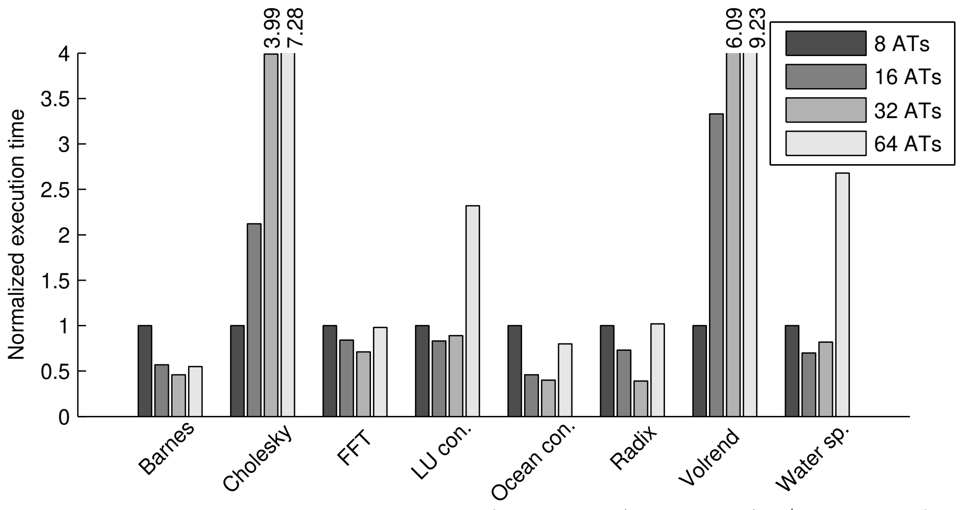


Figure 6.15: Benchmark execution times for 8 to 64 ATs and a AT/DT ratio of 4 using the Clupea architecture normalized to the 8 AT execution time.

with infinite caches shows that this can potentially be reduced for Cholesky by altering the data cache organization and size to reduce the data cache miss rate. However, for Volrend only little improvement is possible which indicates that these are coherence or cold misses. On the other hand, Ocean con. shows huge potential in having larger caches with more associativity. However, the data cache miss rate is already low, thus only a modest execution time increase is caused by the tile latencies in the more realistic timing models. An interesting observation in Fig. 6.16 is that Radix has higher relative execution time than Ocean con. despite the fact that it has lower data cache miss rate. The reason for this can be explained by the directory queuing times of the two benchmarks shown in Fig. 6.20, which is almost twice as large for Radix compared to Ocean con. The reason for this is discussed later.

6.4.6 Directory Tile Load

The previous discussions have indicated that the directory load has large influence on the benchmark execution time. Fig. 6.17 shows the normalized max., min., and average NIP loads in the DTs for the Clupea architecture in a system configuration with 16 ATs and 2, 4 and 8 DTs. A load of 100% means

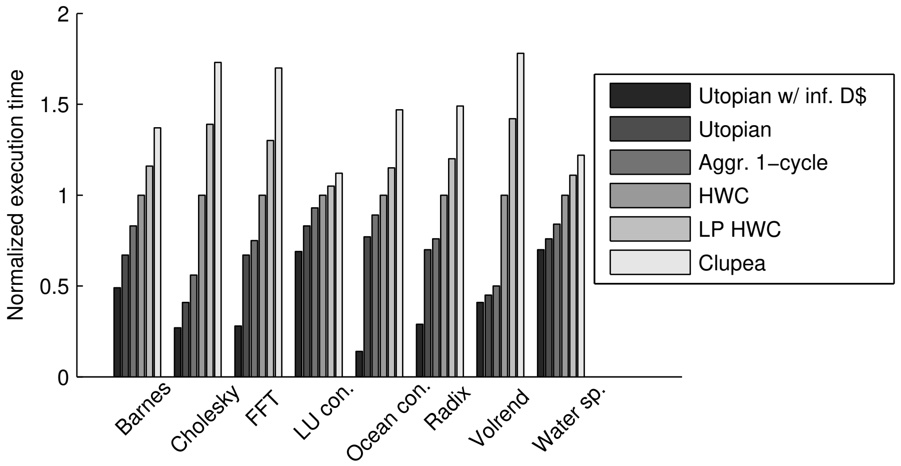


Figure 6.16: Benchmark execution times for 16 ATs and a AT/DT ratio of 4 for all timing models normalized to the HWC timing model.

fully loaded, i.e., the DT has no idle cycles. The directories are generally heavily loaded, especially when the AT/DT ratio is high. High DT load leads to increased contention which means that requests are more likely to experience queuing time in the incoming NoC buffer of the DT rather than being processed immediately. Thus, high DT load increases the cache miss latency.

Fig. 6.17 shows a general trend towards an increasing span between the max. and min. NIP loads in the directories when the number of DTs increases. The Cholesky and Volrend are two extreme case of this due to their saturation of the memory system. The observed max. DT load of these benchmarks indicate that the DT NIP is the bottleneck of the memory system. Saturation of a DT significantly increases the cache latency due to queuing at the directory. Thus, the most heavily loaded DT is dominating the cache miss latency. The queuing effect is discussed in more detail later. Cholesky and Volrend are examples of this as the average DT NIP load decreases when the AT per DT ratio is reduced, but the max. DT NIP load and the relative execution time in Fig. 6.13 are largely unchanged. For other benchmarks, reduced max. DT NIP loads as the AT per DT ratio decreases can be associated with similar decreases in the relative execution time. Thus, the most effective way to improve the relative execution time is to minimize the max. DT NIP and not just the average. Load balancing is discussed in more detail later. For comparison Fig. 6.18 shows the

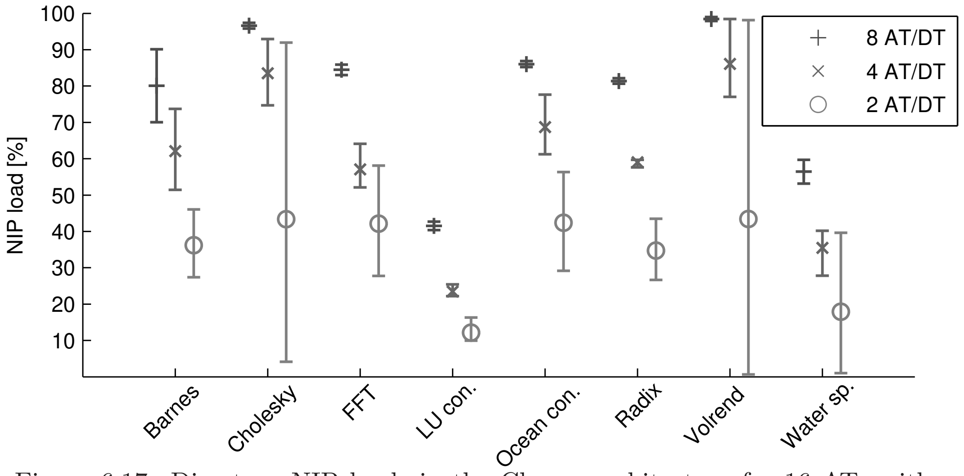


Figure 6.17: Directory NIP loads in the Clupea architecture for 16 ATs with AT/DT ratios of 8, 4, and 2. Each bar indicates max., min., and avg. load.

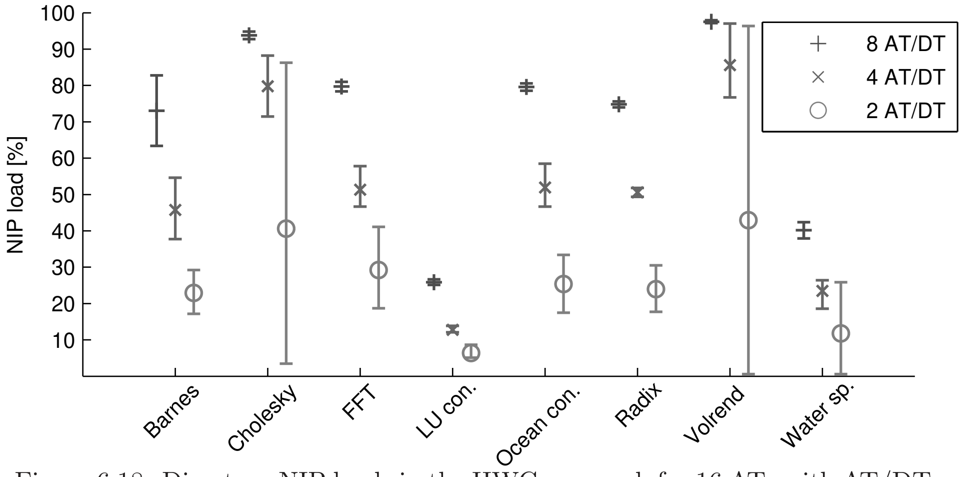


Figure 6.18: Directory NIP loads in the HWC approach for 16 ATs with AT/DT ratios of 8, 4, and 2. Each bar indicates max., min., and avg. load.

DT NIP loads for the HWC approach for a similar configuration. In spite of the lower directory latency in the HWC approach the same directory load pattern can be observed. This confirms that DTs are the bottlenecks of the system. The max. Cholesky and Volrend directory loads are still close to fully loaded showing that the rest of the system is able to saturate the DTs.

The DTs also use the processor cores to handle corner cases and DC misses. Fig. 6.19 shows the normalized loads on the DT processor core in the Clupea architecture for 16 allocated ATs. The processor core load is modest for most benchmarks. Radix has the highest processor load due to a high DC miss rate. Overall, the processor core loads decrease similar to the NIP load in Fig. 6.17, which indicates that the processor core load generally follows the NIP load in the DTs.

High DT NIP loads lead to contention and thus queuing of requests which are waiting for being processed by the DT. Using basic queuing theory [67] it can be shown that the length of the queue is proportional to the arrival rate of new requests. However, as the queuing time depends on the number of requests in the queue and the DT latency, the queuing time will increase dramatically when the directory reaches saturation. Fig. 6.20 clearly shows this effect illustrated by the average directory request queuing latency in the NoC buffer of the all DTs in the Clupea architecture with 16 allocated ATs. The rapid decrease in queuing time when the AT/DT ratio is decreased is due to the distribution of the load across more directories, which reduces the queue lengths. Doubling the number of DTs can reduce the average buffer queuing time more than 50% for several benchmarks due to the queuing effects, i.e., the queuing time increases rapidly when the NIP load reaches saturation as incoming requests always find the DT busy. Similar trends can be observed for the HWC approach, but due to the shorter DT latency, processing the entire queue takes shorter time and thus the saturated buffer queuing time is significantly lower and the HWC requires a higher rate of incoming directory requests to saturate the DTs. It should be noted that while the average request queuing latency decreases as more DTs are added in Fig. 6.20, the request queuing time at the most heavily loaded DT is not necessarily improved due uneven DT load balancing. For the Volrend benchmark, the queuing time of the most loaded DT stays largely constant as the AT/DT ratio dicreases. Thus, the max. queuing time is approximately five times the average for AT/DT ratio of 2. The unchanged max. queuing time is a direct consequence of the unchanged max. DT NIP load as discussed previously.

Correlating Fig. 6.20 and Fig. 6.17 it can be seen that high NIP load does not necessarily lead to high directory latency for all cases as one might expect. This counter intuitive observation is caused by the temporal behaviour of the

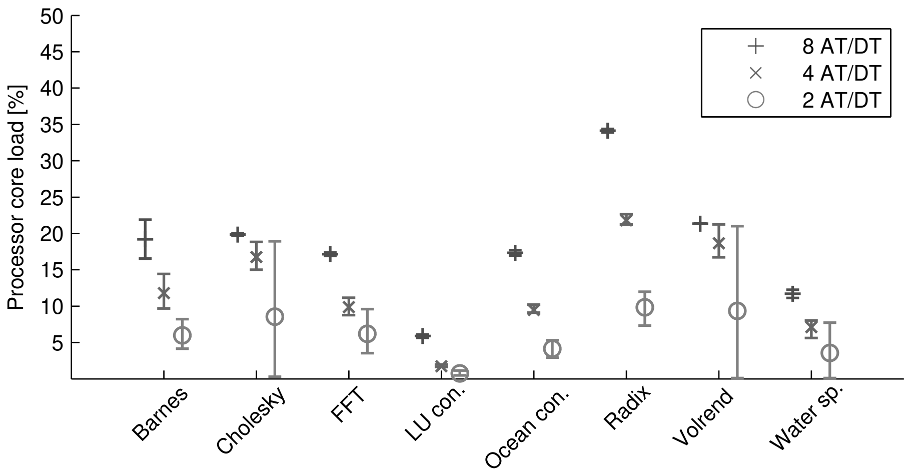


Figure 6.19: Processor core loads in the DTs in the Clupea architecture for 16 ATs with AT/DT ratios of 8, 4, and 2. Each bar indicates max., min., and avg. load.

benchmark applications. The NIP load is the average with regards to time, while the directory queuing time is the average with regards to the number of directory requests. Thus, the directory queuing time will reveal if the directory experiences contented periods during the execution. Fig. 6.21 illustrates this for FFT by showing the NIP load over time for one DT in a 16 AT configuration with 4 DTs. Several periods with full load are interleaved with long periods of very little activity. These contended periods increase the directory queuing time significantly, while idle periods causes the load to appear low on average. Similar observations can be made for the Radix benchmark that also causes long directory queuing time. For comparison, Fig. 6.22 shows the directory NIP load over time for the Ocean con. benchmark which has roughly the same directory NIP load as FFT in Fig. 6.17. In this case, the directory NIP load has a more uniform temporal distribution, which is confirmed by the lower directory queuing time in Fig. 6.20.

Similar effects were observed in the Stanford FLASH multiprocessor [40], but the Clupea architecture mitigates high directory loads by using the additional flexibility to increase the number of directories according to the needs of the application. Hence, unused directories are not wasted hardware resources when they are not needed and the performance of the memory system can be

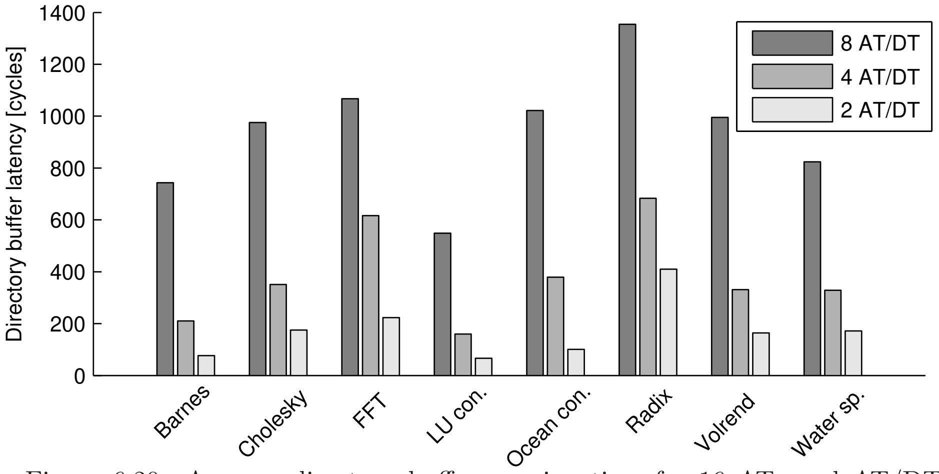


Figure 6.20: Average directory buffer queuing time for 16 ATs and AT/DT ratios of 8, 4, and 2 for the Clupea architecture.

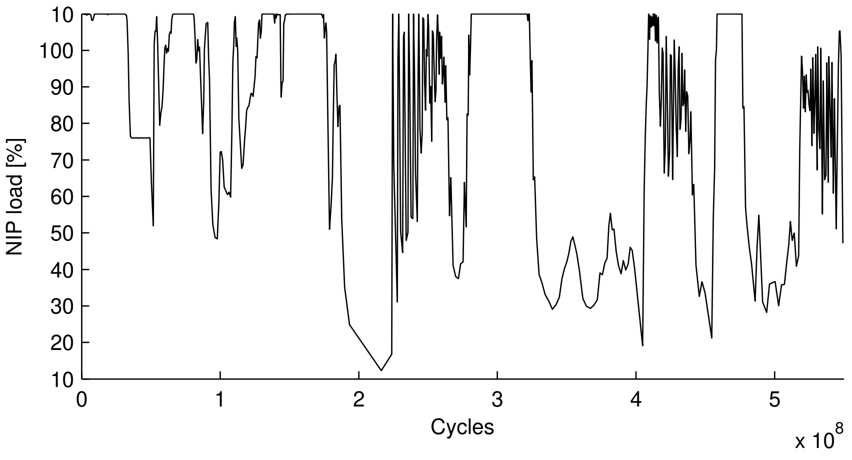


Figure 6.21: Temporal FFT DT NIP load for a single DT for 16 ATs and 4 DTs in the Clupea architecture.

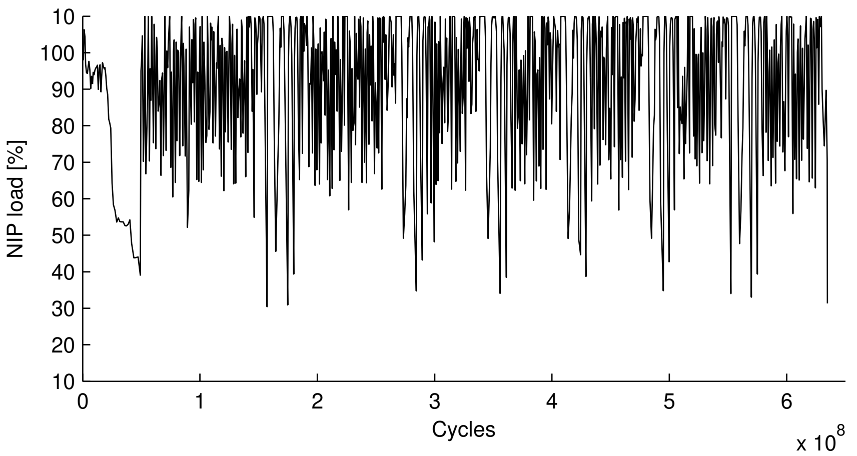


Figure 6.22: Temporal Ocean con. DT NIP load for a single DT for 16 ATs and 4 DTs in the Clupea architecture.

matched to the requirement of the application. Though, the number of DTs can not be changed during the execution of the application in the proposed cache coherent shared memory implementation. Scaling the number of DTs dynamically leads to a number of directory issues related to dynamically distributing directory requests which will not discussed further here. The major limitation of the scalability of the proposed directory implementation is the load balancing issue, which must be addressed by the compiler and operating system possibly combined with a more sophisticated directory load balancing function in the NIP.

6.4.7 Application Tile Load

Fig. 6.23 shows the max., min., and average NIP loads in the ATs for 16 allocated ATs in the Clupea architecture. The low NIP load shows that the hardware threaded NIP architecture provides the necessary processing power to manage both local and remote cache line requests. The potential for improved latency in the ATs by having multiple NIP pipelines executing simultaneously is therefore low. Furthermore, complicated thread synchronization and more contention for shared resources, such as the NoC buffers, are likely to reduce potential of such an approach even further. Alternatively, hardware resources can be

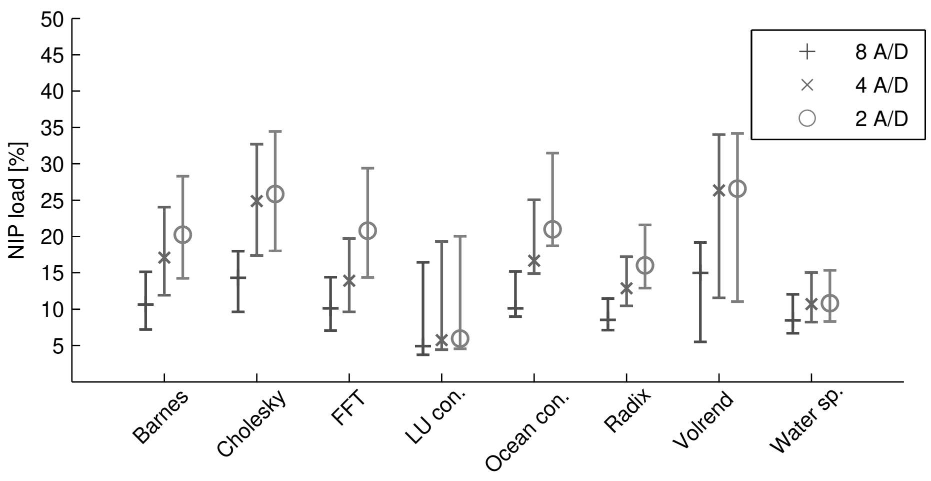


Figure 6.23: Application NIP load in the Clupea architecture for 16 ATs with AT/DT ratios of 8, 4, and 2. Each bar indicates max., min., and avg. load.

saved by reducing the number of hardware threads in NIP architecture, which already only uses three of the four hardware threads in the cache coherent shared memory configuration.

As mentioned earlier, MC_sim does not capture instruction cache misses and neglects the NIP load implied by these. The low NIP utilization shows that there is sufficient NIP processing time to handle instruction misses without overloading the NIP.

6.4.8 Memory Interface Tile Load

All experiments have been done with a single memory interface tile regardless of the number of ATs. The cache coherence protocol is optimized for serving cache miss on-chip whenever possible. Thus, it is interesting to see the load on the memory interface. Fig. 6.24 shows the memory interface tile access rate per data cache miss for the Clupea architecture with 8 to 64 ATs and an AT/DT ratio of 4, i.e., the fraction of directory requests to cache lines which are not found in any data cache of any AT. While the general rate of memory interface tile accesses is moderate, the rate is significantly reduced as the number of ATs increases. The cause of this is the fact that the benchmark input is unchanged, while the effective on-chip data cache is enlarged by using more ATs. Slight variations in

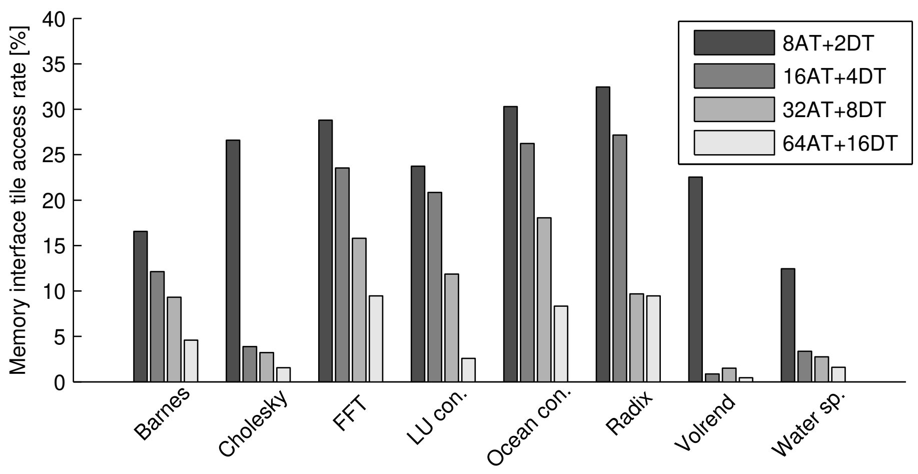


Figure 6.24: Memory interface tile access rate per data cache miss for the Clupea architecture with 8 to 64 ATs and an AT/DT ratio of 4.

the memory access rates for Volrend can be seen for an increasing number of ATs. This variation is attributed differences in the memory reference pattern of the memory reference traces. Since the memory interface tile allows multiple outstanding memory requests, the memory tile access rates shown in Fig. 6.24 do not saturate the memory interface tile. Memory interface tile contention was observed for 1% to 27% of the time for the benchmarks. Thus, the memory interface tile is not a major bottleneck in the memory system. It should be noted that approximately half the memory interface tile accesses are cache line write-backs that do not contribute to the cache miss latency.

6.4.9 Cache Miss Latency

The DT occupancy has already been identified as a major contributor to the cache miss latency in the previous discussion. Fig. 6.25 shows all contributions to the average cache miss across all eight benchmarks for both the Clupea architecture and the HWC approach with 16 allocated ATs and four DTs. The illustrated cache miss represent a typical cache miss, where the directory request is processed in the DT NIP and the missing cache line is obtained from another "remote" AT. The average directory request queuing time at the DTs constitutes 69% and 45% of the total cache miss latency for the two approaches respectively.

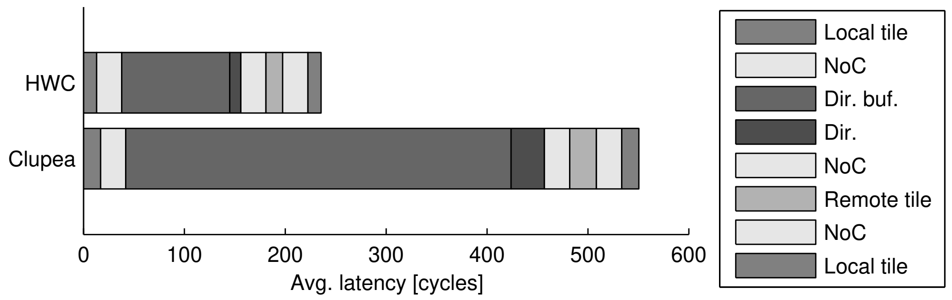


Figure 6.25: Average latencies contributing to the cache miss latency for 16 ATs with a AT/DT ratio of 4.

This latency varies significantly across the benchmarks as shown in Fig. 6.20 and decreases rapidly to less than 200 cycles for six of the benchmarks when more DTs are added. In contrast, the processing in the NIP of the directory has only a small impact on the cache miss latency.

Based on the latency contributions it is clear that the greatest potential for cache miss latency reduction is to reduce the DT queuing time. The queuing time is a result of two factors: The DT latency and the arrival rate of directory requests. Thus, there are two ways to reduce the DT queuing time. Reducing the DT latency through a more efficient implementation will reduce the processing time of each request and thus reduce the latency of processing all requests in the queue. Alternatively, the arrival rate can be reduced by distributing the directory requests across more DTs. While the former approach may require modifications to the NIP architecture, the latter has good potential due to the significant variance observed in the DT NIP loads. Comparing the DT queuing time to the NIP latency in the ATs indicates that sacrificing extra cycles in the AT to implement a better directory request distribution function has great potential. However, as it is discussed later, fixed mapping of address segments to DTs does not provide a reasonable distribution of directory requests.

The NIP latencies of the two involved ATs and the NoC only have modest contributions to the cache miss latency compared to the DT. The above observations are also valid for the HWC approach, although the directory queuing time is relatively lower due to the shorter DT latency.

The three NoC messages in the critical path of the cache miss contribute

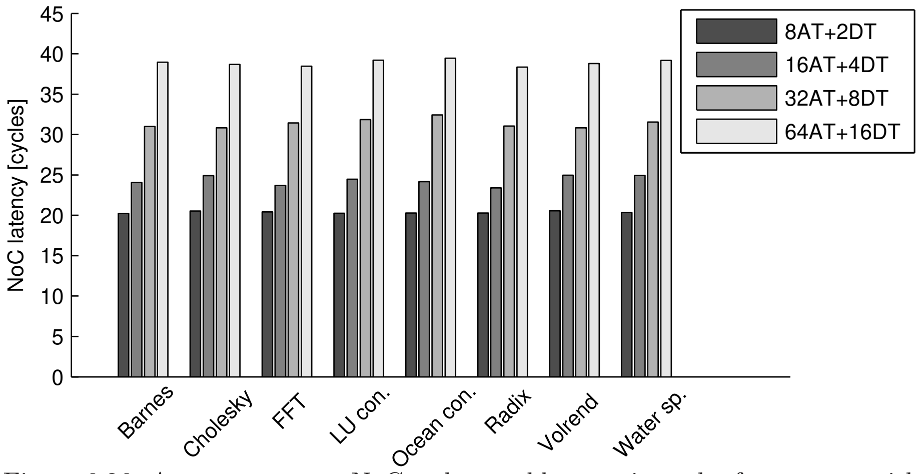


Figure 6.26: Average one-way NoC end-to-end latency in cycles for systems with 8, 16, 32, and 64 ATs and 2, 4, 8, and 16 DTs.

with a total of 75 cycles for both approaches. This contribution increases with the number of ATs, as the average hop distance increases. Fig. 6.26 shows the average one-way NoC latency for all eight benchmarks with 8 to 64 allocated ATs and four ATs per DT. For 64 ATs, the NoC latency will be the largest contributor to the cache miss latency in the HWC approach and this trend is likely to continue beyond 64 ATs. Thus, the relative impact of the NIP latencies of the Clupea approach will decrease.

6.4.10 Directory Cache Organization

The previously discussed results have been based on the default DC configuration in Tab. 6.2. The DC miss rate has a significant impact on the directory latency as DC misses block directory requests for the cache line while the DT processor core updates the DC. The DC miss rates for the Clupea architecture are shown in Fig. 6.27 and show significant variance across the benchmarks. While the general trend shows improved DC miss rates when the number of directories increases, load balancing issues may lead to more DC misses as it can be seen for a few benchmarks. Radix has a very high DC miss rate compared to the other benchmarks as mentioned earlier. The effects of this can be seen in both in the relative execution in Fig. 6.13 and the DT processor core load in

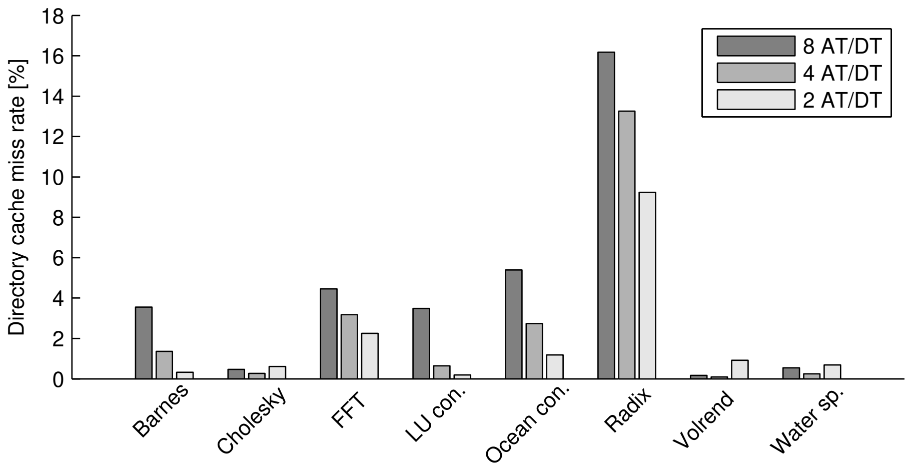


Figure 6.27: Directory cache miss rates for 16 ATs and 8 to 2 application tiles per directory.

Fig. 6.19.

Adding more DTs is not the only way to take advantage of the Clupea architecture. The DC organization can easily be reconfigured to optimize the miss rate. Fig. 6.28 shows the relative execution time of the benchmark applications for 16 ATs and four DTs compared to the default DC organization for the Clupea architecture. The DC organization is varied from direct mapped to four-way associativity and the block size, i.e. the DC line size, is varied from 1 to 16 entries. The DC size is fixed to 8192 entries for all configurations. The default configuration gives the best overall performance, but the Ocean con., Radix and FFT benchmarks can reduce their execution times by up to 17% for alternative organizations. These benchmarks all have high DC miss rates with the default configuration and thus have good potential for reductions. Generally, increasing the DC block size improves the DC miss rate due to better exploitation of spatial locality. However, when increasing the block size to 16 and beyond, the negative effects of conflict and capacity misses start to dominate. For DC associativity, having 2-way associativity reduces the DC miss rate significantly compared to a direct mapped DC. Increasing the associativity further to 4-way associtivity generally improves the DC miss rate slightly for most applications, but increases the relative execution time. This observation is atributed to the increased DC latency in the Clupea architecture, which must check more tags

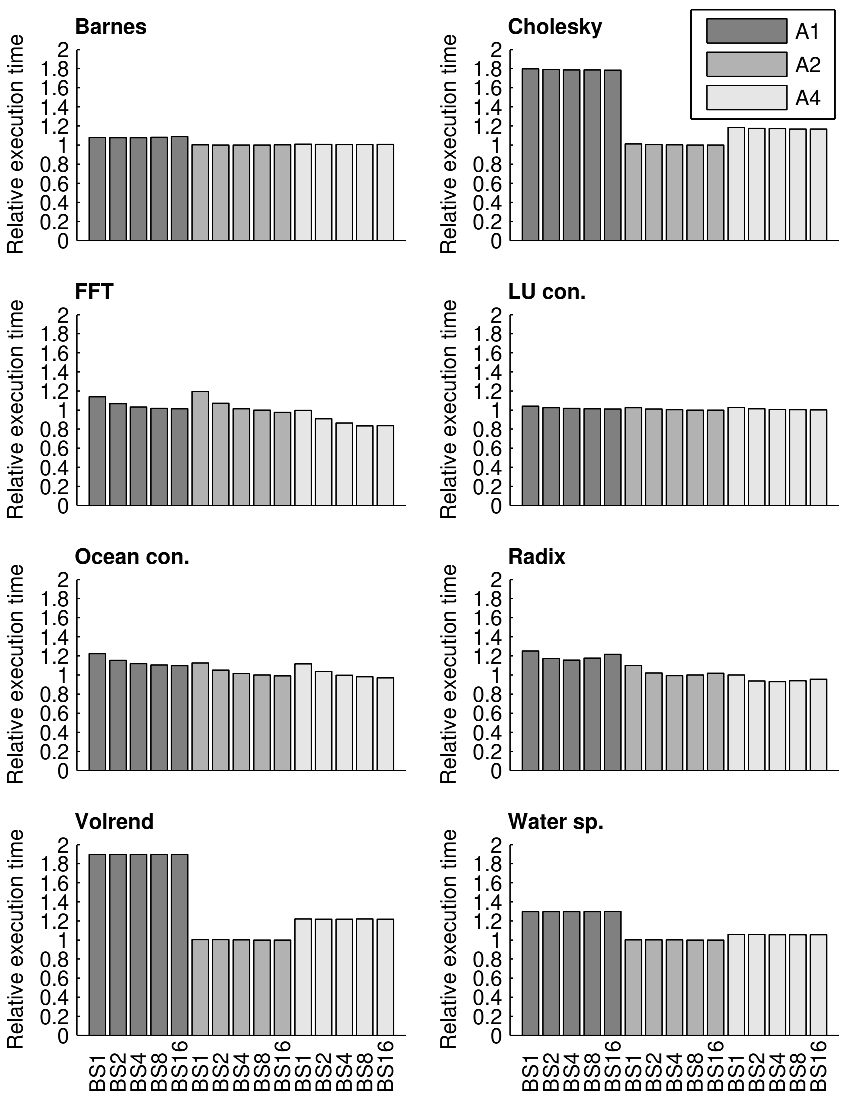


Figure 6.28: Relative execution time for different DC configurations for the Clupea architecture with 16 ATs and four DTs. The DC configuration is varied across associativity, A, 1 to 4 ways, and block size, BS, 1 to 16 cache entries.

for each DC lookup, and changes to the memory reference interleaving.

The Clupea architecture can exploit optimization through variable DC block size while the HWC approach must aim for the best overall approach. However, an average DC miss rate of 2.73% for the default configuration leaves little room for general improvements.

6.4.11 Data Cache Size

Data cache misses in the ATs is the source of directory requests and thus also load on the DTs. It is therefore interesting to see if increasing the data cache size can reduce the data cache miss rate and thereby reduce the DT load and the execution time. Fig. 6.29 shows the relative execution time for the Clupea architecture for 16 allocated ATs with data cache sizes from 64 kilobytes to 256 kilobytes per tile normalized to execution time of the default 64 kilobyte data cache configuration. Increasing the data cache size to 128 kilobytes and 256 kilobytes per tile decreases the relative execution time by 18% and 26% on average respectively for the two configurations.

Fig. 6.30 shows the max., min. and average data cache misses per 1000 instructions across the ATs for the benchmark applications for the Clupea architecture with 16 ATs and 4 DTs for the different data cache sizes. While the average data cache misses per 1000 instructions of the 16 ATs is largely unchanged when the data cache size increases, the max. data cache miss rate is reduced significantly. The effect of the large reductions can be seen directly on the execution time in Fig. 6.29 suggesting that the execution of the benchmark applications are determined by the AT with the highest data cache miss rate per 1000 instructions. Hence, increasing the data caches sizes for all ATs may be waste of resources. Alternatively, the compiler or the programmer should attempt to optimize the data accesses to achieve a more uniform data cache miss rate distribution across the ATs.

6.4.12 Directory Load Balancing

Ideally the directory requests should be distributed according to the number of requests recieved per DT and not by address segment. However, this is not practically feasible due to the fact that one DT must be responsible for all requests to a particular cache line to avoid synchronization of information between directories. All simulations until now have used directory load balancing based on 256 kilobyte segments. Decreasing the size of these segments intuitively leads to better load balancing. To investigate this, Fig. 6.31 shows the max., min. and

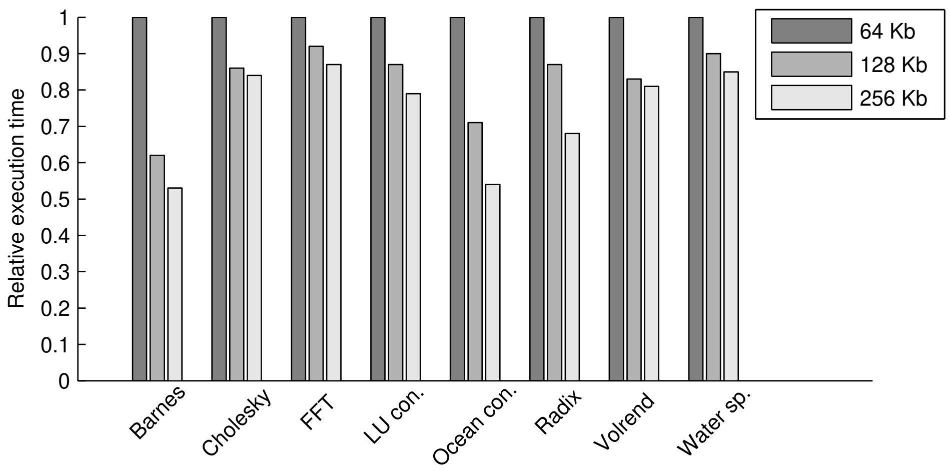


Figure 6.29: Execution time for different data cache sizes for systems with 16 ATs and four DTs normalized to the default 64 kilobyte configuration for the Clupea architecture. The data cache size is increased from 64 kilobytes to 256 kilobytes.

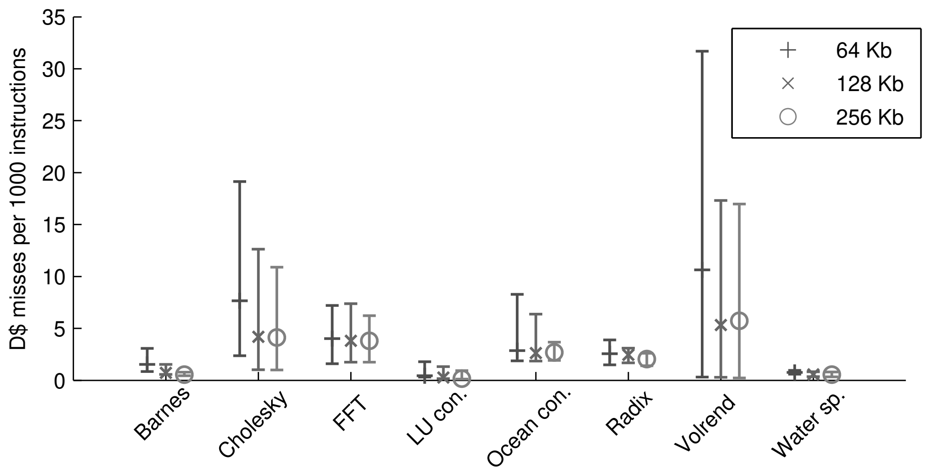


Figure 6.30: Data cache misses per 1000 instructions for the Clupea architecture with data caches of 64, 128 and 256 kilobytes.

average number of directory requests per directory for load balancing based on 256 kilobyte segments and 512 byte segments normalized to the average number of directory requests. Counterintuitively, the 512 byte segments lead to significantly worse distributions of the directory requests as the max. increases significantly for all benchmarks. The effect of this can also be observed on the execution time, which increases by 80% on average. The largest execution time increases are observed for Cholesky and Volrend, which both increase by over 200%.

The source of this uneven distribution comes from the combination of the local data caches in the ATs and the memory references in the memory reference traces. Analysis of the memory reference traces themselves show more uniform distributions with the 512 byte segments for five of the eight benchmarks. Fig. 6.32 shows the max. and min. number of memory refences that map to each DT normalized to the average number of memory refences per DT. Note that the three benchmarks that have their memory reference distribution negatively affected by the 512 byte segment approach in Fig. 6.32 are also the benchmarks which have the worst DT load balancing in Fig. 6.31 using the 512 byte approach.

A key difference between the two segment sizes is the bits used to determine the DT that should handle the data cache miss. For load balancing based on 256 kilobyte segments, the DT is determined using address bits that belong to the tag part of the miss address. In contrast, the 512 byte segment approach must use address bits which are also used as the data cache index. This means that for the 512 byte segment approach, all cache misses to a particular cache set always maps to the same DT. Contended data cache sets with a high rate of conflict misses will therefore lead to increased pressure on a single DT. In the 256 kilobyte segment approach, the cache miss address to DT mapping is determined by bits in tag part of the address and thus cache misses due to contended data cache sets are likely to be distributed across several DTs. Reducing the conflict misses through higher data cache associativity will therefore decrease this effect. Fig. 6.33 shows the max. and min. requests per DT for the Ocean con. benchmark for the Clupea architecture normalized to the average requests per DT as an example. The data cache associativity is increased from 4 to 16 and the cache size is fixed. Increasing the data cache associativity to 8 reduces the distribution skew significantly for the 512 byte segment approach, while the 256 kilobyte segments approach is only improved slightly. For 16-way data cache associativity, the 512 byte approach reduces the DT load imbalance even further and shows a significantly better load balace than the 256 kilobyte approach which experiences slightly worse load balancing due to the increased

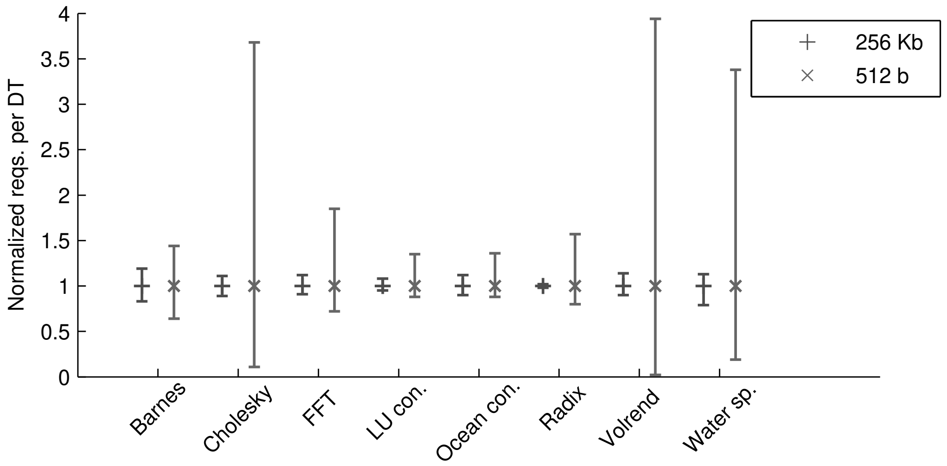


Figure 6.31: Directory requests per DT for Clupea architecture using 16 ATs and 4 DTs for directory load balancing based on 256 kilobyte and 512 byte address segments. The figure shows the max. and min. directory requests per DT normalized to the average requests per DT.

associativity. Similar trends can be observed for all the benchmark applications.

6.4.13 Interconnect Latency

The NoC latencies of listed in Tab. 6.3 assume that the NoC operates a the processor core clock frequency. Operating at this frequency may not be feasible in lower-power systems. Lowering the NoC clock frequency and thus increasing the NoC latency will influence the execution time negatively for all implementations of the programming model support. However, since the NoC latency contribution to the total cache miss latency is relatively smaller for the Clupea approach as shown in Fig. 6.25, the Clupea architecture is less affected by the increasing NoC latency. Fig. 6.34 shows the relative execution time of the Clupea architecture normalized to the HWC approach with the same NoC clock frequency for NoC clock frequencies divided by 2 and 4 for a 16 AT configuration with 4 DTs. The decreasing relative execution time of the Clupea approach is due to the fact that both implementations are affected equally by the increasing NoC latency. Furthermore, the DT load may be lowered by the fact that NoC messages spend more time in flight and thus the rate at which ATs generate new

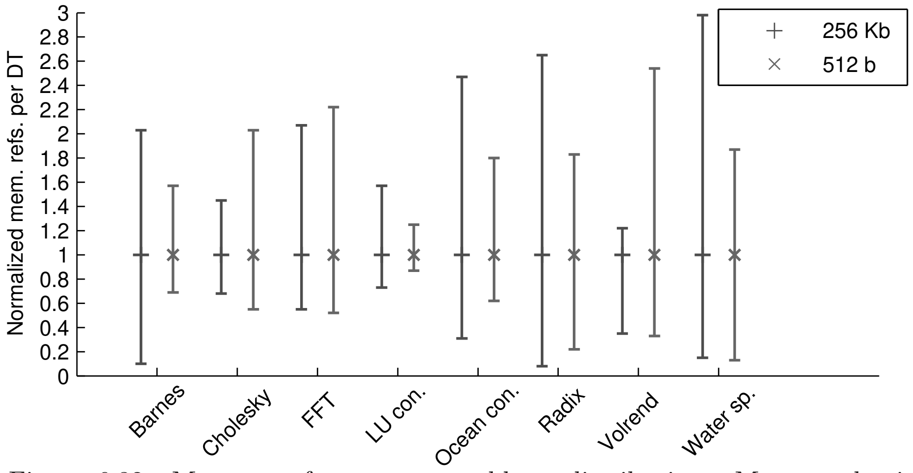


Figure 6.32: Memory reference trace address distribution. Max. and min. references per DT normalized to the average number of references per DT.

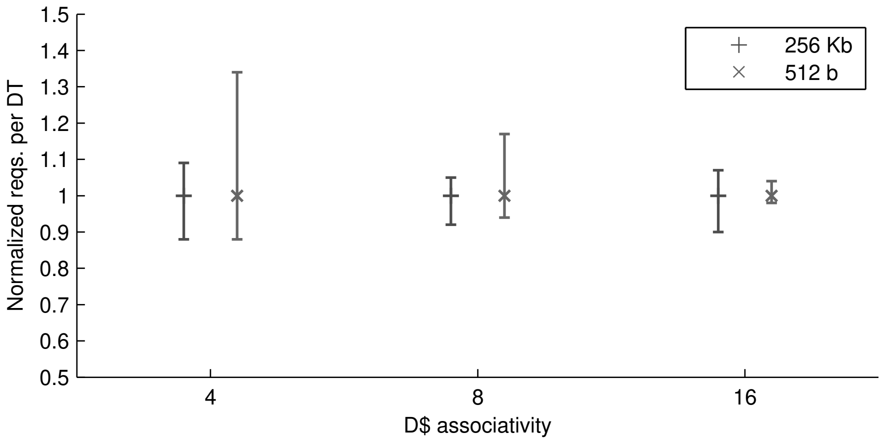


Figure 6.33: Directory requests per DT in Ocean con. for the Clupea architecture for directory load balancing based on 256 kilobyte and 512 byte address segments with 4, 8 and 16-way data cache associativity. The figure shows the max. and min. directory requests per DT normalized to the average requests per DT.

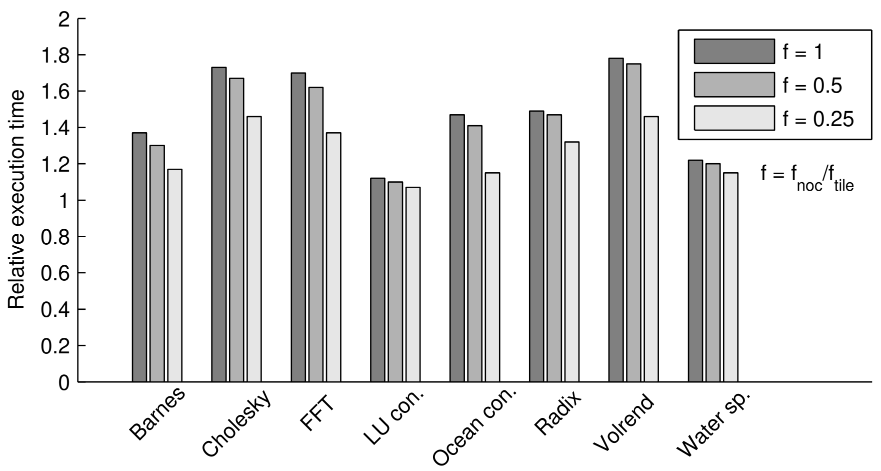


Figure 6.34: Relative execution time for the Clupea architecture with 16 ATs and four DTs compared to the HWC approach where the NoC clock frequency is divided by 2 and 4.

directory requests is lower. Overall, the relative execution time of the Clupea architecture drops from an average of 1.48 across all benchmarks to 1.27 when the NoC clock frequency is reduced to 0.25 of the processor core clock frequency.

6.5 Summary

This chapter has introduced an analytical model for early evaluation of shared memory many-core systems and the MC_sim simulator which allows detailed simulation that models the complex interaction between tiles in tile-based many-core architectures. Both models were used to evaluate the previously described cache coherent shared memory configuration of the Clupea architecture and compare it to a fixed hardware implementation of the same programming model support.

The analytical model provided early estimates of the overhead of the NIP compared to a direct hardware implementation showing a cache miss latency increase of 8 to 41% and an application execution time increase of 2% to 35%. Exploiting the flexibility of the Clupea architecture by handling private and shared data differently in the NIP showed potential for a significant reduction in the overhead and even reduction in application execution time for applications with a high rate of access to private data.

The MC_sim provided detailed simulation results for eight SPLASH2 benchmark applications on the Clupea architecture. Compared to a direct hardware implementation of the same programming model support, the Clupea architecture incurs an average relative execution time increase of 18% to 61% on average for 8 to 64 ATs compared to fixed hardware programming model support, however, overheads below 10% are observed for several benchmarks. The directory implementation is the main cause of this increase, however, scalability is ensured by allocating additional DTs. High and uneven DT loads lead to long queuing latency at the DTs. This latency can be addressed by reducing the DT latency or by reducing the arrival rate of directory requests.

The Clupea cache coherent shared memory configuration has been shown to scale up to 64 ATs with a relative execution time increase of less than 81% for configurations with 4 ATs per DT. Only load imbalance between the DTs is limiting the scalability. Experiments showed that better load balancing is achieved when the load balancing is based on 256 kilobyte address segments compared to 512 byte segments. Furthermore, experiments show that lower NoC clock frequencies reduce the Clupea relative execution time even further.

Compared to the analytical modeling results, MC_sim shows larger execution time overheads. This is due to lack of contention modeling in the analytical model. MC_sim simulations show that DT contention contributes by far the largest contribution to the cache miss latency. Thus, the relative execution time remains largely constant with the number of ATs instead of decreasing as estimated by the analytical model.

The modest execution time increase of as low as 18% on average for the Clupea architecture should be considered in relation to the flexibility that is offered. In the hardware approach a significant fraction of the tiles are fixed for the purpose of being cache coherence directories. In contrast, the Clupea architecture provides a platform of generic tiles that can be used for any purpose including support for programming models. While the Clupea architecture can not compete with fixed hardware support for cache coherent shared memory, it has the potential to exploit application-specific optimizations and allows the programmer to choose a suitable programming model for the application. Furthermore, the Clupea architecture is a valuable platform for common ground comparison of programming models.

Chapter 7

Future Research Directions

The work in this thesis has focused on support for the cache coherent shared memory programming model using configurable network interface processors in NoC-based many-core systems.

A major question that remains to be answered concerns how much can be gained by having configurable support for a plethora of programming models. Comparison of programming models is often difficult as evaluation platforms typically only have support for one programming model. Platforms with more generic support for programming models, such as the Clupea architecture, are needed for further studies and comparisons of programming models. Further studies of alternative programming models are necessary to answer this question. Message passing and data streaming programming models are good alternative candidates that avoid the contention caused by the directories in the cache coherent shared memory programming model. Thus, towards this end, the Clupea architecture offers an excellent reference platform for further studies of programming models.

Another interesting question that remains unanswered is the hardware implementation cost of the Clupea network interface. Increasing the complexity of the network interface and the NoC itself means fewer resources for processor cores and on-chip memories. On the other hand, having no programming model support in the network interface may lead to longer communication latencies and counter any advantages of having more processor cores or memories. As a result of this, the network interface design and support for programming models can be considered as a trade-off between efficiency and simplicity which also has to consider the need for supporting a plethora of programming models to support future applications. The Clupea architecture emphasises flexible support for programming models while keeping complexity in mind by having a single specialized network interface processor in each tile. Prototyping the architecture would give more insight into the costs and allow comparison to other approaches.

Generally, many-core systems are an emerging area where many questions needs to be answered and many aspects need to be investigated. One of them is how to design and program heterogeneous many-core architectures. In this direction one could look into using the programmable network interface processor in each tile to implement an abstraction layer between the processor core and the rest of the system. This flexibility could allow processor cores of different architecture to be fitted seamlessly together.

Another aspect is to look into operating systems for many-core architectures. Traditional multiprocessor capable operating systems are more or less extensions to operating systems that were originally designed for uniprocessor systems. This view is fundamentally changed in many-core systems where processor cores are an abundant resource and thus require strong emphasis on parallelism.

Chapter 8

Conclusions

This thesis has addressed the challenge of providing support for programming models in future many-core architectures with Network-on-Chip interconnects. Providing support for programming models is a non-trivial problem due to the large number of parallel programming models and the fact that none of these are suitable for all applications. Addressing this challenge has lead to three main parts of this thesis: i) A case study of parallelization of an image processing applications. ii) The Clupea many-core core architecture which offers configurable support for programming models. iii) Modeling and evaluation of the Clupea architecture using both analytical modeling and detailed simulation.

Parallelization of an Image Processing Application: The case study has considered parallelization and scalability issues for a potential embedded image processing application. The application has been parallelized using the OpenMP shared memory programming model and evaluated on a multiprocessor system to study the limitations of parallelization caused by the application and the execution platform. Results showed issues related to non-uniform memory access and cache utilization caused by lack of control of the distribution of tasks. These results inspired the configurable support for programming models in the Clupea architecture.

The Clupea Many-core Architecture: The Clupea architecture has been presented as a flexible tile-based many-core architecture with configurable support for programming models. The architecture features specialized programmable network interface processors which enable individual support for programming models for each application that executes on the system. The network interface processor architecture has been designed with the limited on-chip hardware resources in mind, which distinguishes the Clupea architecture from previous multiprocessor architectures. Tight integration of the network interface processor into the local caches allows the memory system to be fully configurable. The Clupea is not intended to, and can not, compete with fixed hardware support for programming models. Instead, it provides a flexible hardware platform that can exploit application-specific characteristics and allow the programmer to choose a suitable programming model for the particular application. Possible implementations of a selection of programming models using the Clupea architecture have been discussed and a detailed implementation of cache coherent shared memory has been presented.

The Clupea cache coherent shared memory implementation shows a highly flexible implementation of a directory-based cache coherence protocol. Directories are implemented using generic processing tiles and take advantage of both the network interface processor and a general purpose processor. Using generic processor tiles as directories allows the cache coherent shared memory implementation be easily matched to demands of the application and scale to a large number of tiles by allocating more tiles to implement directories. Thus, hardware resources are not wasted on unnecessary support for programming models in the Clupea architecture.

Modelling and Evaluation: To aid the evaluation of many-core architectures, two models have been developed which complements each other by offering early analytical modeling and thorough cycle accurate simulation respectively. The analytical model offers valuable early estimates based on a small set of architectural model parameters that are obtainable early in the design process. In contrast, the MC_sim simulator offers cycle accurate modeling of the memory system and interconnect which captures contention and occupancy. The trace-driven simulation approach enables fast simulation which is essential for thorough analysis of many-core architectures. Detailed modeling of the cache coherence protocol and extensive simulation statistics enable detailed analysis of the memory system.

Evaluation of the Clupea architecture showed modest execution increases for a range of parallel benchmark applications when compared to fixed hardware support for the cache coherent shared memory programming model. MC_sim simulation has shown execution time increases ranging from 4% to 81% for different configurations of the programming model support for many-core systems with 8 to 64 processor tiles executing the benchmark applications. Request queuing at the directories due to contention has been identified as the main contributor to this increase. Evaluation of several optimizations of the directory implementation have shown that this overhead is generally hard to reduce due to the behavior of the applications. However, for most benchmarks the effect can be significantly reduced by adding more directories to distribute the load. Generally, the relative execution time increases only slightly as the number of processor tiles increases. The Clupea shared memory implementation is sensible to the data cache miss rate, thus the execution times can be significantly reduced by increasing the data cache size and associativity. Overall, the execution times on the Clupea architecture shows that configurability has a price when comparing support a single programming model, however, the architecture offers a good potential for application-specific optimizations. Thus, major improvements are expected if the Clupea architecture is combined with software tools that can take advantage of its configurability.

Bibliography

[1] K. Goossens A. Hansson and A. Radulescu. Analysis of message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007.

[2] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A two-level directory architecture for highly scalable cc-numa multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(1):67–79, 2005.

[3] Advanced Micro Devices Inc. *Family 10h AMD Phenom II Processor Product Data Sheet*, April 2010. http://www.intel.com.

[4] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[5] S.V. Adve, V.S. Adve, M.D. Hill, and M.K. Vernon. Comparison of hardware and software cache coherence schemes. *Int. Symp. on Computer Architecture*, 1991.

[6] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: architecture and performance. In *ISCA '95: Proceedings of the 22nd annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA, 1995. ACM.

[7] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. *SIGARCH Comput. Archit. News*, 16(2):280–298, 1988.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view

from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[9] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 282–293, New York, NY, USA, 2000. ACM.

[10] L. Benini and G. De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, 2002.

[11] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. Noc synthesis flow for customized domain specific multiprocessor systems-on-chip. *Parallel and Distributed Systems, IEEE Transactions on*, 16(2):113 – 129, feb. 2005.

[12] P. Bhojwani and R. Mahapatra. Interfacing cores with on-chip packet-switched networks. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 382 – 387, jan. 2003.

[13] T. Bjerregaard, S. Mahadevan, R. G. Olsen, and J. Sparsø. An OCP compliant network adapter for GALS-based soc design using the MANGO network-on-chip. In *Proceedings of the International Symposium on System-on-Chip (SoC'05)*, pages 171–174. IEEE, nov 2005.

[14] T. Bjerregaard and J. Sparsø. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1226–1231, Washington, DC, USA, 2005. IEEE Computer Society.

[15] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny. The power of priority: Noc based distributed cache coherency. In *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, pages 117–126, Washington, DC, USA, 2007. IEEE Computer Society.

[16] J. A. Brown, R. Kumar, and D. Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 126–134, New York, NY, USA, 2007. ACM.

[17] D. Chaiken, J. Kubiatowicz, and A. Agarwal. Limitless directories: A scalable cache coherence scheme. *SIGPLAN Not.*, 26(4):224–234, 1991.

[18] X. Chen, Z. Lu, A. Jantsch, and S. Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *DATE '10: Proceedings of the conference on Design, automation and test in Europe*, 2010.

[19] L. H. Clemmensen, M. E. Hansen, J. C. Frisvad, and B. K. Ersbll. A method for comparison of growth media in objective identification of penicillium based on multi-spectral imaging. *Journal of Microbiological Methods*, 69(2):249 – 255, 2007.

[20] F. Clermidy, R. Lemaire, Y. Thonnart, and P. Vivet. A communication and configuration controller for noc based reconfigurable data flow architecture. In *NOCS '09: Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 153–162, Washington, DC, USA, 2009. IEEE Computer Society.

[21] UPC Consortium. Upc language specifications, v1.2. Technical report, Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005. http://upc.gwu.edu.

[22] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann, August 1998.

[23] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor socs. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, page 536, Washington, DC, USA, 2003. IEEE Computer Society.

[24] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[25] W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnection networks. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 684–689, New York, NY, USA, 2001. ACM.

[26] J. Duato, S. Yalamanchili, and N. Lionel. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[27] A. Duran, M. Gonzalez, and J. Corbalan. Automatic thread distribution for nested parallelism in openmp. *Proceedings of the International Conference on Supercomputing*, pages 121–130, 2005.

[28] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *International Workshop on Language Runtimes, OOPSLA*, 2004.

[29] R. Esser and R. Knecht. Intel paragon xp/s - architecture and software enviroment. In *Supercomputer '93: Anwendungen, Architekturen, Trends, Seminar*, pages 121–141, London, UK, 1993. Springer-Verlag.

[30] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1.3 edition, 1995. http://www.mcs.anl.gov/∼itf/dbpp/.

[31] P. Francesco, P. Antonio, and P. Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 736 – 741 Vol. 2, March 2005.

[32] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and design of alphaserver gs320. *SIGPLAN Not.*, 35(11):13–24, 2000.

[33] K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, Sept-Oct 2005.

[34] K. Goossens, A. Radulescu, and A. Hansson. A unified approach to constrained mapping and routing on network-on-chip architectures. *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, pages 75–80, 2005.

[35] H. Grahn and P. Stenström. Efficient strategies for software-only protocols in shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 23(2):38–47, 1995.

[36] S. Han, A. Baghdadi, M. Bonaciu, S. Chae, and A. A. Jerraya. An efficient scalable and flexible data transfer architecture for multiprocessor soc with massive distributed memory. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 250–255, New York, NY, USA, 2004. ACM.

[37] 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.

[38] A. Hansson and K. Goossens. An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 99–108, New York, NY, USA, 2009. ACM.

[39] N. Hardavellas, M. Ferdman, B. Falsafi, and A Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *ISCA '09: Proceedings of the 36th annual International Symposium on Computer architecture*, pages 184–195, New York, NY, USA, 2009. ACM.

[40] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The performance impact of flexibility in the stanford flash multiprocessor. In *Proc. of ASPLOS-VI*, pages 274–285, New York, NY, USA, 1994. ACM.

[41] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[42] T. Henriksson and P. van der Wolf. Ttl hardware interface: A high-level interface for streaming multiprocessor architectures. In *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 107–112, Washington, DC, USA, 2006. IEEE Computer Society.

[43] W. D. Hillis and L. W. Tucker. The cm-5 connection machine: a scalable supercomputer. *Commun. ACM*, 36(11):31–40, 1993.

[44] M. A. Holliday. Techniques for cache and memory simulation using address reference traces. *Int. J. Comput. Simul*, 1:129–151, 1990.

[45] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: memory performance feedback mechanisms and their applications. *ACM Trans. Comput. Syst.*, 16(2):170–205, 1998.

[46] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, 7-11 2010.

[47] Intel Corp. *Pin*, 2009. http://www.pintool.org.

[48] Intel Corp. *Intel Core i7-900 Desktop Processor Extreme Edition Series and Intel Core i7-900 Desktop Processor Series - Datasheet*, february 2010. http://www.intel.com.

[49] ITRS. International techonolgy roadmap for semiconductors - system drivers, 2009. http://www.itrs.net.

[50] J. Jaehyuk Huh, C. Changkyu Kim, H. Shafi, L. Lixin Zhang, D. Burger, and S.W. Keckler. A NUCA substrate for flexible CMP cache sharing. *IEEE Trans. on Parallel and Distributed Systems*, 18(8), 2007.

[51] A. Jalabert, S. Murali, L. Benini, and G. De Micheli. ×pipescompiler: A tool for instantiating application specific networks on chip. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20884, Washington, DC, USA, 2004. IEEE Computer Society.

[52] A. A. Jerraya, A. Bouchhima, and F. Pétrot. Programming models and hw-sw interfaces abstraction for multi-processor soc. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 280–285, New York, NY, USA, 2006. ACM.

[53] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[54] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd. Power7: Ibm's next-generation server processor. *Micro, IEEE*, 30(2):7 –15, March-April 2010.

[55] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *SIGARCH Comput. Archit. News*, 37(3):140–151, 2009.

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

[57] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGPLAN Not.*, 37(10):211–222, 2002.

[58] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 162–173, New York, NY, USA, 2007. ACM.

[59] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *ISCA '94: Proceedings of the 21st annual International Symposium on Computer Architecture*, pages 302–313, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[60] S. Kwon, Y. Kim, W. Jeun, S. Ha, and Y. Paek. A retargetable parallel-programming framework for mpsoc. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–18, 2008.

[61] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *Computers, IEEE Transactions on*, C-28(9):690 –691, Sept. 1979.

[62] J. Laudon and D. Lenoski. The sgi origin: a ccnuma highly scalable server. *SIGARCH Comput. Archit. News*, 25(2):241–251, 1997.

[63] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[64] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: implementation and performance. In *ISCA '92: Proceedings of the 19th annual International Symposium on Computer Architecture*, pages 92–103, New York, NY, USA, 1992. ACM.

[65] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. *Int. Symp. on Computer Architecture*, 2007.

[66] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Comput. Surv.*, 25(3):303–338, 1993.

[67] J. D. C. Little. A proof for the queuing formula: L= w. *Operations Research*, 9(3):383–387, 1961.

[68] P. Magarshack and P. G. Paulin. System-on-chip beyond the nanometer wall. *Proceedings - Design Automation Conference*, pages 419–424, 2003.

[69] R. Marculescu, U.Y. Ogras, Li-Shiuan Peh, N.E. Jerger, and Y. Hoskote. Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(1):3 –21, jan. 2009.

[70] G. Martin. Overview of the mpsoc design challenge. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 274–279, New York, NY, USA, 2006. ACM.

[71] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4), 2005.

[72] M. M. Michael and A. K. Nanda. Design and performance of directory caches for scalable shared memory multiprocessors. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 142, Washington, DC, USA, 1999. IEEE Computer Society.

[73] M. M. Michael, A. K. Nanda, and B. Lim. Coherence controller architectures for scalable shared-memory multiprocessors. *IEEE Trans. Comput.*, 48(2):245–255, 1999.

[74] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. An efficient synchronization technique for multiprocessor systems on-chip. *SIGARCH Comput. Archit. News*, 34(1):33–40, 2006.

[75] S. Murali, L. Benini, and G. de Micheli. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference 2005 (IEEE Cat. No.05EX950C)*, (Vol. 1):27–32 Vol. 1, 2005.

[76] S. Murali and G. De Micheli. Bandwidth-constrained mapping of cores onto noc architectures. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20896, Washington, DC, USA, 2004. IEEE Computer Society.

[77] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Bus, K. Goossens, R. Peset Llopis, and P. Lippens. C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.

[78] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *ISCA '90: Proceedings of the 17th annual International Symposium on Computer Architecture*, pages 138–147, New York, NY, USA, 1990. ACM.

[79] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, 1996.

[80] Open Handset Alliance. *Android SDK*, May 2010. http://code.google.com/android/.

[81] OpenMP Architecture Review Board. *OpenMP Application Program Interface 2.5*, 2005. http://www.openmp.org.

[82] OpenMP Architecture Review Board. *OpenMP Application Program Interface 3.0*, 2008. http://www.openmp.org.

[83] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonnard, B. Lavigueur, and D. Lo. Distributed object models for multi-processor soc's, with application to low-power multimedia wireless systems. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 482–487. European Design and Automation Association, 2006.

[84] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu. Parallel programming models for a multi-processor soc platform applied to high-speed traffic management. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 48 – 53, sept. 2004.

[85] F. Poletti, A. Poggiali, D. Bertozzi, L. Benini, P. Marchal, M. Loghi, and M. Poncino. Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support. *Computers, IEEE Transactions on*, 56(5):606 –621, may 2007.

[86] A. Radulescu, J. Dielissen, S.G. Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):4 – 17, jan. 2005.

[87] M. S. Rasmussen, S. Karlsson, and J. Sparsø. Adaptable support for programming models in many-core architectures. In *Workshop on New Directions in Computer Architecture*, 2009.

[88] M. S. Rasmussen, S. Karlsson, and J. Sparsø. Performance analysis of a hardware/software-based cache coherence protocol in shared memory mpsocs. In *Programming Models for Emerging Architectures*, 2009.

[89] M. S. Rasmussen, M. B. Stuart, and S. Karlsson. Parallelism and scalability in an image processing application. In *IWOMP*, pages 158–169, 2008.

[90] M. S. Rasmussen, M. B. Stuart, and S. Karlsson. Parallelism and scalability in an image processing application. *International Journal of Parallel Programming*, 37(3):306–323, 2009.

[91] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: user-level shared memory. In *ISCA '94: Proceedings of the 21st annual International Symposium on Computer Architecture*, pages 325–336, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[92] E. Salminen, A. Kulmala, and T. D. Hämäläinen. Survey of network-on-chip proposals. Technical report, OCP-IP, March 2008.

[93] H. S. Sandhu and K. C. Sevcik. An analytic study of dynamic hardware and software cache coherence strategies. *Performance Evaluation Review*, 23(1):167–177, 1995.

[94] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *ISCA '03: Proceedings of the 30th annual International Symposium on Computer Architecture*, pages 422–433, New York, NY, USA, 2003. ACM.

[95] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[96] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 667–672, New York, NY, USA, 2001. ACM.

[97] S. Srbljic, Z.G. Vranesic, M. Stumm, and L. Budin. Analytical prediction of performance for cache coherence protocols. *IEEE Transactions on Computers*, 46(11):1155–1173, 1997.

[98] M. B. Stensgaard and J. Sparsø. Renoc: A network-on-chip architecture with reconfigurable topology. In *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 55–64, Washington, DC, USA, 2008. IEEE Computer Society.

[99] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12 –24, jun 1990.

[100] T. Suh, D. Kim, and H.-H.S. Lee. Cache coherence support for non-shared bus architecture on heterogeneous mpsocs. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 553 – 558, june 2005.

[101] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[102] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, New York, NY, USA, 2004. ACM.

[103] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29 –41, jan. 2008.

[104] Virtutech. *Simics*, 2009. http://www.virtutech.com.

[105] W. Wolf. *Modern VLSI Design: System-on-Chip Design.* Prentice Hall Press, Upper Saddle River, NJ, USA, third edition, 2002.

[106] W. Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 681–685, New York, NY, USA, 2004. ACM.

[107] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[108] Z. Yu, M. J. Meeuwsen, R. W. Apperson, O. Sattari, M. A. Lai, J. W. Webb, E. W. Work, T. Mohsenin, and B. M. Baas. Architecture and evaluation of an asynchronous array of simple processors. *J. Signal Process. Syst.*, 53(3):243–259, 2008.

[109] H. Zeffer, Z. Radović, M. Karlsson, and E. Hagersten. Tma: a trap-based memory architecture. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 259–268, New York, NY, USA, 2006. ACM.

[110] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.