# Conservative application-level performance analysis through simulation of a multiprocessor system on chip

Nelson, A.T.

*Award date:*
2009

# Conservative Application-Level Performance Analysis through Simulation of a Multiprocessor System on Chip

Andrew T. Nelson

24th March 2009

**Abstract**

Real time applications require temporal guarantees to ensure the validity of their output. Firm RT applications must meet their temporal requirements otherwise the validity of their output sharply decreases. For these applications formal models are used to analytically calculate bounds on temporal behaviour. The validity of early, or late, output from soft RT applications does not decrease as sharply after the deadline has been missed. These applications are often more complicated to model and do not adhere to equally strict programming models, e.g. input dependent application execution. Running RT applications on MPSoCs only complicates the issue further. Individual application tasks may be mapped to different cores complicating the modelling of the RT application further. Simulation of RT applications on MPSoCs offers an alternative to formally modelling RT applications. In this thesis the application of a conservative simulation technique to provide application-level performance guarantees in network-based SoC, is demonstrated. The approach is application independent allowing its application to any deterministically executing application that can run on the system. Conservative guarantees are produced on a per execution trace basis. Different execution traces may be produced by inputting different datasets. The approach is verified through the application of the technique to artificial test-case applications along with a real-life application in the form of a JPEG Decoder. The simulation results of the test cases are compared with an FPGA synthesised instance of an MPSoC system to show that the model does facilitate conservatively timed simulation.

# Contents

# Chapter 1

# Introduction

Soft Real Time applications such as a video decoder have temporal deadlines that need to be met to maintain smooth playback. A combined formal model of the application and the hardware to which it is mapped facilitates per trace analytical performance analysis. Creating formal models can be a complex and time consuming task for all but the simplest of applications. In a measure-modify design cycle, such as that illustrated in Figure 1.1a, time between iterations may be relatively slow.

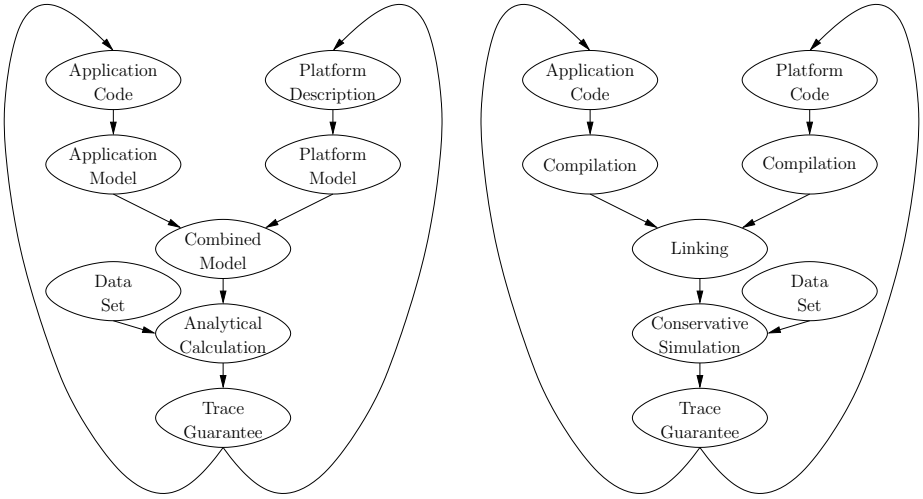
In this thesis conservative simulation is proposed as an alternative to formal modelling for application-level performance analysis. In a measure-modify design cycle, such as that illustrated in Figure 1.1b, modifications to the application and hardware description only need to be compiled, linked and simulated to generated per trace guarantees. This allows the designer to focus on tweaking the application and hardware platform without having to worry about how to formally model it.

In this Chapter a detailed introduction to this thesis work is given in Section 1.1. In Section 1.2 an overview of work related to this thesis and where this thesis fits in is given. The hardware architecture that is used throughout this thesis follows a predictable hardware template. The specific configuration of the hardware platform is presented in Section 1.3. An overview of the contributions made in this thesis is presented in Section 1.4 along with an overview of the contents of the rest of this thesis.

## 1.1 Introduction

Real Time applications have temporal constraints to adhere to. The rigidity of this adherence can be defined subjectively as a firm or soft constraint [3]. Firm RT applications must meet their deadlines or the validity of their output will be considerably devalued, e.g. a VOIP application outputting "garbled" speech. Soft RT applications can miss some deadlines without such a steep devaluation of their output, e.g. a video decoder dropping frames.

Some applications composed of tasks mapped on a multiprocessor system can be modelled as Cyclo-Static Dataflow (CSDF) Graphs [2] to facilitate analytical calculation of performance guarantees that are independent of the input data through the application of dataflow analysis techniques [8]. The technique




(a) Per trace guarantees that are analytically calculated from CSDF graphs.

(b) Per trace guarantees that are generated through conservative simulation.

Figure 1.1: Platform based design flows for formal modelling and simulation.

models the application, interconnect and mapping combined, capturing the entire system in a single analytical model. Due to the restricted applicability of CSDF graphs, this technique only applies to a small subset of applications. A design flow for this technique is shown in Figure 1.1a. The hardware platform is modelled as a CSDF graph as described in [8]. The application must also be modelled as a CSDF graph, which is potentially a complex and time consuming task if it is possible at all in some instances. The hardware and application models are combined facilitating per trace performance analysis, which is also a potentially complex and time consuming task. The analytically calculated performance guarantee for a trace is used to make decisions on application and hardware tweaks.

In this thesis conservative application-level performance analysis through simulation of a MPSoC is proposed as an alternative to formal modelling of RT applications. Applications that are deterministic in execution may be compiled and simulated to produce per trace guarantees. Non-determinism occurs through the application execution depending on random numbers, synchronisation with non deterministic hardware components and clock time. The non-determinism effects the trace through the application execution making it impossible to give guarantees through simulation. The conservative simulation of a MPSoC that will be described in this thesis is a combination of cycle-accurate simulation and Latency-Rate (LR) modelling. Cycle-accurate simulation is used to model the IP-cores and parts of the NoC, while LR servers are used to conservatively model the run-time arbitrated components as described in [14]. Guarantees can subsequently be given on a per trace basis for applications that are deterministic in execution. A trace in this instance is defined as a unique execution path through the application's program instructions.

A predictable hardware design template is described in [7] that has been followed in the creation of the MPSoC implementation that is used in this the-

sis. Components are combined at the IP level to create a predictable system. The computational cores are simple VLIW processors. The cores are stripped of components that effect the predictability of their execution. There are no optimisations such as out of order execution of instructions or memory caching. Parallelisation of instructions is achieved in the construction of the VLIW instructions by the application compiler. Components are connected via an Æthereal NoC that is configured to provide guaranteed service. In this configuration bounds on latency and throughput are provided for messages traversing the NoC and for memory arbitration. Details of the MPSoC system can be found in Section 1.3.

The SDK from Silicon Hive [13] is used as a platform for the modelling and simulation of the MPSoC system. The Silicon Hive SDK has the ability to create and simulate multiprocessor systems on chip. Simulation of the processors is carried out through cycle accurate Instruction Set (IS) simulation. Using the Silicon Hive SDK's simulation as a starting point a NoC model is implemented that conservatively abstracts away from the fine detail of the NoC's mechanics while still capturing the NoC's behavioural details. Non-runtime arbitrated components are modelled cycle accurately as a delay. Run time arbitrated components are modelled conservatively as Latency Rate (LR) servers. LR servers are a method of generalising the timing characteristics of a run time arbitrated component into two values representing the sustained Rate that the component can maintain and the Latency until it can conservatively maintain that Rate. This reduces the complexity of realising the component in a model for simulation, instead shifting the responsibility to derivation of the LR values.

## 1.2 Related Work

A lot of research has been carried out in the last few years on the application-level modelling and simulation of MPSoCs. A common thread has been the abstraction from RTL-level simulation in order to speed up simulation times. The variability of the abstraction approach has lead to many different MPSoC simulation frameworks being developed.

In [1] a cycle accurate MPSoC simulator called MP-ARM is described. The MP-ARM platform uses SystemC as its modelling and simulation environment. The main focus of the work is on providing a complete platform for MPSoC research, e.g. exploring the MPSoC design space. Completeness in this sense is an MPSoC platform where all the IP-level components are simulated, a fully operational OS port and code development tooling. This facilitates the simulation of applications that were compiled for the ported OS.

A RTOS is used for scheduling in the ARTS MPSoC platform simulation framework described in [10]. Unlike in [1] where the application could be compiled and run on the OS, in [10] applications first need to be modelled as dataflow graphs before they can be simulated. In [10] it does not state how the timing values for the dataflow graphs are calculated. The work in this thesis avoids the necessity for the potentially complex and time consuming task of formal modelling of the application. Applications in this work can simply be mapped to the hardware, compiled, linked and simulated to provide per trace guarantees.

While not specifically aimed at MPSoC simulation, it is described in [9] how a hybrid simulation model can be implemented using a combination of ISS and

abstract RTOS. The RTOS is abstracted as a systemC module that calculates the schedule to be executed by the ISS. In order to provide timed performance analysis the RTOS model is annotated with average timings. The simulation timings can therefore not be guaranteed to be conservative, as is bore out by the simulation results in [9]. An OS or RTOS is not used for the work in this thesis. Instead a fast measure and redesign cycle, illustrated in Figure 1.1b, facilitates mapping an application directly to the hardware.

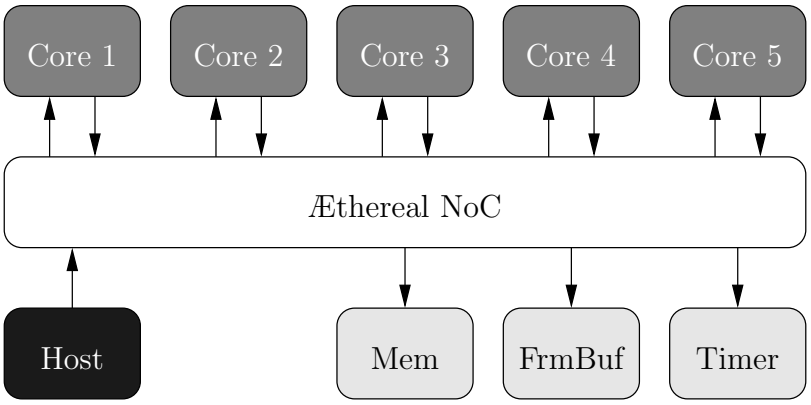
The work in [14] shows that it was possible to conservatively model run-time arbitration as Latency Rate servers and incorporate the Latency Rate servers in a dataflow graph. In [8] a formal conservative application-level MP-SoC modelling method is proposed, using CSDF graphs to model applications on a predictable SoC. The formal model allows the analytical calculation of temporal bounds, on a per-trace basis, for the application on the SoC. In [8] it is described how individual Æthereal NoC connections can be represented as two channels abstractly modelled as Latency Rate servers.

In this thesis simulation is proposed as an alternative to formal modelling for conservative application-level performance analysis. A connection model using the principles of the two channel connection model in [8] is used to conservatively model transaction times in simulation. Runtime arbitrated components are modelled as Latency Rate (LR) servers as described in [14]. This thesis demonstrates that the method to calculate the Latency component of the LR values for TDMA arbitration described in [14] is overly conservative for TDMA arbitration tables that do not provide service in a continuous block. An algorithmic method is described in this thesis that facilitates the derivation of shorter yet conservative Latency values.

## 1.3 Hardware Architecture

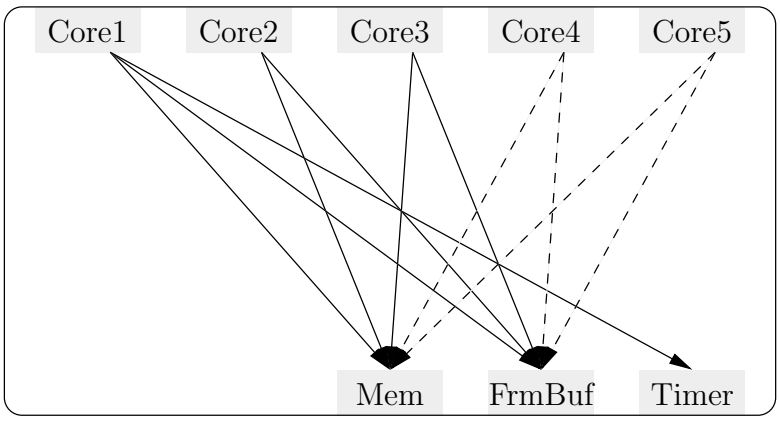
The hardware architecture of the system that is used throughout this thesis follows the predictable system architecture template from [7]. The work in this thesis is applicable to the template and not just the system used for the work in this thesis. The system used for the work in this thesis consists of 5 processing cores, some shared memory, a frame buffer, and a timer peripheral, as illustrated in Figure 1.2a. The processing cores are simple VLIW processing cores without features that effect determinism such as memory caching or out of order instruction execution. Silicon Hive [12] Pearlray processing cores are used as they satisfy these requirements. The cores contain some internal memory, program memory and 3 issue slots. Instruction level parallelism is achieved in the compiler. This makes the complexity of optimising the code execution a compiler, instead of a processor task. The instructions execute on the cores in a deterministic manner. In software the cores are modelled using Instruction Set Simulation (ISS) to provide cycle accurate simulation. Internal memory access times are taken into account in the ISS. One of the issue slots and the Pearlray's slave port have access to the internal memory. The arbitration time of any contention is not taken into account by the ISS. As such the case studies in this thesis avoid contention by not providing cores direct access to the internal memory of other cores, and by the host not interacting with the cores internal memory during the timing of the tasks.

The memory (*Mem*) and framebuffer (*FrmBuf*) are shared memory mapped




| FPGA | SIM |
|---|---|
| x86 Code + HRT API | x86 Code + HRT API |
| HDL + Control Code | HSS API + LR Abstraction |
| HDL + Program Code | ISS + Program Code |
| HDL | HSS API |

(a) System hardware configuration.




(b) The NoC Connections use case used for the work in this thesis. All arrows represent NoC Connections. All arrows depict the Master to Slave relationship. Dashed arrows indicate the extra connections that are created by the 5 core JPEG case study.

Figure 1.2: System Configuration.

data storage locations. *Mem* is a shared memory location that can be read and written to making it suitable for inter-core communication, as well as storing relatively large datasets that do not fit in the core's internal memory. *FrmBuf* is a shared memory location that can be written to by the cores but is read from by a display device, connected to the FPGA board. Both of these components are modelled in software using the provided Silicon Hive toolset memory devices.

The provided Silicon Hive toolset memory devices do not model access times. Memory accesses take zero time. The simulator therefore does not perform memory arbitration as concurrent memory accesses are possible at any instance. Memory access times and arbitration are taken into account in the connection model, as described in Section 2.7.

The *Timer* device is a 32 bit counter that increments every clock cycle. The counter is accessible on a memory mapped address allowing the clock to be read and reset. For software simulation the functionality of this device was recreated using the Silicon Hive Hive System Simulator (HSS) API to make a custom device that could be integrated into the system.

The Æthereal NoC [5] provides a predictable interconnect for the inter-IP communication. The Æthereal NoC is predictable as bounds can be given on latency and throughput of transactions across the NoC. In Chapter 2 details are given as to how the Æthereal can be conservatively modelled for use in conservative application-level performance analysis through simulation of a MPSoC.

## 1.4 Contributions & Thesis Overview

The rest of this thesis is organised as follows. In Chapter 2 it is explained how the connection model from [7] can be modified to model an Æthereal NoC [5] connection. A connection model is contributed that uses a combination of cycle accurate simulation, and LR abstraction for runtime arbitrated components. A brief explanation of the principles of LR abstraction is given along with the mathematical theory to calculate LR values for the Æthereal NoCs TDMA routing tables. It is shown that the analytical LR derivation method is unnecessarily over conservative. An algorithmic method to calculate a tightly conservative Latency component is contributed that uses the principles of the analytical method.

In Chapter 3 it is explained how the theory of modelling the Æthereal NoC, as described in Chapter 2, is implemented as a NoC model system component that can be integrated into the virtual hardware platform. The implementation of the Æthereal NoC model is also a contribution of this thesis. In Chapter 3 it is demonstrated that application level conservativeness is not automatically conferred, for application level inter-IP synchronisation in shared memory, even when the hardware level is conservatively modelled. An overview of the Silicon Hive SDK is also given in Chapter 3. The Silicon Hive SDK is used to model the hardware platform and provide the simulation environment, in which the application is simulated on the virtual hardware platform. A detailed description of how the Æthereal NoC model is actually implemented. A description is also given as to how the model operates to conservatively model NoC transactions. It is demonstrated that application level conservativeness, for inter-IP synchronisation in shared memory, is not obtained solely by conservatively modelling the hardware level. A method is demonstrated in this thesis, to conservatively model the inter-IP synchronisation in shared memory through the use of a communication library. The rationality behind how the implemented model can conservatively bound the implementation through simulation is also explained.

In Chapter 4 the evaluation of conservative simulation through the use of case studies is contributed. Artificial applications, and a real life application in the form of a JPEG decoder, are simulated on the virtual hardware system.

Results from the simulations are compared with results from running the applications, on an FPGA synthesised version of the hardware system. Results are also compared to a virtual hardware system where NoC transactions and memory accesses are instantaneous, allowing the contribution of the NoC timings to be isolated from the other results. All simulations are carried out using both methods of LR value calculation. Individual Æthereal NoC connection tests are carried out where tests are performed on load and store transactions to and from external memories, from one of the cores. The aim of the connection tests is to isolate the behaviour of the different transaction types so that they may be analysed. In another case study the communication library is used in a streaming application. The simple application streams data from the first core, via the second core to the third core using the C-HEAP [11] communication protocol. The effect of application-level synchronisation granularity is investigated in this case study.

A JPEG decoder application is used to illustrate the applicability of the conservative simulation technique on a real life application. The JPEG decoding application is mapped onto the hardware system, conservatively simulated and the results analysed in comparison to an FPGA implementation of the system. In a separate case study the JPEG decoding application is mapped onto the hardware system based on a theoretical NoC connection use case. This case study illustrates the flexibility of using simulation to produce conservative timing analysis.

In Chapter 5 concluding statements are made relating to the contents of this thesis. Proposals are made for possible future applications of the simulation technique described in this thesis. Future modifications are also proposed to increase the functionality of the technique.

# Chapter 2

# Modelling the Æthereal NoC

NoCs such as Æthereal described in Section 2.2 offer bounds on latency and throughput of transactions sent across the network. In [8] it is described how to extend the end-to-end guarantees of the NoC beyond the NoC's Network Interfaces (NI) to incorporate the IPs. This is achieved by modelling individual NoC connections as Cyclo Static Dataflow (CSDF) graphs [2] and conservatively modelling run-time arbitration components as Latency-Rate (LR) servers [4, 14]. Applications that can be described as CSDF graphs, could then be conservatively bounded through the analytical calculation of worst-case timings on a per execution trace basis.

In the method described in this thesis, connections are modelled as a combination of cycle accurate representation of components and LR servers for run-time arbitrated components in order to reduce the level of abstraction and therefore increase accuracy while maintaining conservativeness. The NoC model, when integrated into a MPSoC simulation environment, facilitates the conservative simulation of applications that exhibit deterministic execution, i.e. The execution path or trace does not depend on random numbers, synchronisation with non-deterministic hardware or the clock time.

In this Chapter a connection model is contributed that uses a combination of cycle accurate and Latency Rate abstraction. This model is described in Section 2.1. The Æthereal NoC is examined in Section 2.2, in particular how Æthereal NoC connections can be modelled independently. Section 2.4 explains how the producing NIs can be modelled using various degrees of abstraction. The principles of Latency Rate abstraction are explained in Section 2.3. Section 2.5 describes how LR values can be calculated analytically from Æthereal TDMA arbitration tables. It is shown in Section 2.6 that the LR values derived using the analytical method can be unnecessarily overly conservative for some TDMA tables and demonstrates an algorithmic method for LR value derivation that produces tight conservative LR values. The LR abstraction of the Slave side memory arbitration and memory access is described in Section 2.7.

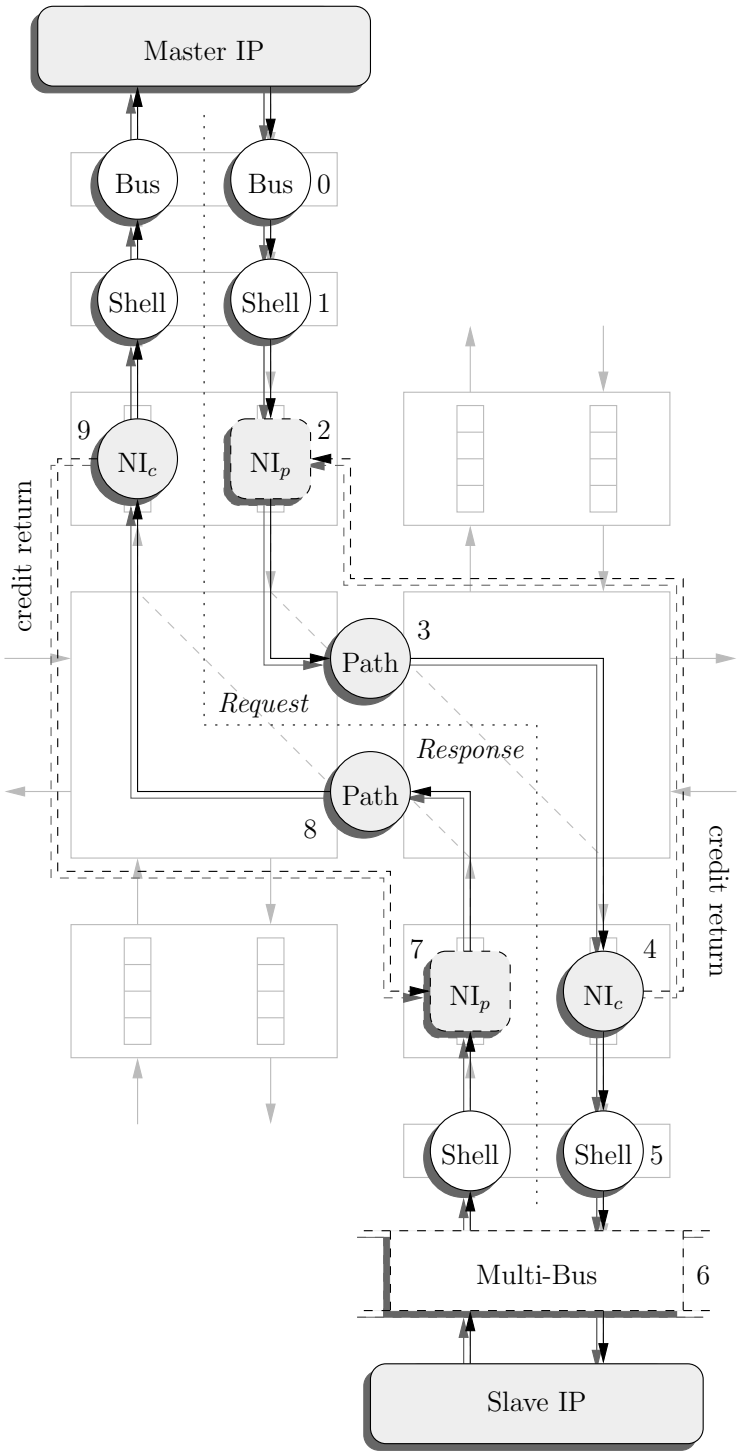



Figure 2.1: Example connection through contention free routing using pipelined time-division-multiplexed circuit switching. The numbered sections cause a delay and therefore need to be modelled for simulation.

## 2.1 Connection Model

Æthereal NoC connections provide guaranteed bounds on latency and throughput. These bounds are guaranteed regardless of network load allowing individual connections to be modelled independently. Figure 2.1 illustrates a model that encapsulates the behaviour of an Æthereal connection, using a combination of cycle accurate and abstract LR component representation. Circular nodes of the graph represent components that exhibit cycle accurate latency compared to their real life counterparts. The latency is derived from the components operational characteristics. Nodes with dashed outlines represent components with run-time arbitration that are LR abstracted. The LR abstraction in the NIs models the TDMA arbitration tables, as explained in Section 2.6. The multibus LR abstraction is explained in Section 2.7.

The Master IP initiates *load/read* or *store/write* transactions. *Load* and *store* transactions can be visualised as flowing through the model, along the request path, from the Master IP to the Slave IP. In the case of *load* transactions the fetched data can be visualised as flowing through the model, along the response path, from the Slave IP to the Master IP. In the implementation of the Æthereal NoC used for the work in this thesis, a phit is equal in size to one word and only one word of data may be transferred at a time. Word sized tokens are therefore used as a unit of transfer with the connection model in Figure 2.1.

Transactions are put onto the *bus* on the *request* channel, represented as section 0, by the master port. The *bus* decodes the address and routes the transaction request to the appropriate shell. This takes 2 cycles in the HDL implementation so it is represented as a 2 cycle latency.

The master side *shell*, represented as section 1, adds the necessary message headers for the encapsulation of the bus transactions. The action of the *shell* depends on the transaction type. A *load* transaction carries no data payload on the request path so is encapsulated as 2 phit sized message headers. A *store* transaction carries data on the request path so is encapsulated as the data preceded by 2 phit sized message headers. The first header leaves the *shell* 1 cycle after the transaction arrives, if there is space in the producing NI buffer. The subsequent header, and possible data, leave every cycle after that. E.g. A *load* arrives at the *shell* at cycle 0, with enough space in the producing NI buffer for the *shell's* output. Two message headers are produced leaving the *shell* at 1 and 2 cycles. A *store* arrives under similar circumstances. Two message headers are produced in the same manner followed by the data at 3 cycles.

The producing *NIs*, represented as sections 2 and 7, queue the message phits in a limited sized buffer while awaiting scheduling for transmission over the NoC. Run-time arbitration is carried out through the use of TDMA tables. The abstraction of this process through the use of LR servers is explained in Section 2.6.

The *paths*, represented by sections 3 and 8, represent the delay caused by the number of routers that the phit passes through en route to its destination. Each hop delays the phit 3 cycles.

The consuming *NIs*, represented by sections 4 and 9, also contain a limited size buffer to store phits until they can be passed into the slave side *shell*. This is represented by a latency of 2 cycles after the phit becomes the head of the queue.

The slave side *shell*, represented as section 5, strips the message headers

from the transaction. This is represented as a latency of 1 cycle per word.

The *multibus*, represented as section 6, arbitrates the transactions arriving from multiple *shells*. The *multibus* uses Round Robin (RR) run-time arbitration that can also be LR abstracted. At the Slave IP data is either *loaded* or *stored*. The temporal behaviour of the *multibus* and the Slave IP are modelled together. This explained in more detail in Section 2.7.

Data, from a *load* returns along the response channel. The data can pass directly through the slave side *shell* without delay. The producing, and consuming, *NIs* along with the *path*, of the response channel, operate as before. The *path* taken by the response channel across the router network may not be the same as the request channel. The master side *bus* and *shell* transfer the data without delay.

The Æthereal NoC uses a credit system for flow control. The producing *NI* contains a credit counter that is set to the buffer size at the consuming *NI*. Whenever the producing *NI* releases a phit it decrements its counter. Whenever the counter is zero the producing *NI* will not release phits until a credit has returned from the consuming *NI*. Figure 2.1 illustrates the connection model, including credit return represented as dashed arrows, originating at the consumer *NIs*. Although not illustrated, credits from the consuming *NI* return via the opposite channel to the producing *NI*.

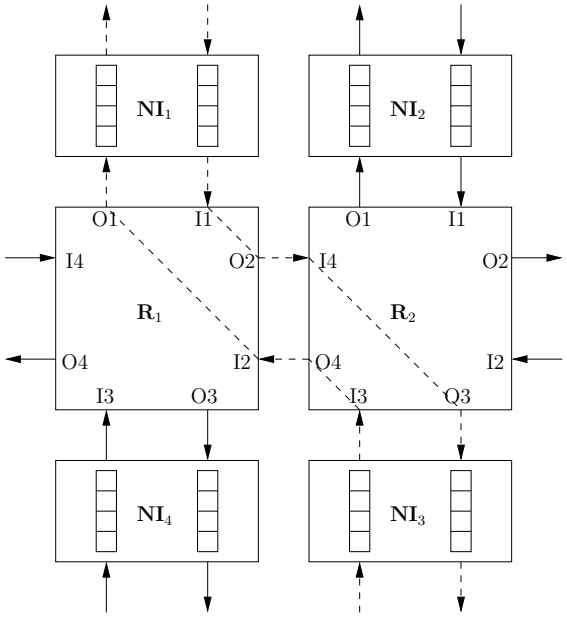
In this Section a model that can be used to conservatively simulate individual Æthereal connections independently, is contributed. The model uses a combination of cycle accurate and Latency Rate abstraction to achieve this. In Section 2.2 it is shown how Æthereal achieves contention free routing and provide bounds on latency and throughput for individual NoC connections.

## 2.2 Æthereal NoC

The Æthereal NoC, as described in [5], can provide the Guaranteed Services (GS) that are required by a predictable platform. As regards the GS, Æthereal is a contention free, wormhole routed, packet switching NoC that provides end-to-end guarantees, from NI to NI, on throughput and bounded latency. This is achieved using contention-free routing through pipelined time-division-multiplexed circuit switching.

Contention-free routing, in the form of pipelined time-division-multiplexed circuit switching, can be used to prevent the contention of multiple packets for shared resources. Virtual circuit switching is used over the packet switching network to create connections with fixed latency and throughput. Time-division-multiplexing permits multiple circuit switched connections to share the same resource, just not at the same time, while still maintaining a guaranteed latency and throughput.

Figure 2.2 illustrates an example connection over an Æthereal NoC. The example illustrates an implementation of the Æthereal NoC using routing tables on the routers. The routing tables are synchronised at the same clock frequency and each table has the same period. Every cycle the slot pointer increments, modulo the table period, connecting the inputs to the outputs for that slot in the table. E.g. a single phit of data arrives and is buffered in $NI_1$ to be communicated to $NI_3$. When the phit is at the head of the buffer and $R_1$'s routing table is in slot 0 then the phit can proceed across $R_1$ to $R_2$. The




| slot | O1 | O2 |
|---|---|---|
| 0 | | I1 |
| 1 | | |
| 2 | | |
| 3 | I2 | |

(a) Routing table for $R_1$

| slot | O3 | O4 |
|---|---|---|
| 0 | | |
| 1 | I4 | |
| 2 | | I3 |
| 3 | | |

(b) Routing table for $R_2$

Figure 2.2: Example connection through contention free routing using pipelined time-division-multiplexed circuit switching.

following cycle the tables are in slot 1 permitting the phit to traverse $R_2$ and arrive at $NI_3$.

Traffic can also be routed across the Æthereal NoC using source based routing strategies where the TDMA routing tables are located in the NIs. The creation of source routed TDMA tables, for the Æthereal NoC, is beyond the scope of this thesis. More information on the creation of source routed TDMA arbitration tables may be found in [6]. This is the Æthereal configuration that is modelled as it is the configuration that is used in the FPGA implementation.

Taking the TDMA arbitration table `0X0XX` as an example. The 0s and Xs represent three phit sized service slots, with 0 representing no services and X representing phits where services take place. In some of these phit sized slots services may be performed, as illustrated in Figure 2.3.

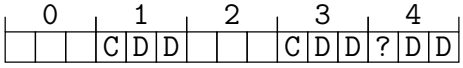



Figure 2.3: Slot Table `0X0XX`

The service slot types from Figure 2.3 are defined as:

`C` Header in which credits may be returned. Occurs at the start of each group of slots and at the start of every $8^{th}$ slot in the group.

`D` Slot in which data may be communicated.

`?` If a transaction is to be serviced by this slot after an idle period then the slot will act as a `C` slot. Otherwise it is normally a `D` slot.

The table's behaviour may be represented graphically as illustrated in Figure 2.4. After the `D` service slot has completed the amount of data words processed is incremented. The `?` service slot is not counted as a `D` service slot in Figure 2.4. This is because the figure illustrates the guaranteed data service graph.

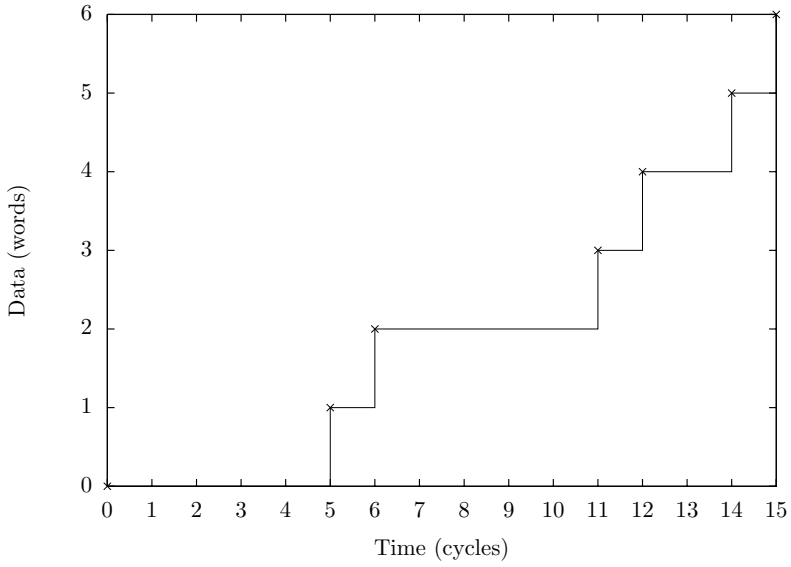



Figure 2.4: Service graph for slot table from Figure 2.3 with distributed service slots.

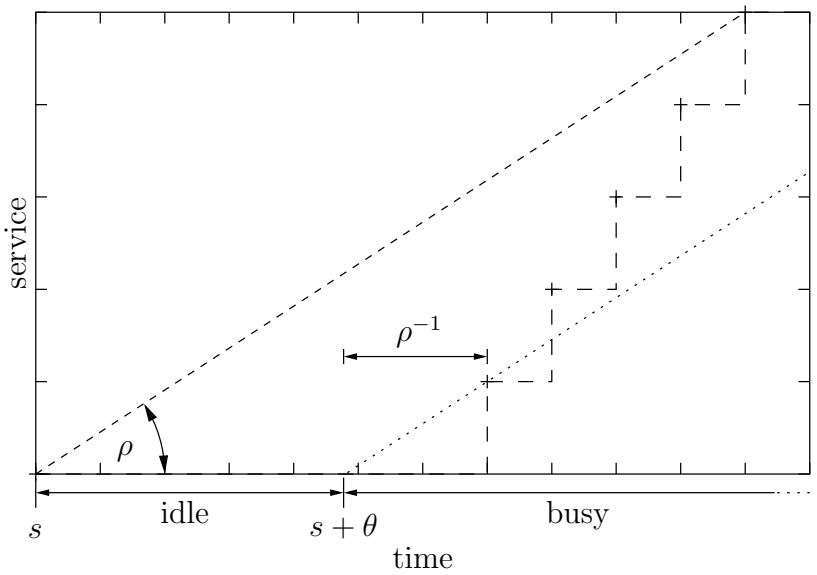
Throughout the rest of this thesis arbitration tables that contain a single block of Continuous Service Slots (CSS) are classified as being CSS tables. Arbitration tables with Distributed Service Slots (DSS), as illustrated in Figure 2.3, are classified as being DSS tables.

In this section it is explained how the routing strategy of the Æthereal NoC facilitates the ability to model connections independently. The structure of a source routed TDMA table for use on the Æthereal NoC is described along with how its behaviour may be represented graphically. In Section 2.4 it is described how the producing NIs, that perform the TDMA arbitration, may be abstractly modelled.

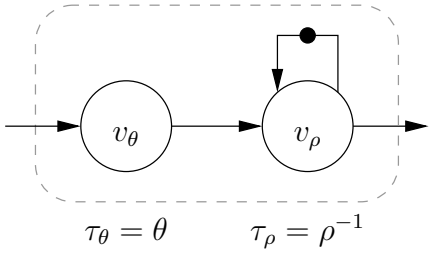
## 2.3 Latency Rate Abstraction

LR servers [4] facilitate the conservative modelling of run-time arbitration components, such as the TDMA involved in the contention-free routing of the Æthereal NoC as described in Section 2.2. LR servers were originally developed to aid in the production of quality of service guarantees for networks composed of heterogeneous routers by generalising the different scheduling algorithms. The

generalisation is realised by encapsulating the temporal characteristics of the scheduling algorithms using two variables, *latency* and *rate*. Denoting *latency* as $\theta$ and *rate* as $\rho$, the relationship of these two variables to the actual behaviour is illustrated in Figure 2.5a.




(a) Relationship of the temporal behaviour of a Latency Rate server to the actual scheduling behaviour.




(b) Latency Rate server dataflow representation.

Figure 2.5: Latency Rate abstraction.

The LR server is represented by the dataflow model illustrated in Figure 2.5b. The latency vertex $v_\theta$ is not self timed permitting the latency delay to be pipelined for multiple tasks. The rate vertex is self timed only allowing one task to be serviced at a time. The LR server is said to be *busy* whenever the rate vertex is busy.

In Figure 2.5a the lines starting at $s$ and $s+\theta$ are both parallel and therefore have the same gradient of $\rho$. For the line starting at $s$, $\rho$ is the minimal rate of service required for the LR server to maintain a constant *busy* state. For the line starting at $s+\theta$, $\rho$ is the *rate* of execution that can be maintained by the LR server while still remaining conservative. The *latency* $\theta$ is the duration until the *rate* of execution $\rho$ can be conservatively maintained.

Defining a task as an atomic entity requiring service the principle behind the

temporal behaviour of the LR server, during a *busy* period, is that the start, and therefore finish, of the execution of a task is dependent upon the finishing time of the preceding execution. An execution of the task cannot begin until the preceding execution has finished. Letting $U$ be the set of tasks, by conservatively bounding the finishing time of the preceding execution $f(u_x, i-1)$ the finishing time of the succeeding execution $f(u_x, i)$ is conservatively bounded as $f(u_x, i) = f(u_x, i-1) + \rho_x^{-1}$. The LR server can subsequently be represented as illustrated in Figure 2.5b. $\tau_x$ represents the delay caused by the actor $v_x$. $v_\rho$ has a self timed edge that makes sure that only one task is serviced by the rate at a time.

The first execution of a busy period is conservatively bounded independent of any preceding executions from other busy periods. By taking the start time of the execution $s(u_x, i)$ and adding the *latency* $\theta_x$ the finishing time of the execution $f(u_x, i)$ is conservatively bounded by Equation 2.1.

$$f(u_x, i) = s(u_x, i) + \theta_x + \rho_x^{-1} \tag{2.1}$$

Combining the two possibilities of the task requiring service during a *busy* period or during an *idle* period, the finishing time of an execution of a task is given by Equation 2.2.

$$\forall i \in \mathbb{N}^*, \forall u_x \in U : f(u_x, i) = \max(s(u_x, i) + \theta_x, f(u_x, i-1)) + \rho_x^{-1} \tag{2.2}$$

As there are no preceding tasks for the first execution $i = 0$ Equation 2.1 holds in this case.

In this Section the principles of LR abstraction have been described. It is demonstrated graphically in Figure 2.5a how a Latency offset can be used to bound the sustained Rate of a TDMA table. The mathematical principles of the LR servers implementation for use in simulation are encapsulated in the Equations 2.1 and 2.2. In Section 2.5 an analytical method of LR value derivation is described.

## 2.4 Modelling the Producing NIs

In [8] the *request/response* channel, from producing NI to consuming NI, is modelled as a dataflow graph. The main principle, behind the modelling of the communication channels, is that the scheduling of data, in the producing NI, is dependent on the availability of credits, that represent space in the consuming NI buffer, as described in Section 2.1. Four models were proposed in [8], of varying levels of abstraction. Figure 2.6 illustrates the four proposed channel models from [8] that have been modified to only model the producing NI instead of the representing the NoC connection request or response channel. The modifications are made because the NoC connection is now modelled as shown in Figure 2.1. The four models have varying degrees of abstraction

**L** Most abstract model combining latency, rate, credit return and data arbitration into a single latency $\tau_{cd,\theta\rho}$. Illustrated in Figure 2.6a.

**LR** Less abstract than the **L** model by modelling latency and rate separately as delays $\tau_{cd,\theta}$ and $\tau_{cd,\rho}$ respectively. Illustrated in Figure 2.6b.

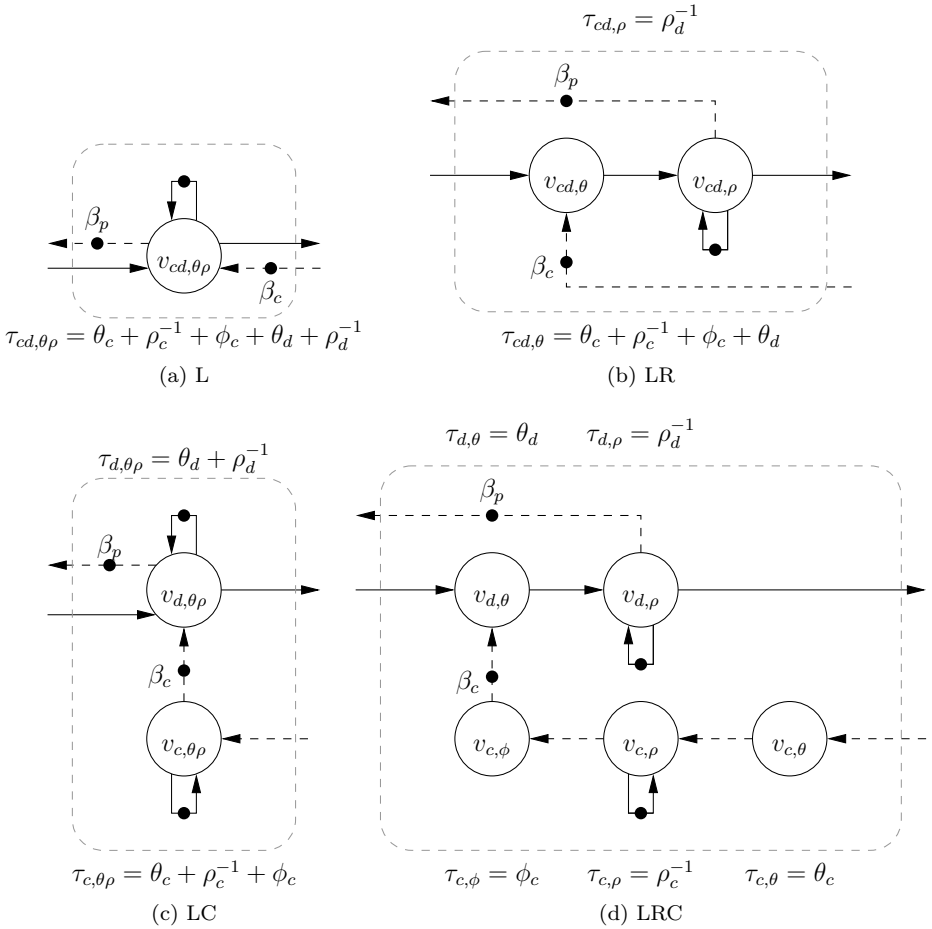



Figure 2.6: Producing NI Models.

**LC** Less abstract than the **L** model by modelling the credit return and data arbitration separately as delays $\tau_{c,\theta\rho}$ and $\tau_{d,\theta\rho}$ respectively. Illustrated in Figure 2.6c.

**LRC** Least abstract model that combines the separation of concerns from the **LR** and **LC** models. Credit return latency, rate and NoC path are represented as delays $\tau_{c,\theta}$, $\tau_{c,\rho}$ and $\tau_{c,\phi}$ respectively. Data latency and rate are represented as delays $\tau_{d,\theta}$ and $\tau_{d,\rho}$ respectively. Illustrated in Figure 2.6d.

In Figure 2.6 $\tau_x$ represents the delay produced by actor $v_x$ on an arriving task. In Figure 2.6a, in order to model the behaviour of the producing NI conservatively, it is assumed that there are never consumer NI buffer credits $\beta_c$ whenever data needs serviced. $\tau_{cd,\theta\rho}$ in this case is calculated as the worst case time before a credit return is scheduled. $\theta_x$ and $\rho_x$ being representative of Latency and Rate values, $\theta_c + \rho_c^{-1}$ represent the delay due to the runtime TDMA scheduling for the return of credits across the NoC. $\phi_c$ represents the time taken for 1 phit to cross the NoC. $\theta_d + \rho_d^{-1}$ represents the delay due to

the runtime TDMA scheduling of data for transmission across the NoC. The modelling of runtime TDMA arbitration is addressed in detail in Sections 2.5 and 2.6. The actor $v_{cd,\theta\tau}$ has a self timed edge meaning that only one task can be serviced by the actor at a time. $\tau$ in Figure 2.6a is a constant latency, hence the model is known as **L**.

It is possible to refine the producing NI model, in Figure 2.6a, further by splitting the model along the Latency/Rate domain and the Data/Credits domain. Refining the abstraction of the model, to a lower level, through splitting actors permits the latencies attributable to those actors to be pipelined. This permits the production of tighter conservative bounds for the latency across the producing NI.

In Figure 2.6b model **L** is split in the Latency/Rate domain for data transfer, creating model **LR**. In this model the Latency and Rate values, for the TDMA runtime arbitration of data across the NoC, are split into two separate actors to form a Latency Rate server, as in Figure 2.5b but with buffer credits $\beta$ being modelled. Credit return is still assumed to be worst case, i.e. there are no credits at the producing NI and there are currently none scheduled.

In Figure 2.6c model **L** is split in the Data/Credits domain, creating model **LC**. This model is split acknowledging that while Data transmission is dependent on the availability of credits, the scheduling of both are independent.

The most refined model **LRC**, illustrated in Figure 2.6d, splits the model **L** in the Latency/Rate domain and the Data/Credits domain. In this model the actors represent atomic delay values permitting the maximum amount of pipelining. This makes the **LRC** model the most tightly conservative model of the four.

In this Section four producing NI models of various degrees of abstraction are described. In Chapter 5 results of the simulation of the case study applications are presented from NoC models using the four different producing NI models, illustrated in Figure 2.6. In Section 2.3 the principles behind Latency Rate abstraction are explained, with Sections 2.5 and 2.6 demonstrating two methods how LR values, $\theta_x$ and $rho_x$, may be calculated from source routed TDMA tables.

## 2.5 Analytical CSS LR Value Derivation for TDMA Arbitration

In [14] it is shown that Time Division Multiple Access (TDMA) is a form of LR server. Æthereal uses slot tables to achieve TDMA routing. In this section it is shown how to generalise the temporal behaviour of Æthereal routing using the method of [14].

As in [14] let $U$ be the set of tasks, $f\left(u_x, i\right), f : U \times \mathbb{N} \rightarrow \mathbb{R}$ be the finish time of execution $i$ of task $u_x$, $P$ be the slot table period, $S_x$ be the time slice allocated to task $u_x$, $D_{x,i}$ be the execution time of execution $i$ of task $u_x$ once the task has been scheduled, and $\bar{D}_x$ be the worst-case execution time of task $u_x$ once the task has been scheduled. The NI arbitration services data on phit granularity. Upon scheduling a phit is serviced in one cycle, i.e. $\bar{D}_x = D_x = 1$.

The guaranteed rate $\rho_x$ at which task $u_x$ finishes in a busy period can be obtained by observing that in a busy period the start time $i$ of task $u_x$ will

follow the finishing of execution $i-1$ of the same task as soon as possible. In the case of execution $i$ of task $u_x$ starting immediately after execution $i-1$ the finishing time of execution $i$ is given by $f(u_x, i) = f(u_x, i-1) + D_{x,i}$. Using TDMA scheduling, tasks can only be serviced for a duration $S_x$ in every period of length $P$. Taking this into account the finishing time of execution $i$ of task $u_x$, scheduled using TDMA, is given by $f(u_x, i) = f(u_x, i-1) + D_{x,i}\frac{P}{S_x}$. It follows that during a busy period, the period of repetition of task $u_x$ can be conservatively bounded by $\bar{D}_x\frac{P}{S_x}$ since it must hold that $\forall i \in \mathbb{N} : f(u_x, i) - f(u_x, i-1) \leq \bar{D}_x\frac{P}{S_x}$. Taking into account that $\bar{D}_x = 1$, the guaranteed rate of execution $\rho_x$ of task $u_x$ is given by Equation 2.3.

$$\rho_x = \frac{S_x}{P} \tag{2.3}$$

The latency $\theta_x$ of task $u_x$ can be obtained by observing that the worst case response time $\bar{r}_x$ of the task occurs after the first execution in a busy period given by $\bar{r}_x = \bar{D}_x + (P - S_x)\lceil\frac{\bar{D}_x}{S_x}\rceil$. Taking into account that $\bar{D}_x = 1$ the worst case response time of task $u_x$ is given by Equation 2.4.

$$\bar{r}_x = 1 + (P - S_x) \tag{2.4}$$

Equation 2.4 ignores the arrangement of the service slots in the table. By assuming the worst case arrangement, i.e that the table is arranged with one block of Continuous Service Slots (CSS), Equation 2.4 calculates a conservative worst case response time that may not be accurate. Figure 2.7 illustrates a Distributed Service Slot (DSS) table while Figure 2.8 shows the assumed CSS table configuration.

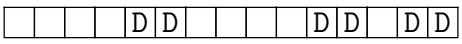



Figure 2.7: Example DSS TDMA table

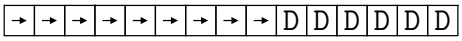



Figure 2.8: In Equation 2.4 $\bar{r}$ is calculated by assuming that the table consists of Continuous Service Slots (CSS). This creates the largest period of non service for the table.

The worst case response time of the first execution in a busy period of a LR server is given by $\bar{r}_x = \theta_x + \rho_x^{-1}$. Substituting this into Equation 2.4 produces Equation 2.5.

$$\theta_x + \rho_x^{-1} = 1 + (P - S_x) \tag{2.5}$$

Taking the inverse of $\rho_x$ in Equation 2.3 and combining this with Equation 2.5 produces Equation 2.6

$$\theta_x + \frac{P}{S_x} = 1 + (P - S_x) \tag{2.6}$$

which can be rewritten to form Equation 2.7.

$$\theta_x = 1 + (P - S_x) - \frac{P}{S_x} \tag{2.7}$$

In this Section it is shown how the analytical method described in [14] for deriving LR values for TDMA tables applies to the properties of the Æthereal NoC and the Æthereal NoC source routed TDMA tables. The equations in this Section form an integral part of the algorithmic LR value derivation, described in Section 2.6.

## 2.6 Algorithmic DSS LR Value Derivation for TDMA Arbitration

It is described in Section 2.5 how to calculate the values of *latency* $\theta$ and *rate* $\rho$ to represent an Æthereal NoC as an LR server. In this Section it is demonstrated that the analytical method from Section 2.5 calculates an unnecessarily overly conservative Latency component of the LR values for DSS arbitration tables. An algorithmic method is contributed that can produce tightly conservative LR values for CSS and DSS arbitration tables.

The analytical method described in Section 2.5 assumes that service is provided in a continuous sequence of service slots. This is not always the case. The slot table may be configured with distributed service slots. Assuming a continuous sequence of service slots is always conservative for a distributed slot table of the same period and number of service slots. This is easily rationalised as the sustainable rate of both tables are the same due to the equalities in the period and the number of service slots. For a distributed table, the latency before this rate is conservatively sustainable may be less than or equal to the latency for a table with continuous service slots. The table with continuous service slots also has a period of continuous idle slots. As all the idle slots are clustered together this is the longest possible period of non-service for the table. If the service slots are distributed the groups of idle slots can only become shorter.

An observation that can be made is that the worst case response time for a distributed table is not necessarily caused by a single transaction arriving at a specific instance but by multiple transactions arriving at a specific instance and being serviced sequentially. For the purposes of calculating the worst case response time the distributed table can be visualised as being made up of multiple sub-tables that only contain continuous service slot distribution, at the end of their length. The worst case response time for a continuously distributed table always occurs for a single transaction arriving at the start of the table's idle period. Any subsequent transactions to be serviced in the same sub-table lower the average response time as they are serviced immediately. The combined effect of a series of transactions can cause the worst case response time for the table if the transactions must span multiple sub-tables.

To demonstrate the effect of this observation conservative LR values are derived for the TDMA table illustrated in Figure 2.3. The table has a period $P = 15$ cycles and a task service time of $S_{D\_busy} = 7$ cycles or $S_{D\_idle} = 6$ cycles. The `?` slot will act as a `D` during busy periods so can be included in Equation 2.3 to reflect its effect on the sustained rate $\rho_D$.

$$\rho_x = \frac{S_{x\_busy}}{P} \tag{2.8}$$

The latency $\theta_D$ required before the rate $\rho_D$ is sustainable is derived from the worst case response time $\bar{r}_D$. The worst case response time $\bar{r}_D$ does not occur during a busy period and as such the `?` slot can not be guaranteed to act as a `D` slot. This can be taken into account in Equation 2.7 along with the changes to the sustained rate $\rho_D$.

$$\theta_x = 1 + (P - S_{x\_idle}) - \frac{P}{S_{x\_busy}} \tag{2.9}$$

Calculating the integer rate $\rho_D$, for slot table `0X0XX`, using the Continuous Service Slots (CSS) method:

$$\rho_D^{-1} = \frac{15}{7} \leq \left\lceil \frac{15}{7} \right\rceil = 3 \text{ cycles per word}$$

And the integer latency $\theta_D$:

$$\begin{aligned} \theta_D &= 1 + (15 - 6) - \rho_D^{-1} \\ &\geq 1 + (15 - 6) - 3 = 7 \text{ cycles} \end{aligned}$$

Since the network simulator can only handle integer delays, the latency and rate have to be scaled appropriately to maintain conservativeness. The sustained inverse rate $\rho_D^{-1}$ has to be rounded up to maintain conservativeness. The integer latency $\theta_D$ can then be derived from the rounded rate effectively avoiding rounding the latency up while still maintaining conservativeness.

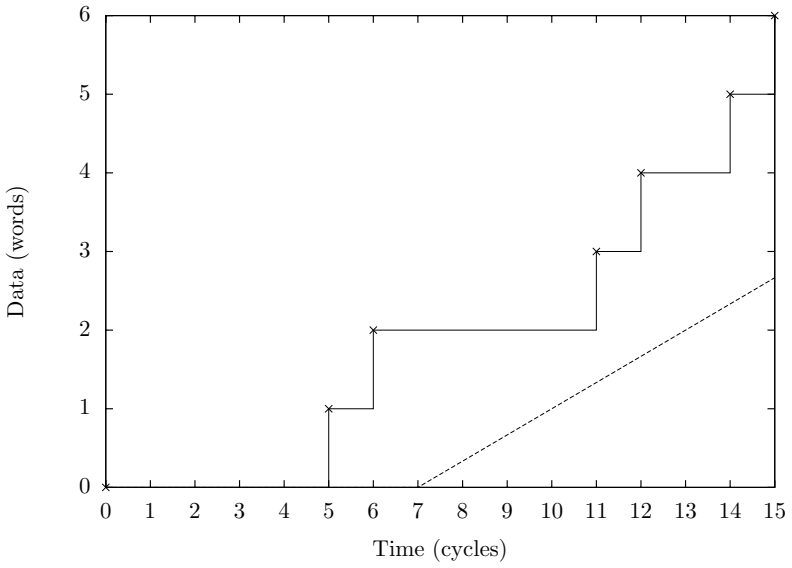



Figure 2.9: All points are conservatively bounded although the latency value is unnecessarily large.

While the CSS method is conservative for slot tables with non-continuous groups of slots, as can be seen in Figure 2.9 it can be unnecessarily overly conservative. The points marked as x's in Figure 2.9 must be bound in order to maintain conservativeness. As can be seen the points are bound by an unnecessarily large margin. The Distributed Service Slot (DSS) method takes the distribution of the service slots into account. In essence the CSS method assumes the table structure is worst case with all the service slots in a single group. Changing the distribution of the table slots does not effect the sustained rate of execution of a task $\rho_x$ but does effect the worst case response time of the task $\bar{r}_x$. The structure of the CSS table creates the greatest $\bar{r}_x$ as it inherently has the longest period of non-service for any configuration of the table. The latency $\theta_x$ calculated from the rate $\rho_x$ and $\bar{r}_x$ will therefore be conservative for any configuration of the table, but may be more conservative than necessary.

In the DSS method the distribution of the slots in the slot table is taken into account when calculating the worst case response time $\bar{r}_x$. Whereas for the CSS simplified table $\bar{r}_x$ is simply the number of non-service slots in the table for the particular task plus one for the task execution, $\bar{r}_x$ is not so straight forward to calculate for a DSS table. The worst case execution time in a DSS table is not necessarily caused by a single task execution as is shown in Section 4.1.

The DSS method to calculate latency and rate works by breaking the DSS table down into its component CSS sub-tables. The worst case response for a CSS table is easily calculated and is valid for the individual CSS sub-tables.

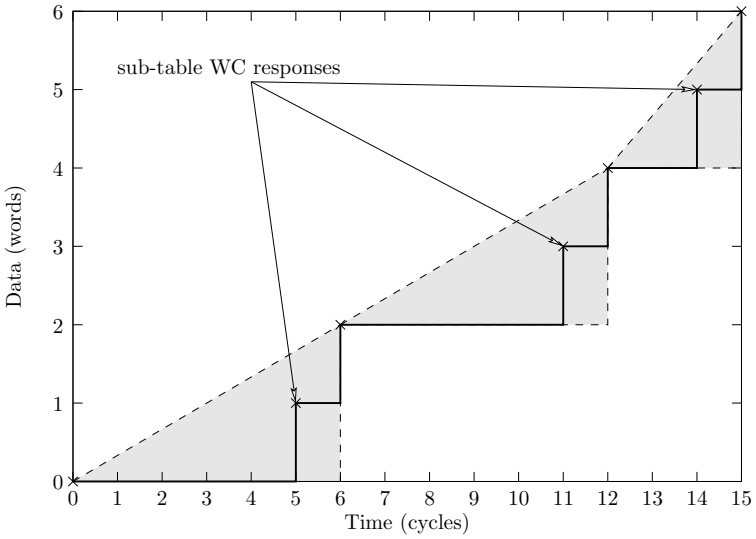



Figure 2.10: Splitting the table into multiple sub-tables, as illustrated by the dashed triangles.

Since $\bar{r}_x$ may be caused by more than one task execution, after a period of idleness, any possible sequential combination of sub-tables may be responsible. In Figure 2.10 sub-tables are encapsulated by grey dashed right angle triangles. Their local worst case response times are also indicated. The gradient of the hypotenuse of the triangles is the rate of execution of the task within the time frame of the triangle.

It is obvious that the worst case response time $\bar{r}_x$ for the entire table is found at one or more of the local worst case responses for the CSS sub-tables. This is the case because any executions of a task within a continuous group will always have a lower response time than the preceding execution. Similarly the worst case starting points are found at one or more of the CSS sub-table starting points. These are the points before groups of non-service. Entrance at any succeeding slot, in the group of non-service, would only lower the response time.

Finding $\bar{r}_x$ is a matter of checking the response times of all the starting points to all the ending points. To do this the latency of the CSS sub-table $\theta_{x(n\rightarrow m)}$ required to conservatively bound the local worst case responses (dotted triangles in Figure 2.10) must be calculated. A latency offset $\delta_{x(n\rightarrow m)}$ is assigned to each sub-table (dashed triangles in Figure 2.10). This is necessary to maintain conservativeness and tightness of the sustained Rate bounding of the TDMA behaviour. If a sub-table has a local rate faster than the DSS table rate then a negative offset is required to maintain tightness. If a sub-table has a local rate slower than the DSS table rate then a positive offset is required to maintain conservativeness.

To accomplish this a new term $\delta_{x(n\rightarrow m)}$ is defined, that represents the latency offset requirement for the CSS sub-table. Equation 2.10 describes how the offset is calculated. Multiplying the inverse rate $\rho_x^{-1}$ for the entire table by the number of service slots $S_{x(n\rightarrow m)}$ in the CSS sub-table gives the length of time required to transmit the same amount of words at the entire table's rate. Subtracting this from the CSS sub-table period $P_{(n\rightarrow m)}$ returns the latency offset that this table may contribute when calculating the latency for the entire table.

$$\delta_{x(n\rightarrow m)} = P_{(n\rightarrow m)} - \rho_x^{-1} S_{x(n\rightarrow m)} \tag{2.10}$$

For each CSS sub-table from $n \rightarrow m$ the latency $\theta_D(n \rightarrow m)$ and latency offset $\delta_D(n \rightarrow m)$ is calculated.

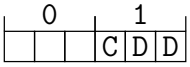



Figure 2.11: CSS sub-table 0–1

For the CSS sub-table illustrated in Figure 2.11, the table's period and number of data service slots are used to calculate the worst case response time in the same manner as the CSS method.

$$\begin{aligned} P_{(0\rightarrow 1)} &= 6 \\ S_{D(0\rightarrow 1)} &= 2 \\ \bar{r}_{D(0\rightarrow 1)} &= 1 + P_{(0\rightarrow 1)} - S_{D(0\rightarrow 1)} \\ &= 1 + 6 - 2 = 5 \text{ cycles} \end{aligned}$$

The worst case response time is subsequently used to calculate the latency in the same manner as the CSS method. The DSS method also requires the latency offset to be calculated for the sub-table to account for the differing average rates between the DSS table and the CSS sub-table.

$$\begin{aligned}\theta_{D(0\to1)} &= \bar{r}_{D(0\to1)} - \rho_D^{-1} \\ &= 5 - 3 = 2 \text{ cycles} \\ \delta_{D(0\to1)} &= P_{(0\to1)} - \rho_D^{-1} \times S_{D(0\to1)} \\ &= 6 - 3 \times 2 = 0 \text{ cycles}\end{aligned}$$

This is repeated for the rest of the CSS sub-tables to obtain the latency and latency offset values.

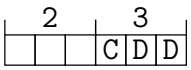



Figure 2.12: CSS sub-table 2–3

$$\begin{aligned}P_{(2\to3)} &= 6 \\ S_{D(2\to3)} &= 2 \\ \bar{r}_{D(2\to3)} &= 1 + 6 - 2 = 5 \text{ cycles}\end{aligned}$$

$$\begin{aligned}\theta_{D(2\to3)} &= 5 - 3 = 2 \text{ cycles} \\ \delta_{D(2\to3)} &= 6 - 3 \times 2 = 0 \text{ cycles}\end{aligned}$$


Figure 2.13: CSS sub-table 4

$$\begin{aligned}P_{(4)} &= 3 \\ S_{D(4)} &= 2 \\ \bar{r}_{D(4)} &= 1 + 3 - 2 = 2 \text{ cycles}\end{aligned}$$

$$\begin{aligned}\theta_{D(4)} &= 2 - 3 = -1 \text{ cycles} \\ \delta_{D(4)} &= 3 - 3 \times 2 = -3 \text{ cycles}\end{aligned}$$

In order to bound the worst case data arbitration the combined effects of the CSS sub-tables must be taken into account. The CSS sub-tables are used in a sequential manner reducing the combinations that must be examined. Three starting points to three ending points generates a set of nine possible latencies. The greatest latency value (or values) from this set is the conservative latency value. Table 2.1a displays how the latencies are calculated using the latencies $\theta_{x(n\to m)}$ and the latency offsets $\delta_{x(n\to m)}$. The results of these calculations are displayed in Table 2.1b

| | CSS $(0 \rightarrow 1)$ | CSS $(2 \rightarrow 3)$ | CSS (4) |
|---|---|---|---|
| CSS | $\theta_{D(0\rightarrow1)}$ | $\theta_{D(2\rightarrow3)}$ | $\theta_{D(4)}$ |
| CSS +1 | $\delta_{D(0\rightarrow1)} + \theta_{D(2\rightarrow3)}$ | $\delta_{D(2\rightarrow3)} + \theta_{D(4)}$ | $\delta_{D(4)} + \theta_{D(0\rightarrow1)}$ |
| CSS +2 | $\delta_{D(0\rightarrow1)} + \delta_{D(2\rightarrow3)} + \theta_{D(4)}$ | $\delta_{D(2\rightarrow3)} + \delta_{D(4)} + \theta_{D(0\rightarrow1)}$ | $\delta_{D(4)} + \delta_{D(0\rightarrow1)} + \theta_{D(2\rightarrow3)}$ |

(a) Table of latency calculations for all possible sequential combinations of CSS sub-tables.

| | CSS $(0 \rightarrow 1)$ | CSS $(2 \rightarrow 3)$ | CSS (4) |
|---|---|---|---|
| CSS | 2 | 2 | $-1$ |
| CSS +1 | 2 | $-1$ | $-1$ |
| CSS +2 | $-1$ | $-1$ | $-1$ |

(b) Table of latencies resulting from the calculations in Table 4.1a. The maximum value is the conservative latency for the table.

Table 2.1: Algorithmic latency value derivation for a DSS TDMA table. CSS sub-tables are represented horizontally and sequential combinations of CSS sub-tables are represented vertically, e.g. CSS +1 indicates the CSS sub-table in the horizontal domain in combination with the following CSS sub-table.

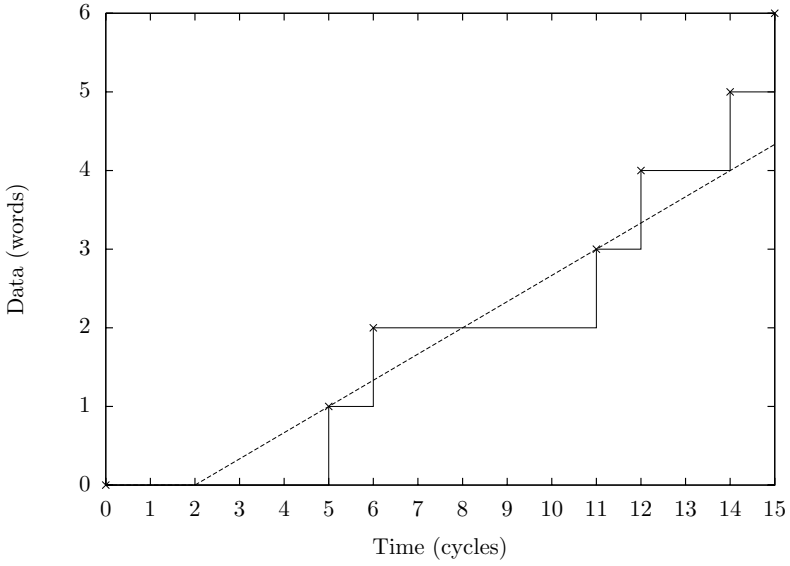



Figure 2.14: The sustained rate is still provided conservatively while the latency is reduced compared to the CSS method.

The maximum latency in Table 2.1b is the conservative latency. There are three start to end point combinations that require the latency to be 2 cycles. Working back from the values' positions in Table 2.1b it is possible to work out the combination of events that create the worst case responses. A latency of 2 cycles is required to conservatively provide an inverse rate of 3 cycles whenever:

1. 1 word of data enters at the start of slot 0
2. 1 word of data enters at the start of slot 2

3. 3 words of data start to enter at the start of slot 0

For this table the tightest Latency $\theta_D$, while still remaining conservative for a sustained inverse rate $\rho_D^{-1} = 3$ cycles per word, is $\theta_D = 2$ cycles. It can be seen in Figure 2.14 that the DSS derived latency is conservative while not being overly conservative as in the CSS method in Figure 2.9.

In this Section it is shown that the LR values calculated using the analytical CSS method are unnecessarily overly conservative for DSS TDMA tables. An algorithmic method is contributed that uses the principles of the analytical method to calculate tight conservative LR values for both CSS and DSS TDMA tables. The method is demonstrated by deriving conservative LR values for the example slot table illustrated in Figure 2.3. In Section 2.7 it is described how conservative LR values can be derived to encapsulate memory arbitration and access times at the Slave IP.

## 2.7 LR Abstraction of Slave Side Arbitration and Memory Access

In the implementation the Multibus component, in Figure 2.1, arbitrates the connections from multiple shells for access to the Slave IP. Arbitration is carried out on a Round-Robin (RR) basis. The Multibus and the Slave IP may run at different clock frequencies, so they must communicate through a clock bridge, not illustrated in Figure 2.1. The Multibus, clock bridge and Slave IP can be visualised as being in the configuration illustrated in Figure 2.15.

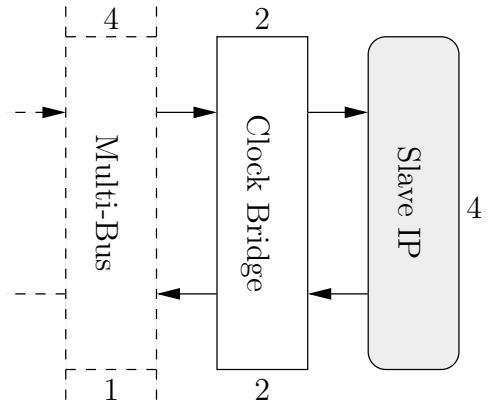



Figure 2.15: Slave side

The numbers in Figure 2.15 represent the number of cycles of delay caused by the particular component. The RR scheduling in the Multibus causes a minimal delay of 4 cycles, i.e. RR arbitration of a single incoming connection takes 4 cycles. The clock bridge in the request direction causes a 2 cycle delay. The Slave IP takes 4 cycles to read or write a word. If data has to be returned by a read then it must pass through the clock bridge again on the response channel causing a further 2 cycle delay and the Multi-Bus incurring a 1 cycle delay. The connection model, as described in Section 2.1, models individual connections independently.

In order to conservatively bound the temporal behaviour of the Multibus and Slave IP access, each connection must assume the worst case for the Round Robin arbitration, i.e. that all other incoming connections to the Multibus require servicing at the same time. The temporal behaviour of the Multibus, clock bridge and Slave IP may be conservatively modelled as single LR server. Assuming $n$ to be the number of possible incoming connections to the Multibus in reality, then the worst case response time $\bar{r}$ for reading a single word from the Slave IP can be calculated as in Equation 2.11.

$$\bar{r} = 4n + 2 + 4 + 2 + 1 = 4n + 9 \text{ cycles} \tag{2.11}$$

The sustainable rate $\rho$ is governed by the bottleneck, which in this case is the RR arbitration in the Multibus, therefore $\rho^{-1} = 4n$. As is explained in Section 2.5, latency $\theta$ plus the inverse rate $\rho^{-1}$ must at least equal the worst case response time $\bar{r}$ in order to remain conservative, i.e $\bar{r} = \theta + \rho^{-1}$. The latency $\theta$ can then be calculated using this equation by substituting the values calculated for worst case response time $\bar{r}$ and inverse rate $\rho^{-1}$ to calculate the latency $\theta$, as calculated in Equation 2.12.

$$\theta = \bar{r} - \rho^{-1} = 4n + 9 - 4n = 9 \text{ cycles} \tag{2.12}$$

In this Section it is shown how the memory arbitration and memory access times can be conservatively abstracted as LR values. In Chapter 3 it is described how the NoC model is implemented using the connection model from Section 2.1 and the producing NI models from Section 2.4. The NoC model is used in combination with the LR values that are derived using both the CSS and DSS methods in Chapter 4 to evaluate the effectiveness of the NoC model to perform conservative application-level performance analysis through simulation, by applying case study applications.

# Chapter 3

# Implementation of the NoC Model

The work from Chapter 2 explained how the NoC connection could be broken into atomic components, and a LR abstraction applied to run-time arbitrated components. In this Chapter it is explained how the NoC model is realised using the Silicon Hive SDK. The NoC model expands the already existing capability of the Silicon Hive SDK to simulate multiprocessor systems to also conservatively bound NoC transactions and memory arbitration timings.

The Chapter is structured as follows. In Section 3.1 an overview of the relevant parts of the Silicon Hive SDK is given. Using this as a starting point the section further goes on to explain some design options for integrating the NoC model into the tool flow. Section 3.2 looks at how the model is actually implemented using C and some of the algorithms that are used to achieve functional operation.

Section 3.3 explains how conservatively modelling individual connections on the transaction behaviour level for MPSoCs alone, does not guarantee conservativeness at the application level when tasks are distributed across multiple processors. A solution for the conservativeness of the communication is also explained and how this can be achieved using an inter-processor communication library. In Section 3.4 the rationality is explained behind how the implemented NoC model can be used in simulation to provide conservative application-level guarantees.

## 3.1 Silicon Hive Development Tools

The Silicon Hive SDK [12] provides an environment in which Multiprocessor Systems on Chip (MPSoC) can be developed and simulated. For this project the SDK provided the ability to:

- Create virtual custom devices, i.e. custom hardware components at the IP level.
- Create custom virtual hardware systems, in software, from proprietary or custom IP level components.

- Compile and simulate applications mapped to the custom hardware systems in software
- Compile and simulate applications on an FPGA.

### 3.1.1 Custom Devices and Systems

The Silicon Hive SDK provides the ability to describe a SoC using the Hardware System Description (HSD) language [13]. The HSD language is used to define how the individual IP level system components (or "devices" as seems to be the interchangeable naming convention in [13]) are connected and also assign values to devices configurable properties. The C programming language is used to describe the individual devices, with the assistance of the Hive System Simulator (HSS) API. The Silicon Hive SDK comes with some existing device descriptions for ubiquitous SoC devices such as memory and interconnect.

The Silicon Hive SDK provides the framework in which to describe custom components/devices for use in the simulation hardware platform. Key characteristics such as the device's functionality, communication, memory and temporal behaviour are able to be captured and replicated in simulation. Replication of a device's behaviour does not necessarily mean that every detail of the devices circuit diagram is mimicked but refers to a more abstract level of replication in which the devices behaviour as experienced by other components is mimicked. The devices temporal behaviour is imitated by a *wait* function to which the desired length of delay time can be passed.

**Communication** behaviour is captured by assigning one or more port devices to the custom device. There are multiple types of port device to help replicate streaming protocols or Memory Mapped IO (MMIO). MMIO ports can be configured as masters or slaves. The functionality of the ports is not inherent but must be specified by assigning call back functions. This provides the possibility to capture unique functional and temporal characteristics of ports.

**Memory** behaviour is captured as a static or dynamically reserved array. As devices are described in C this can be achieved as through the normal C methods for memory reservation. Capacity can be represented by the size of the reservation. Temporal delays for memory transactions can be taken into account by creating a delay for the transactions.

**Functionality** of the device is replicated in a callback function in which code can be placed that contains the operations to mimic the real device behaviour. Operations in the function coordinate the devices computation, communication and memory accesses.

The HSD language is used to declare the system's devices and their hierarchy. A typical system is composed of a Host Processor, a number of Hive Processors, distributed System Memory, possibly some custom devices, all connected to a System Bus. The HSD language uses the Port primitive to describe how the components interact. Checks are done at compile time to ensure that the port polarity is obeyed, e.g. two Slave ports are not connected to each other.

Once compiled the system can be linked in during application compilation allowing the application to run on the system during simulation.

### 3.1.2 Simulation

Simulation of the ported application is carried out in the Silicon Hive SDK using *hivesim*. Compiler arguments for the application dictate the degree of complexity of the simulation.

**sched** maps and schedules the processor resources. Temporal simulation does not include stall cycles due to NoC communication or memory arbitration.

**fpga** generates program code for use in an FPGA implementation of the system.

The *sched* simulation provides the most temporal accuracy out of all the software simulations. Due to the absence of temporal stall and arbitration accounting, this simulation can only provide limited information on the temporal behaviour of the system as a whole. Processor stalling could take up most of the system's runtime, in reality, without being indicated.

## 3.2 Implementing the Model

The NoC model integrates with the Silicon Hive platform model by replacing the SystemBus component in the system's HSD file. The NoC model has a variable number of ports so as to enable it to be easily reused for other system configurations. In order to model the NoC, object classes were created. The relationship between these classes can be seen in Figure 3.1.

The **Master** and **Slave** objects encapsulate the parameters and associated behaviour pertaining to the ports of the NoC. The `name` parameter is a *string* value that contains the name of the port. The `port` parameter is a *hss_mmio_server* or *hss_mmio_slave* object pointer depending on whether the port is a master or slave port. These are objects provided in the Silicon Hive HSS API that represent the communication ports of the custom component, which in this case is the NoC model. The `num_conns` variable records the number of connections from or to the port. The `connection[]` array of size `num_conns` stores **Connection** object pointers, of connections from or to the port. In addition to the other parameters the **Slave** object has `address_begin` and `address_end` values that represent the address range of the port.

The details described in the system's HSD file are used to initiate the NoC model component. The Silicon Hive simulator calls a function that is to be used to initialise the ports. It is in this function that the **Master** and **Slave** objects are initialised.

The **Connection** object encapsulates the parameters and associated behaviour pertaining to the enabled NoC connections. The `master` and `slave` parameters are object pointers to the **Master** and **Slave** objects of the ports that the connection runs between. The `model` parameter is a pointer to the **Model** object for the connection.

The **Connection** object contains the temporal values of the LR server elements in the `ld[]`, `ird[]`, `lc[]` and `irc[]` arrays. These represent the data latencies and inverse rates, and the credit latencies and inverse rates respectively. Two of each of these arrays relate to the request and response channel producing NIs. The *Multi-Bus* is also represented by a data latency and inverse rate, as described in Section 2.7. The `dp[]` array contains the length of the data path in both the request and response directions.

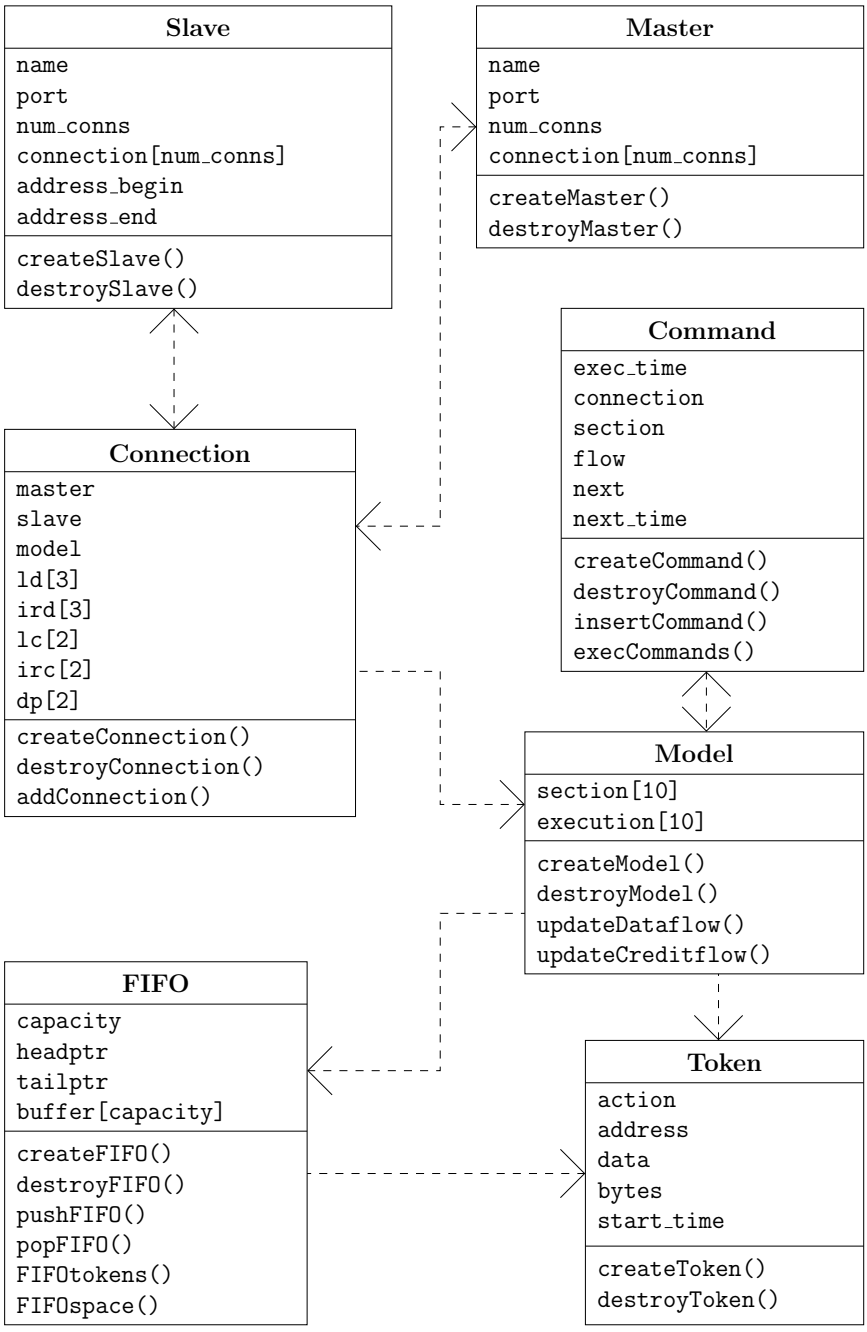



Figure 3.1: Connection model "use" dependency class diagram.

Once a connection has been created it can be added to the `connection` list of the **Master** and **Slave** port objects by calling the `add_connection()` function.

The **Model** object encapsulates the parameters and associated behaviour pertaining to the connection model as illustrated in Figure 2.1. The `section[]` array contains pointers to the **Token** and **FIFO** objects that make up the connection model. In Figure 2.1 sections 0, 1 and 5 are modelled as **Tokens** while sections 2, 3, 4, 6, 7, 8 and 9 are modelled as **FIFOs**. Only these components are modelled as the other components do not create a delay. The `execute[]` array is used to store the execution times of the sections after the wait of the **Token**, in that section, has been calculated.

The `updateDataflow()` and `updateCreditflow()` functions are used to perform and update on a particular section of the model to facilitate the *data* and *credits* flow through the model. This is explained in more detail in Section 3.2.1.

The **Command** object encapsulates the parameters and associated behaviour pertaining to commands that are scheduled to execute `updateDataflow()` or `updateCreditflow()` at some point in the future. The `exec_time` value stores the time in the future at which the command is to be executed. The `connection` is an object pointer to the **Connection** object on which the model is to be updated. The `section` value contains the section number on which the update is to be performed. The `flow` value indicates whether it is a *data* or *credit* flow update so that the appropriate update function is called when the `exec_time` is reached.

**Command** objects can be daisy-chained in two dimensions. This facilitates the formation of a two dimensional queueing structure that is described in more detail in Section 3.2.1. **Command** objects are chained using the `next` and `next_time` **Command** object pointer. **Command** objects can be entered into this queueing structure using the `insertCommand()` function. The `execCommands()` function is used to execute commands in the queue that have an `exec_time` that is the same as the current time.

The most basic building block of the entire connection model is the **Token** object. The **Token** object contains the details of the NoC transaction. The `action` value stores a character, which indicates if the transaction is for a read or a write. The `address` value stores the offset from the begin address of the port. The `data` value is a pointer to the location where the data is to be read from or written to. The `bytes` value is the number of bytes to be read or written. The `start_time` variable is used to store the time at which the **Token** entered a **FIFO**.

The **FIFO** object encapsulates the parameters and the associated behaviour pertaining to a **Token** FIFO. The `capacity` value stores the FIFO's token capacity. The `headptr` and `tailptr` values are pointers to the head and tail of the FIFO respectively. The `buffer[]` array of size `capacity` stores pointers to the **Token** pointers that have been pushed onto the FIFO.

The `pushFIFO()` function is used to push `Token` objects onto the FIFO. The `popFIFO` function is used to pop the head `Token` off the FIFO. The `FIFOtokens` function returns how many `Token` pointers are currently in the FIFO. The `FIFOspace` function returns how much capacity in the FIFO is currently unnoccupied.

### 3.2.1 Model Operation

The NoC provides Memory Mapped IO (MMIO) transactions. When a transaction is initiated by a master port a handler function in the Model is called. In the case of the NoC Model the interface between the ports and the connection model is illustrated in Figure 3.2.

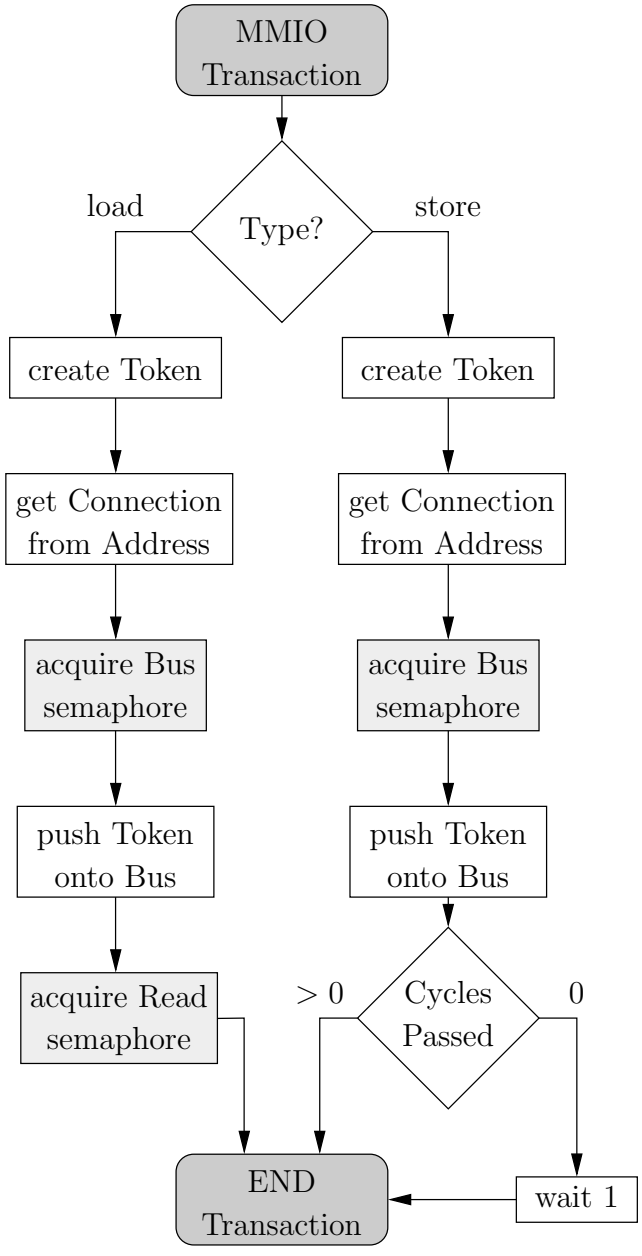



Figure 3.2: MMIO to Connection Model interface flowchart.

In Figure 3.2 a distinction is made between whether the transaction is a load

or a store. Both branches create a **Token** with the transaction information. In order to locate the correct **Slave** port the **Master** object's `connection[]` list is searched to find a **Slave** port with an appropriate address range. A *semaphore* is acquired that signals if the *Bus* section in the connection model is occupied. If the *Bus* is occupied, due to the previous transaction still being processed, the transaction is stalled until the *semaphore* is released. Upon the release of the *semaphore* the **Token** can be pushed into the first *section* of the connection model, representing the *Bus*, and the `updateDataflow()` function called.

If the transaction is a load (read) the master port needs to stall until the data is returned. This is achieved by using another *semaphore* that is released only whenever the read has completed the flow through the connection model. If the transaction is a store (write) the master port does not need to wait on an acknowledgement and may proceed. In order to make sure multiple transactions are not sent simultaneously a minimum total duration of the store is set to 1 cycle.

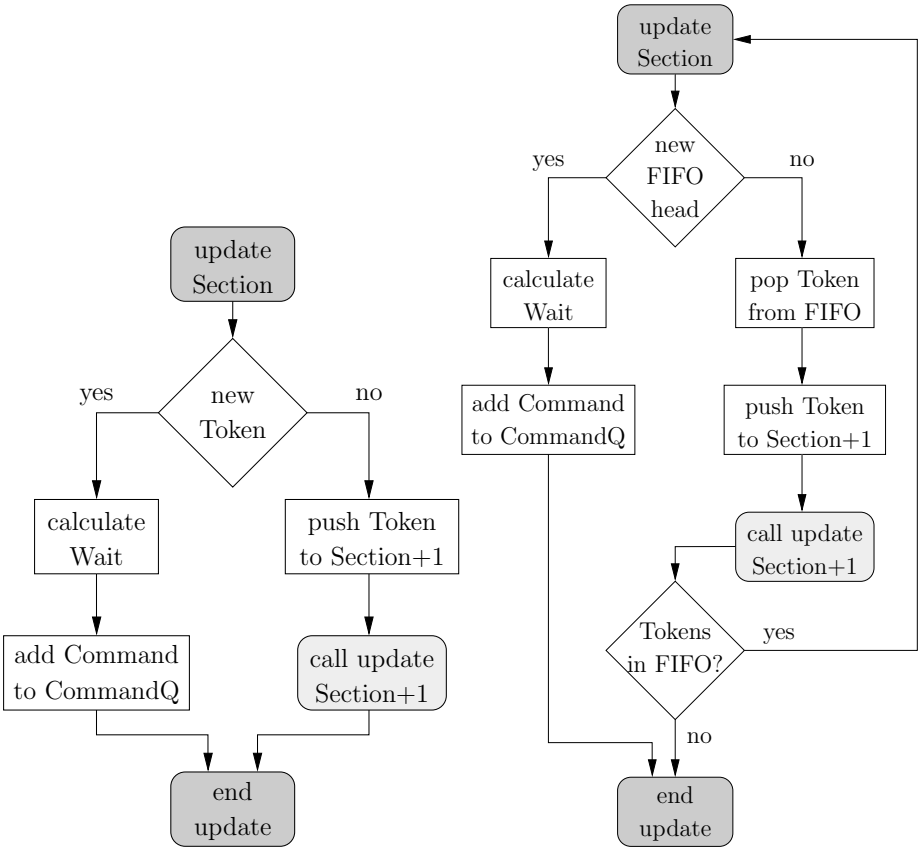
A **Token** flows through the connection model from master to slave, and in the case of a load, back to master again. Points of pipelined delays are represented as *sections* in the connection model. These may be positions that delay a single **Token** or **FIFOs** that delay multiple **Tokens**. Figure 3.3 illustrates the generic flowcharts of how the delays are created, in both instances, when `updateDataflow()` is called.

If a **Token** has just been pushed into a **Token** *section*, or has become the head of the **FIFO** buffer, then the *wait* for the token is calculated. A **Command** is created to update the *section* whenever the wait has expired. The **Command** is added to the `CommandQ` that stores the update callbacks for the entire NoC model. When the callback updates the section after the *wait* the **Token** is pushed to the next *section* and `updateDataflow()` for that *section* is called. In a **FIFO** *section* if there are **Tokens** still in the **FIFO** then `updateDataflow()` is recursively called for that *section*.

The `CommandQ` is structured as illustrated in Figure 3.4. Whenever the `insertCommand()` function is called the `CommandQ` is searched, in the temporal domain from earliest to latest **Command** `exec_time` to find the point where the **Command** should be inserted. If **Commands** already exist at that temporal instance then they must be *ordered* so that credit returns for that instance will be executed first and that updates to the model occurs from highest numbered section to the lowest.

**Commands** are ordered to ensure that *sections* do not have to stall unnecessarily due to credit return or the next *section* updating later in the same time instance. As such the **Commands** are ordered so that `updateCreditflow()` **Commands** that are to be called in that instance are executed first. **Commands** to `updateDataflow()` are ordered to ensure that *sections* further along the pipe from master to slave, and back again, are executed first. In order to facilitate the flow of data and credits the `execCommands()` function is executed every cycle. The **Commands** for that instance are subsequently executed in *order*.

In this Section it is contributed how the modelling theory, from Chapter 2, can be implemented as a model that can be integrated into an MPSoC simulation framework. In Section 3.3 it is demonstrated that application-level inter-IP synchronisation is not guaranteed by conservatively bounding transaction times.




(a) Generic **Token** section flowchart. (b) Generic **FIFO** section flowchart.

Figure 3.3: Section flowcharts.

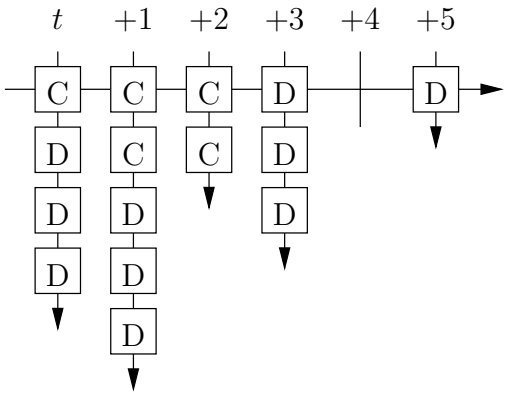
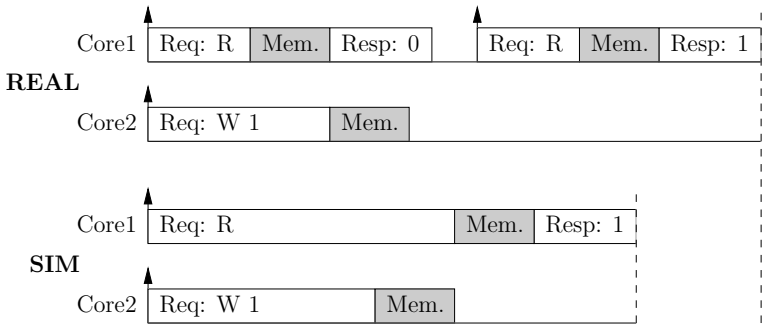
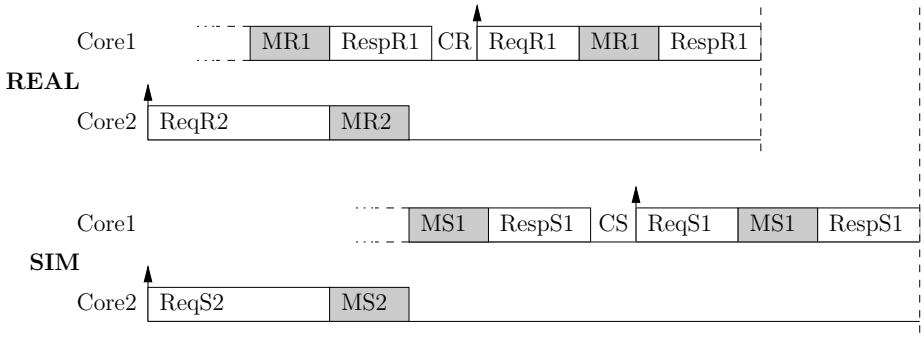



Figure 3.4: `CommandQ` structure obtained by daisy chaining **Command** objects.




(a) Core 1 polling for a value in external memory that Core 2 sets. Even though the individual transactions are conservatively bounded in simulation the synchronisation is not.




(b) Core 1 polling an extra time in simulation, upon receiving a positive poll, to conservatively bound the synchronisation.

Figure 3.5: Guaranteeing conservativeness at the application level when cores synchronise.

## 3.3 Application Level Inter-IP Synchronisation Conservativeness

In order to maintain application level conservativeness of the simulation, synchronisation between cores must also be conservatively bounded. A situation is illustrated in Figure 3.5a whereby even though the individual reads and writes are conservatively bounded, in simulation, the overall act of polling a value updated by another core is not. This may occur when two cores synchronise in shared memory. Even though transaction ordering is maintained in simulation between a core and the memory, the ordering of the transactions at the memory from multiple cores is not maintained.

In Figure 3.5a Core 1 is polling a value, that is initially 0, in the shared memory, and will stop upon a positive poll, when the value polled is 1. Core 2 writes the value 1 to shared memory. In the **REAL** example, the first poll arrives before the value 1 is written and therefore Core 1 must poll again. In the **SIM** example, the first poll arrives after the value is written to 1 and the first poll is therefore sufficient. As can be seen in Figure 3.5a, even though the read and write transactions, in the **SIM** example are bounded conservatively,

the poll terminates earlier than in reality. The synchronisation is therefore not conservatively bounded at the application layer. It is possible that any number of instructions could be executed in between polls making the timing discrepancy unbounded.

It is possible to bound the synchronisation conservatively, at the application level, by simply making the simulated version of the application poll one more time, upon receiving a positive poll. The writing of the polled value is conservatively bound in **SIM**, therefore any subsequent poll after a successful poll also conservatively bounds the possibility that the **REAL** implementation just missed the positive poll and had to poll again. An example of this is illustrated in Figure 3.5b. In order to prove this to be the case the following assertions are made:

1. All timing values exist in the $\mathbb{R}_{\geq 0}$ domain.
2. SIM transactions are never shorter than REAL transactions. This is also true of the timed elements that the transactions are composed of.
3. SIM write will never start earlier than REAL write.
4. SIM Time between polls is never shorter than REAL time between polls.

Assertions 1–3 are obvious observations of the properties of a conservative simulation. Assertion 4 is not inherently true in a conservative simulation. One method of ensuring that this is the case is by stipulating that only a fixed number of instructions may be executed between polls.

In order to maintain conservativeness; $SIM\ Sync\ Time \geq REAL\ Sync\ Time$.

To show that polling an extra time in simulation, upon receiving a positive poll, maintains conservativeness this must be shown to be true in all cases.

The worst case for conservativeness is when the polling core in REAL just misses, while the polling core in SIM just hits, the positive poll value. An extra poll is produced in SIM to conservatively bound the synchronisation. This produces the following synchronisation timings, using the naming from Figure 3.5b:

$$\begin{aligned} REAL\ Sync\ Time &= ReqR2 + RespR1 + CR + ReqR1 + MR1 + RespR1 \\ SIM\ Sync\ Time &= ReqS2 + MS2 + MS1 + RespS1 \\ &\quad + CS + ReqS1 + MS1 + RespS1 \end{aligned}$$

The worst case for conservativeness is also when SIM transactions and REAL transactions are equal in duration. Therefore cancelling out equivalent terms:

$$\begin{aligned} \cancel{ReqS2} + MS2 + MS1 + \cancel{RespS1} + \cancel{CS} + \cancel{ReqS1} + \cancel{MS1} + \cancel{RespS1} \geq \\ \cancel{ReqR2} + \cancel{RespR1} + \cancel{CR} + \cancel{ReqR1} + \cancel{MR1} + \cancel{RespR1} \end{aligned}$$

Leaves:

$$MS2 + MS1 \geq 0$$

This is always true due to Assertion 1; memory access times cannot be negative. Therefore polling an extra time in simulation, upon receiving a positive poll, will always cause the simulation synchronisation to conservatively bound the implementation synchronisation.

To ensure that synchronisation is performed in this manner, i.e. complies with Assertion 4, it is possible to incorporate the extra complexity in a communication API and stipulate that all inter-IP synchronisation should be carried out using it.

### 3.3.1 Communication API

C-HEAP, as described in [11], is an application level communication protocol that abstracts away from underlying hardware protocols. This permits the MPSoC application programmer to communicate data, between possibly heterogeneous components, without having to program specific communication routines for each platform.

This is achievable by creating C-HEAP protocol libraries for each platform, that enable the C-HEAP protocol on top of the platforms hardware communication protocol. These libraries not only aid in the programming of new applications but facilitate the porting of applications designed for other platforms that use the C-HEAP communication protocol.

The C-HEAP protocol works by communicating data through administrated circular buffers in shared memory, from the producing task to the consuming task, while maintaining FIFO ordering. The administration contains static values such as the token size, the start address of the buffer in memory, the maximum number of tokens, etc. The administration also contains the variables *readc* and *writec* that are used to track the buffer occupancy. Two variables are used in this case as the platform does not provide atomic read-modify-write access to memory. This means that it cannot be guaranteed that the variable is not written too by an external source between reading the value and writing it back again. By only allowing the producer and the consumer to modify one of the variables the issue of memory consistency is circumvented.

In [11] a set of standardised primitives were defined.

`claim_space` Claims space in the circular buffer to be written to by the producer. Illustrated by communication 1 in Figure 3.6.

`release_data` Releases the data so that it may be read by the consumer. Illustrated by communication 3 in Figure 3.6.

`claim_data` Claims data in the circular buffer to be read from by the consumer. Illustrated by communication 4 in Figure 3.6.

`release_space` Releases the space so that it may be written to by the producer. Illustrated by communication 6 in Figure 3.6.

In order to maintain a consistent record of the circular buffer's occupancy *readc* and *writec* are implemented as counters that count modulo the buffer capacity. Whenever the producer calls `release_data`, the *writec* variable is incremented. Whenever the consumer calls `release_space` the *readc* variable is incremented. The current buffer occupancy is the difference of the two variables.

To solve the ambiguous case that occurs when both of the values are equal (is the buffer full or empty) a bit in each of the variables can be devoted to be a wrap around flag, that toggles value each time the counter wraps around to 0.

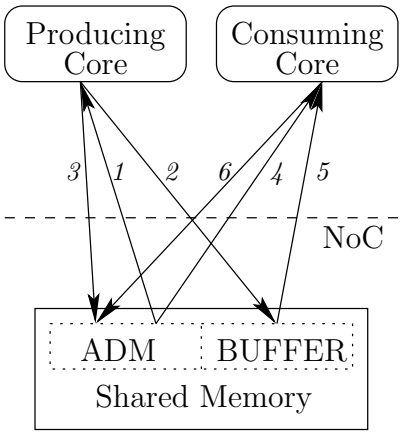



Figure 3.6: Transactions involved for the communication of data via the C-HEAP protocol. An arrow towards a core indicates a read. An arrow away from a core indicates a write.

In Figure 3.6 an example data transfer via the C-HEAP protocol is illustrated. In this example it is assumed that the static values in the administration have also been stored locally on both the producing and consuming cores. Action 1 illustrates the producer calling `claim_space`. This action reads the two counter variables, *readc* and *writec*, from the administration to ascertain if there is space in the buffer. The variables are polled until there is space in the circular buffer, at which point the pointer to the next slot in the circular buffer is returned. Action 2 is the producing core writing the data into the circular buffer at the address acquired through action 1. Now that the data is in the buffer the producing core calls `release_data` in action 3 incrementing the *writec* variable.

The consumer calls `claim_data` in action 4. Like `claim_space`, `claim_data` polls the variables, *readc* and *writec*. This time the variables are polled until there is found to be data in the buffer. When it is found that there is data in the buffer `claim_data` returns the address of the next piece of data in the buffer, maintaining FIFO ordering. Action 5 is the consuming core reading the data starting at the address obtained through action 4. Once the data has been read by the consuming core the space in the buffer can be freed again for use by the producer. The consuming core calls `release_space` incrementing the *readc* variable.

In order to conservatively bound the synchronisation step of the communication the blocking functions `claim_space` and `claim_data` are augmented to perform an additional poll upon receiving a positive poll in simulation, thus bounding the possibility that the positive poll was just missed in reality and needed to be polled for again, as described in Section 3.3. Data can be communicated through the circular buffers as normal as they are guarded by the synchronisation steps.

By separating the concerns of synchronisation and communication in this manner, it is possible to amortise the extra temporal contribution in simulation of the extra poll by increasing the amount of communicated data between synchronisations. The effect of synchronisation granularity on the timing results from simulation in comparison to the FPGA implementation is investigated in Section 4.2.2.

In this Section it is shown that conservatively bounding transaction times is not sufficient to conservatively bound inter-IP synchronisation on the application level. A solution is presented to conservatively bound the synchronisation. To simplify the implementation of the solution it is demonstrated that it can be incorporated into an application-level communication API. In Section 3.4 the rationality behind how application-level conservative guarantees can be generated through simulation using the implemented NoC model and the augmented C-HEAP communication API is described.

## 3.4 Application-Level Conservative Guarantees

In order for the simulation to provide conservative timing guarantees the application runtime generated through simulation must never be less than the implementation application runtime. If the application's runtime is monotonic in comparison to the duration of influencing factors such as computation, inter-task synchronisation and inter-task communication then the application will be bounded conservatively in simulation if these factors are conservatively bounded. The monotonicity of these three factors for the system described in this thesis is explained as follows:

1. Application instructions are monotonic in duration. Instructions dependent on NoC transactions are also monotonic, i.e if the NoC transaction time increases the instruction time will not decrease and similarly if the NoC transaction time decreases the instruction time will not increase.

2. Blocking inter-IP synchronisation at the application-level is monotonic in duration if the Assertions detailed in Section 3.3 are met. The conditions are met if synchronisation is carried out using the augmented C-HEAP communication protocol. An increase in synchronisation time will not decrease the application runtime and similarly a decrease in synchronisation time will not increase the application runtime.

3. Application-level inter-IP communication is monotonic if carried out using explicit synchronisation. This is achievable using the C-HEAP communication protocol, e.g. a synchronisation, followed by one or more write, followed by a synchronisation, followed by one or more reads is monotonic due to 2, 1, 2 and 1 respectively.

There are some application-level stipulations in 2 and 3 that must be followed to maintain the monotonicity of the application in comparison to synchronisation and communication. This is easily achieved by requiring all application-level inter-IP synchronisation and communication are carried out using the modified C-HEAP communication protocol described in Section 3.3. Conservative simulation can be achieved by bounding these three influencing factors on application runtime, which can be explained as follows.

1. Application instructions are conservative in duration. Instructions dependent on NoC transactions are also conservative, i.e SIM_instruction $\geq$ REAL_instruction comparing their durations.

2. Blocking inter-IP synchronisation at the application-level is conservative in duration if the Assertions detailed in Section 3.3 are met and the measures described to bound the synchronisation are implemented. The conditions are met if synchronisation is carried out using the augmented C-HEAP communication protocol. At the application-level SIM_sync $\geq$ REAL_sync comparing their durations.

3. Application-level inter-IP communication is conservative if carried out using explicit synchronisation. This is achievable using the C-HEAP communication protocol, e.g. a synchronisation, followed by one or more write, followed by a synchronisation, followed by one or more reads is conservative due to 2, 1, 2 and 1 respectively. At the application-level SIM_comm $\geq$ REAL_comm comparing their durations.

4. Distributed tasks execute, synchronise and communicate conservatively making the application conservative if there are no influences that negatively effect the determinism of the execution. Non-determinism such as a dependency on random numbers, synchronisation with a non-deterministic hardware component, clock dependent operations may cause the application to execute differently in simulation than on the implementation destroying the conservativeness guarantee. At the application-level SIM_app_runtime $\geq$ REAL_app_runtime comparing their durations.

Another application-level stipulation to maintain conservativeness in simulation in 4, is that the application must execute deterministically. This is to ensure that the simulation is conservatively bounding the same trace as the implementation. By conservatively bounding computation, inter-IP synchronisation and inter-IP communication in simulation the application runtime is conservatively bounded if the application executes deterministically.

In Chapter 4 case study applications are simulated using the NoC model described in this described in this Chapter. Timing results of the simulations are compared with results from the FPGA implementation to ascertain if the model is useful for conservative application-level performance analysis through simulation of an MPSoC.

# Chapter 4

# Case Studies

In this chapter the model introduced in the previous chapters is applied to multiple case studies with the aim of evaluating the model's usefulness. The case studies are carried out on the target system, as described in Section 1.3.

The case studies are broken down into two categories. The Artificial case studies make use of specifically designed applications that expose underlying behaviour that may be more difficult to isolate in more complex real life applications. The real life case studies highlight the applicability of the system to an existing problem in the form of a JPEG decoder.

In this Chapter an example is given, in Section 4.1, of LR value derivation for a TDMA arbitration table from the implementation. The artificial case studies are presented and evaluated in Section 4.2, illustrating the NoC models behaviour. The JPEG decoder case studies are presented and evaluated in Section 4.3, demonstrating that the model can generate conservative performance bounds without requiring an application model.

## 4.1 Example of LR Value Derivation

Taking as an example the NI slot tables for the connection between Core 1 and External Memory. The request NI slot table for the connection is as illustrated in Figure 4.1 . The data arbitration of the table can be represented as illustrated in Figure 4.2. In this figure only the data slots that may be used after an idle period are shown.

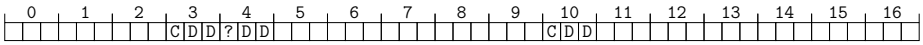



Figure 4.1: Request Table

First the Latency and Rate are calculated by applying the CSS method of calculation. For the inverse rate calculation, `?` slots are taken to be `D` slots as this would be the case during a sustained busy period. The table contains 17 slots each containing 3 service slots giving a period $P = 51$. The table contains 7 data service slots making $S_D = 7$. Equations 2.3 and 2.7 can then be applied as follows.

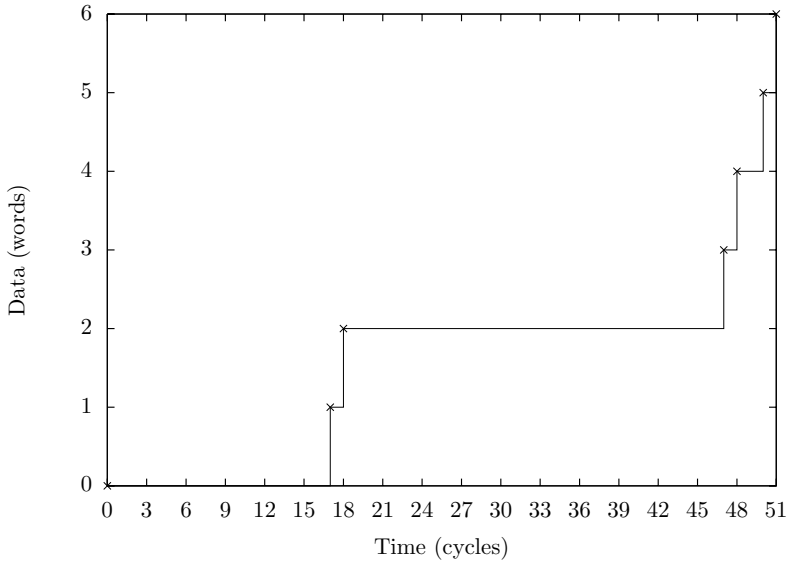



Figure 4.2: Example slot table with distributed service slots.

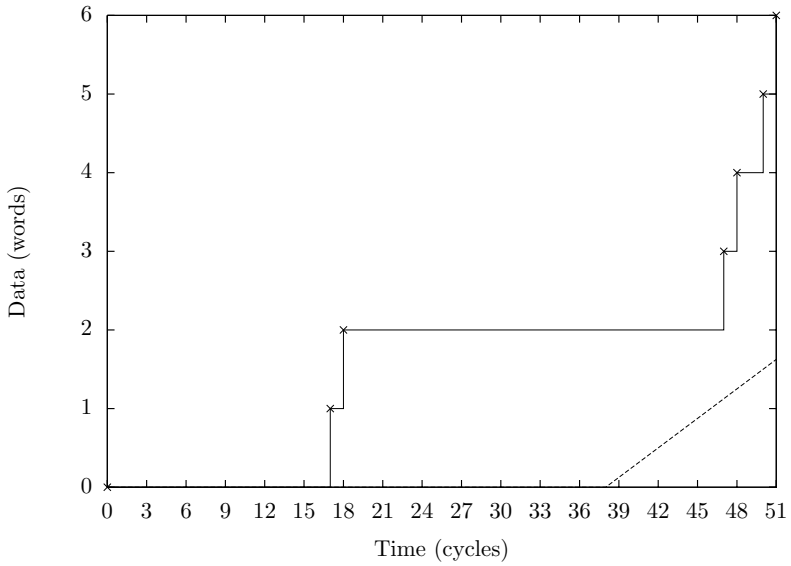



Figure 4.3: All points are conservatively bounded although the latency value is unnecessarily large.

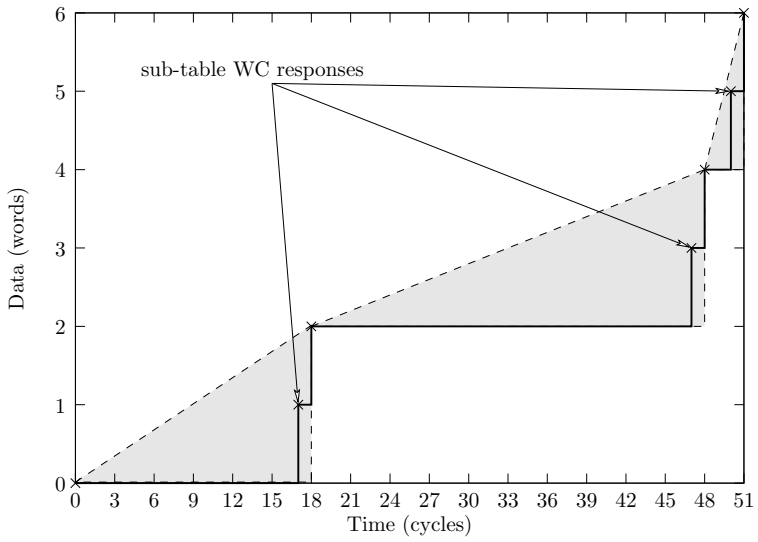



Figure 4.4: Splitting the table into multiple sub-tables, as illustrated by the dashed triangles.

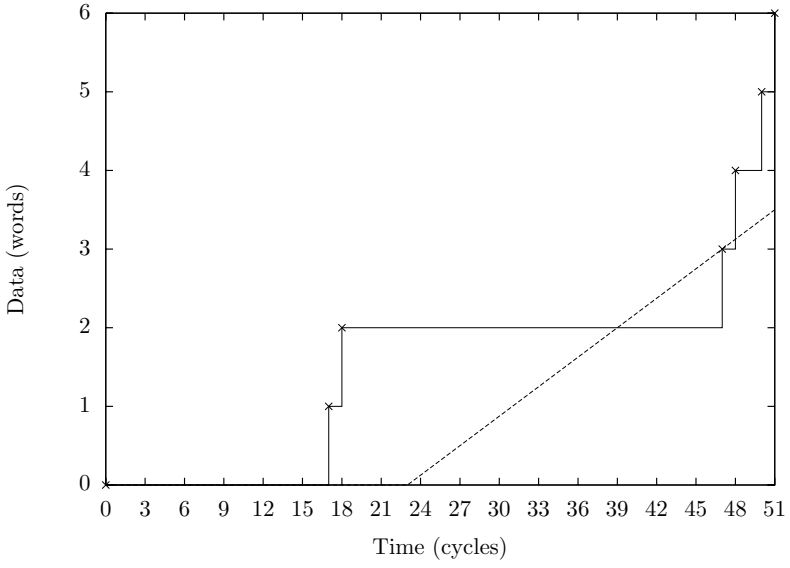



Figure 4.5: The TDMA table's sustained rate is still provided conservatively while the latency is reduced compared to the CSS method.

$$\rho_D^{-1} = \frac{P}{S_D} \leq \left\lceil \frac{P}{S_D} \right\rceil = \left\lceil \frac{51}{7} \right\rceil = 8 \text{ cycles per word}$$

The rounding up of the inverse rate can be taken into account in the Latency equation as the Latency does not have to be as long before the more conservative rate can be sustained.

$$\begin{aligned}
\theta_x &= 1 + (P - S_x) - \frac{P}{S_x} \\
&\geq 1 + (P - S_D) - \left\lceil \frac{P}{S_D} \right\rceil \\
&\geq 1 + (51 - 7) - \left\lceil \frac{51}{7} \right\rceil = 37 \text{ cycles}
\end{aligned}$$

The resultant bounding curve for the CSS LR derivation method is illustrated in Figure 4.3. As can be seen the latency derived is unnecessarily overly conservative. For the DSS method the sustainable rate stays exactly the same. The only difference from the CSS method is the observation that the latency required before the rate is sustainable is calculated assuming that all the service slots are in one continuous group in the table. To take account of the slot distribution's affect on the worst case response time, the table can be visualised as being split into smaller sub-tables that have continuous service slots at the end. This is illustrated in Figure 4.4 using triangles to bound the CSS sub-tables. The worst case latency for each sub-table can then be calculated by applying Equation 2.7 while taking the inverse rate to be 8 cycles.

For each CSS sub-table from $n \rightarrow m$ the latency $\theta_D(n \rightarrow m)$ and latency offset $\delta_D(n \rightarrow m)$ is calculated.

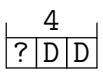



Figure 4.6: CSS sub-table 4

For the CSS sub-table illustrated in Figure 4.6, the table's period and number of data service slots are used to calculate the worst case response time in the same manner as the CSS method.

$$\begin{aligned}
P_{(4)} &= 3 \\
S_{D(4)} &= 2 \\
\bar{r}_{D(4)} &= 1 + 3 - 2 = 2 \text{ cycles}
\end{aligned}$$

The worst case response time is subsequently used to calculate the latency in the same manner as the CSS method. The DSS method also requires the latency offset to be calculated for the sub-table to account for the differing average rates between the DSS table and the CSS sub-table.

$$\begin{aligned}
\theta_{D(4)} &= 2 - 8 = -6 \text{ cycles} \\
\delta_{D(4)} &= 3 - 8 \times 2 = -13 \text{ cycles}
\end{aligned}$$

This is repeated for the rest of the CSS sub-tables to obtain the latency and latency offset values.

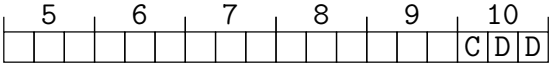



Figure 4.7: CSS sub-table 5–10

$$
\begin{aligned}
P_{(5\rightarrow 10)} &= 18 \\
S_{D(5\rightarrow 10)} &= 2 \\
\bar{r}_{D(5\rightarrow 10)} &= 1 + 18 - 2 = 17 \text{ cycles}
\end{aligned}
$$

$$
\begin{aligned}
\theta_{D(5\rightarrow 10)} &= 17 - 8 = 9 \text{ cycles} \\
\delta_{D(5\rightarrow 10)} &= 18 - 8 \times 2 = 2 \text{ cycles}
\end{aligned}
$$

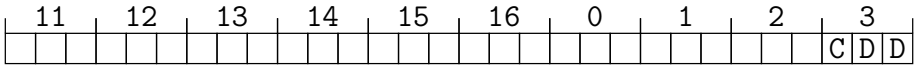



Figure 4.8: CSS sub-table 11–3

$$
\begin{aligned}
P_{(11\rightarrow 3)} &= 30 \\
S_{D(11\rightarrow 3)} &= 2 \\
\bar{r}_{D(11\rightarrow 3)} &= 1 + 30 - 2 = 29 \text{ cycles}
\end{aligned}
$$

$$
\begin{aligned}
\theta_{D(11\rightarrow 3)} &= 29 - 8 = 21 \text{ cycles} \\
\delta_{D(11\rightarrow 3)} &= 30 - 8 \times 2 = 14 \text{ cycles}
\end{aligned}
$$

In order to bound the worst case data arbitration the combined effects of the CSS sub-tables must be taken into account. The CSS sub-tables are used in a sequential manner reducing the combinations that must be examined. Table 4.1a displays the latency calculations for the possible sequential CSS sub-table combinations.

The maximum latency in Table 4.1b is the conservative latency for the entire table. There is only one start to end point combination that requires the latency to be 23 cycles. Working back from its position in the Table 4.1b it can be deduced that this latency is required to conservatively bound when 3 words of data start to enter at the start of slot 5.

For this table the tightest Latency $\theta_D$, while still remaining conservative for a sustained inverse rate $\rho_D^{-1} = 8$ cycles per word, is $\theta_D = 23$ cycles. It can be seen in Figure 4.5 that the DSS derived latency is conservative while not being overly conservative as in the CSS method in Figure 4.3.

| | CSS $(5 \rightarrow 10)$ | CSS $(11 \rightarrow 3)$ | CSS $(4)$ |
|---|---|---|---|
| CSS | $\theta_{D(5\rightarrow10)}$ | $\theta_{D(11\rightarrow3)}$ | $\theta_{D(4)}$ |
| CSS +1 | $\delta_{D(5\rightarrow10)} + \theta_{D(11\rightarrow3)}$ | $\delta_{D(11\rightarrow3)} + \theta_{D(4)}$ | $\delta_{D(4)} + \theta_{D(5\rightarrow10)}$ |
| CSS +2 | $\delta_{D(5\rightarrow10)} + \delta_{D(11\rightarrow3)} + \theta_{D(4)}$ | $\delta_{D(11\rightarrow3)} + \delta_{D(4)} + \theta_{D(5\rightarrow10)}$ | $\delta_{D(4)} + \delta_{D(5\rightarrow10)} + \theta_{D(11\rightarrow3)}$ |

(a) Table of latency calculations for all possible sequential combinations of CSS sub-tables.

| | CSS $(5 \rightarrow 10)$ | CSS $(11 \rightarrow 3)$ | CSS $(4)$ |
|---|---|---|---|
| CSS | 9 | 21 | $-6$ |
| CSS +1 | 23 | 8 | $-4$ |
| CSS +2 | 10 | 10 | 20 |

(b) Table of latencies resulting from the calculations in Table 4.1a. The maximum value is the conservative latency for the table.

Table 4.1: Algorithmic latency value derivation for a DSS TDMA table. CSS sub-tables are represented horizontally and sequential combinations of CSS sub-tables are represented vertically, e.g. CSS +1 indicates the CSS sub-table in the horizontal domain in combination with the following CSS sub-table.

## 4.2 Artificial Case Studies

In this section two case studies are presented that use specially crafted applications to evaluate the NoC model's behaviour through simulation. In Section 4.2.1 a specially crafted application, that tests the time to read or write an array of data over a single connection, is simulated and the results evaluated in comparison to the application run on the FPGA implementation. In Section 4.2.2 an application using the augmented C-HEAP communication API, described in Section 3.3.1, is used to investigate the effect of the choice of C-HEAP Token size on the communication time. The effect of synchronisation granularity on simulation accuracy is also evaluated.

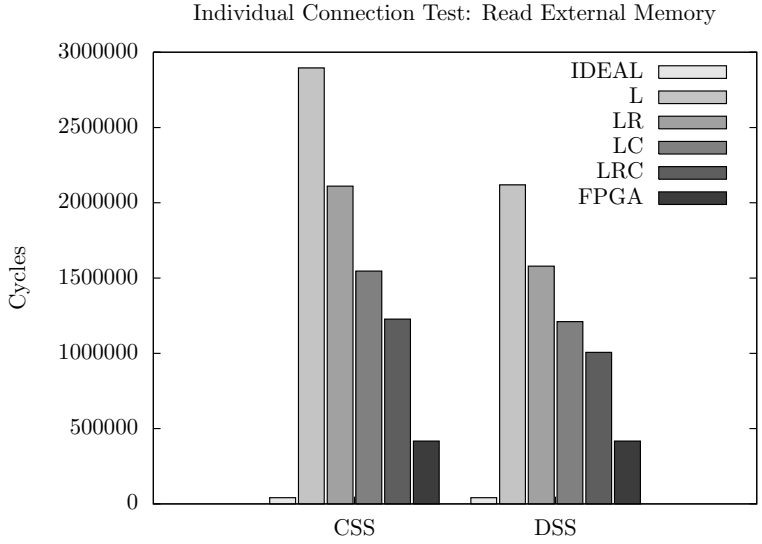
### 4.2.1 Connection Tests

This case study exercises individual connections by carrying out relatively large amounts of the same transaction type over a single connection at a time. A common occurrence of this in real life applications is the reading or writing of arrays to or from external memory. As can be seen in Figure 1.2b only Core 1 has a connection with the Timer peripheral. As such, only the connections associated with Core 1 shall be tested.

The test is performed by Core 1 reading an array of integer values sequentially from external memory. The timer peripheral is read before and after the transmission of the array and the difference taken as the total transmission time. The overhead incurred reading the timer is also conservatively bounded and is negligible compared to the array IO operations. The test is repeated for writing an array of the same size to external memory and the framebuffer. The simulations are carried out using LR values that were calculated using both the CSS and DSS method.

The bar chart in Figure 4.9a depicts the results for reading the array from external memory. The descriptions in the bar chart's legend are defined as follows:

**IDEAL** Zero NoC delay. Transactions across the NoC happen instantly.




(a) Bar chart depicting the timing results for reading the array from external memory.

Figure 4.9: Results of individual connection tests.

**L** Connections modelled as in Figure 2.1 using the NI model as described in Figure 2.6a.

**LR** Connections modelled as in Figure 2.1 using the NI model as described in Figure 2.6b.

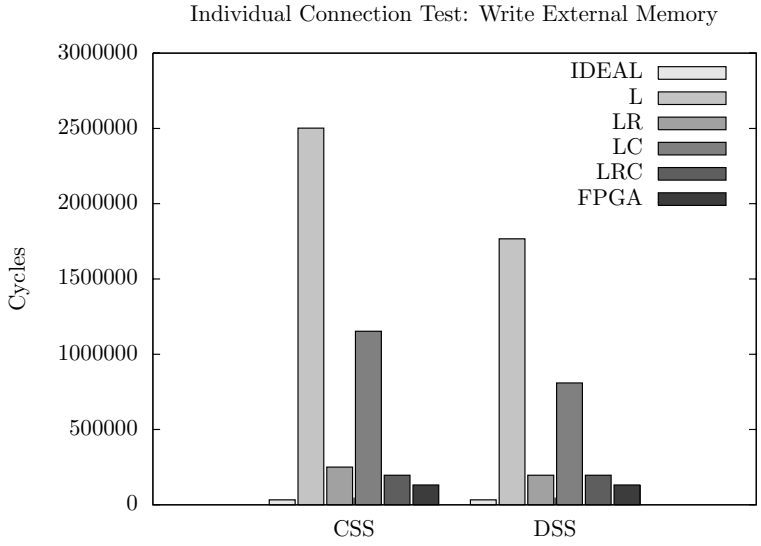
**LC** Connections modelled as in Figure 2.1 using the NI model as described in Figure 2.6c.

**LRC** Connections modelled as in Figure 2.1 using the NI model as described in Figure 2.6d.

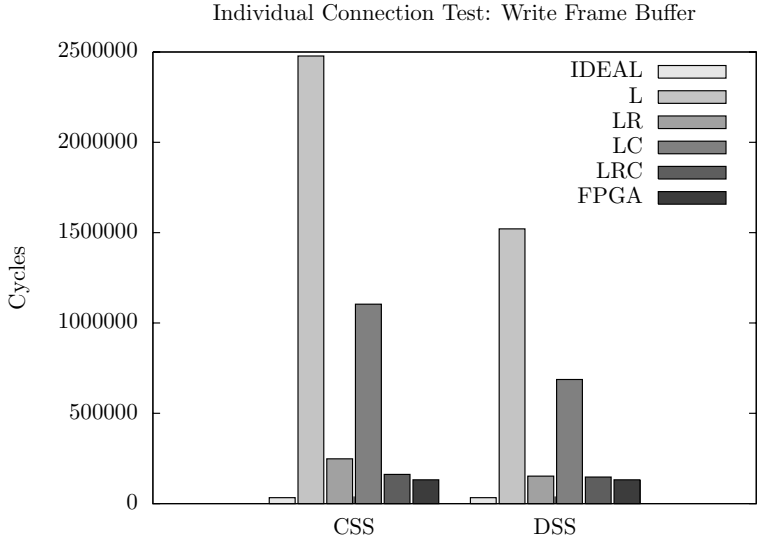
**FPGA** Implementation of the system.

To read values from external memory the transaction has to follow the entire path of the connection model, illustrated in Figure 2.1. After the core puts the read request onto the Master side bus it stalls until the data is returned from the slave before continuing with any processing operations. This has the effect that after a read the entire pipe from core to slave and back to core again will always be empty.

The action of reading an array from external memory is carried out by a series of read transactions. Read transactions are encapsulated at the message level as two word sized message headers, on the request channel. The headers are generated by the shell component. Both headers arrive in the producing NI within one cycle of each other. The later header is queued until the earlier header has been serviced by the LR server. The NI LR model illustrated in Figure 2.6c that models Latency and Rate combined, along with Credit return, starts the clock for the Latency component of the LR server, for the later header, whenever the earlier header has left the NI. The NI LR model illustrated in Figure 2.6d




(b) Bar chart depicting the timing results for writing the array to external memory.




(c) Bar chart depicting the timing results for writing the array to the frame buffer.

Figure 4.9: Results of individual connection tests.

that models Latency and Rate separately, along with Credit return, starts the clock for the Latency component of the LR server, for the later header, as soon as it enters the producing NI buffer, producing a tighter conservative delay. The effect of this on the simulation times can be seen in Figure 4.9a as the difference between the LC and LRC values.

Depending on the specific NoC LR timings, the first read header phit is likely to encounter the producing NI's LR server during an idle period. This means that it is likely that it will incur the full conservative delay of the Latency and Rate combined. The second header phit will encounter the producing NI's LR server during a busy period allowing it to effectively pipeline its Latency while the first header is being serviced. Producing NI models that do not use LR servers, i.e. L and LC, cannot pipeline the Latency component of the LR values in this way.

The bar charts in Figures 4.9b and 4.9c depict the results of writing the array to the external memory and the frame buffer respectively. As opposed to reads, once the core puts the write request onto the Master side bus it can continue with processing operations. The core will only stall in this case if the Master side bus is not able to accept the transaction immediately. This will occur if the Master side bus is still processing the previous transaction, but can also occur if there is a backlog of transactions due to a full buffer at the Master side NI.

Writes are encapsulated at the message level as two phit sized headers followed by the phit sized data. As the request channel's producing NI buffer is not as likely to be empty, during a series of writes and depending on specific NoC LR timings, writes are more likely to encounter the NI's LR server during a busy period allowing the Latency component of the LR values to be pipelined. If a series of writes arrive with a high enough rate, creating a busy period, then only the first phit incurs the penalty of the Latency, for NIs that model Latency and Rate separately. The larger the series of phits, serviced during a busy period, the more the Latency, incurred by the first phit, is amortised.

The effect of the pipelining the Latency overhead is reflected in an increased throughput at the NoC. This in turn reduces the possibility of backlogs, starting at the producing NI. This means that the core will have to stall less frequently than for the producing NI models that model Latency and Rate separately. The ability of the producing NI models, that use LR servers, to pipeline the Latencies of phits in the buffer creates the relatively large contrast in results between the producing NIs that model Latency and Rate separately (L & LC) and the producing NIs that model them together (LR & LRC) as shown in Figures 4.9b and 4.9c.

The simulations using the LR values generated using the algorithmic DSS method show a marked improvement over those using the values from the analytical CSS method. The DSS method is explained in detail for the TDMA slot table, for the producing NI, on the request channel of the Core 1 to external memory connection, in Section 4.1. Even though the L and LC models do not use LR servers to simulate their NIs, the LR values are still used to calculate their delays $\tau_x$, as described in Figure 2.6. The bar chart for reading the array, illustrated in Figure 4.9a, shows a considerable reduction in timings for all producing NI models, when using LR values derived using the DSS method. The reduction in timing is also markedly noticeable, for producing NIs that model Latency and Rate together, for writing the array, as illustrated in Figures 4.9b

and 4.9c. There is also a less evident reduction in timing, for writing the array, using the producing NI models that use LR servers. The timings for these models were already quite conservatively tight with the FPGA timings.

The data from this case study indicates that the NoC model functions as expected. It highlights the fundamental differences in how read and write transactions experience the model and the LR abstractions. Reads are more conservatively bound than writes due to the ability of the platform to pipeline write transactions allowing multiple writes to be traversing the NoC at any one time. Reads cause the core to stall until the data returns and experience conservative LR models at producing NIs on the request and response path and at the Slave IP arbitration on the multibus.

### 4.2.2 Communication API Case Study

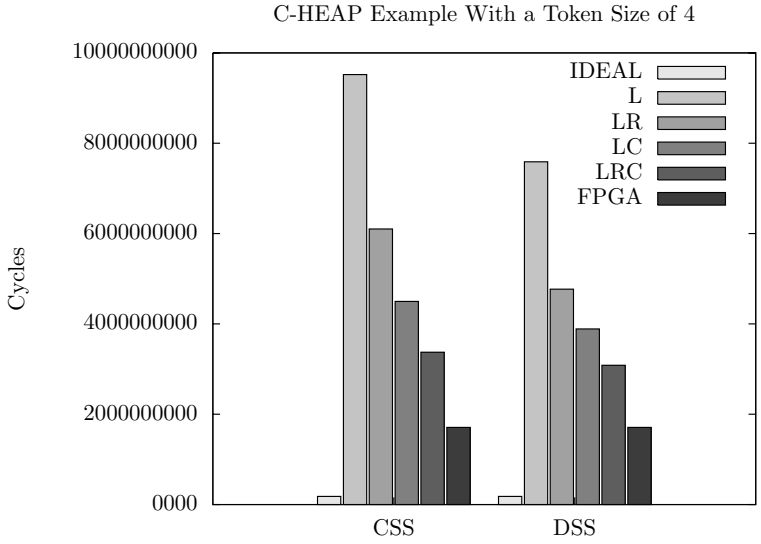
Interprocessor communication is achieved in this system through shared memory. Shared memory communication is used as a result of the limitations of the timing model for the arbitration of the local memories in the Pearl-Ray cores.

C-HEAP, as described in Section 3.3.1, is a communication protocol that uses administrated circular buffers in shared memory to facilitate the transfer of data. There is a certain amount of overhead involved in reading the status of the administration and subsequently updating it relating to the performed action. The purpose of this case study is not only to demonstrate a working C-HEAP example but also to demonstrate the effect synchronisation granularity has on the performance of the application in simulation and on the implementation. This case study uses the Processing Cores 1–3 set with the cores assigned to be a producer, intermediary and consumer respectively. The producing core sets the Timer to zero, then pushes data via the C-HEAP protocol to the intermediary core. The intermediary core pushes the received data via the C-HEAP protocol to the consuming core. Once all the data has been transferred from producer to consumer the consumer sets a flag in external memory so that the Core 1 can stop the Timer.

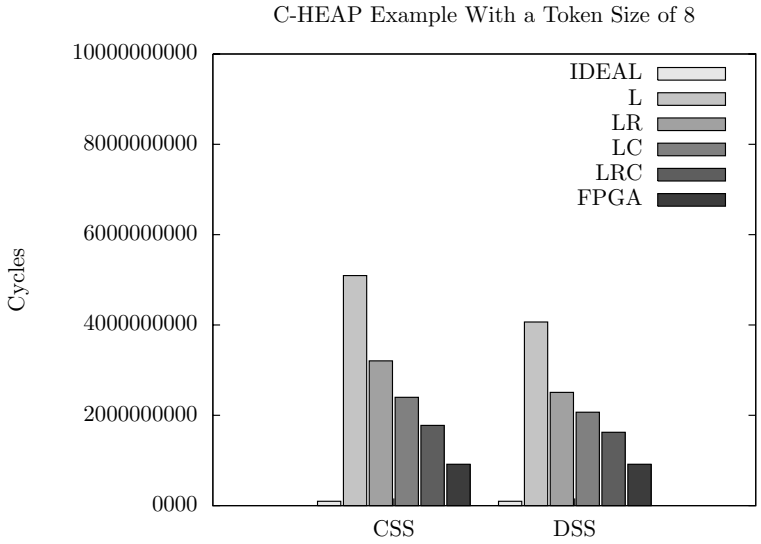
The test is performed for tokens of size 4, 8, 16 and 32 bytes. It was not possible to use token sizes greater than this due to an unresolved tooling error. While the token sizes are changed the total data transferred and the C-HEAP buffer sizes in bytes remain the same. This will help isolate the administration overhead associated with the C-HEAP protocol.

The C-HEAP protocol, as described in Section 3.3.1, uses two counters *readc* and *writec*, to maintain the current position, and occupancy, within the circular buffer. In order to ascertain the occupancy of the C-HEAP buffer the difference between the two variables must be calculated. For the producing and the consuming cores this is captured as the functions `claim_space` and `claim_data` respectively. As is shown in Section 4.2.1 reads are blocking. This means that the next transaction entering the pipe will encounter the LR server during an idle period and incur the delay of the latency before it is serviced by the rate. Smaller token sizes mean more tokens need to be transmitted to transfer the data. This in turn means that more synchronisation is required for the same amount of data transferred.

It can be observed from Figure 4.10 that the shape of the graphs are similar to Figure 4.9a in Section 4.2.1. This indicates that reads in the C-HEAP example are the dominant factor in the overall simulation time. This makes sense as

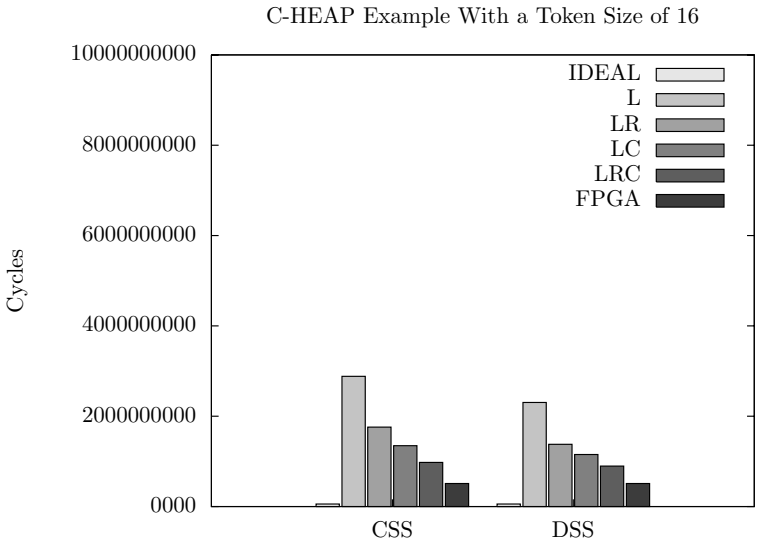



(a) Bar chart depicting the timing results for the C-HEAP communication experiment with a buffer size of 4 bytes.

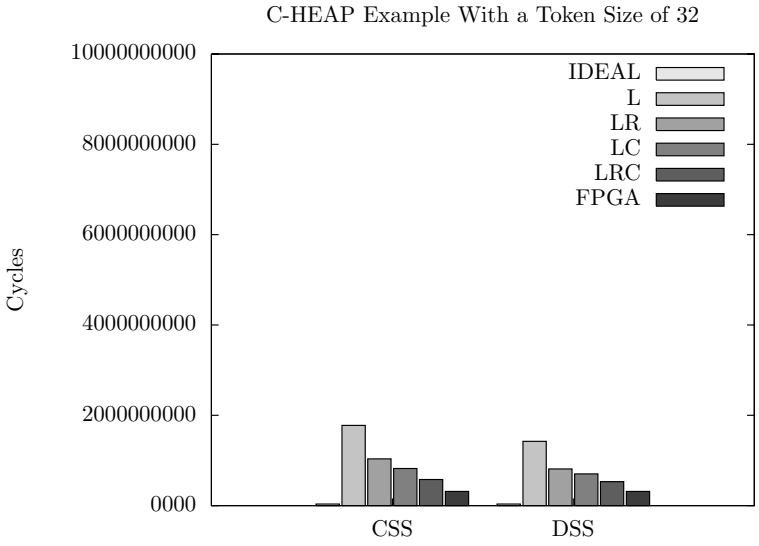



(b) Bar chart depicting the timing results for the C-HEAP communication experiment with a buffer size of 8 bytes.

Figure 4.10: Results of C-HEAP token sizing tests.




(c) Bar chart depicting the timing results for the C-HEAP communication experiment with a buffer size of 16 bytes.




(d) Bar chart depicting the timing results for the C-HEAP communication experiment with a buffer size of 32 bytes.

Figure 4.10: Results of C-HEAP token sizing tests.

for every write made by the producer, to shared memory, there has to be an equivalent read by the consumer. In Section 4.2.1, in Figure 4.9, it can be seen that reading data from shared memory takes much longer than writing the same amount of data to shared memory.

A better solution (providing sufficient local memory is available), both in reality and for the accuracy of the simulation, would be to locate the C-HEAP buffer in the local memory of the consuming core. Both the producer and the consumer keep a local copy of the administration. When the producing core calls `release_data` the local and remote copies of *writec* are incremented. The same principle is applied to *readc* when the consuming core calls `release_space`. In this manner both cores only need to read the variables from the administration located in local memory, when trying to ascertain buffer occupancy, forgoing the lengthier reads over the NoC. The transmission of data from producer to consumer is carried out over the NoC entirely using writes. The consuming core reads the transmitted data out of its local memory.

In Section 4.2.1 the effects of the differences between the read and write transactions were shown. Writes were seen to be closer to reality in simulation than reads. This means that applications using the more desirable C-HEAP configuration would produce results in simulation closer to reality.

It is described in Section 3.3 how the C-HEAP API could be modified to poll once more upon the receival of a positive poll in simulation in order to conservatively bound the application-level. While not apparent from the results graphs, in Figure 4.10, the ratio of the FPGA timing to the LRC timing increases from 55% for a 4 byte token, 56% for an 8 byte token, 57% for a 16 byte token to 59% for a 32 byte token. The proportion of the FPGA timing increases as the temporal effect of the extra poll in simulation is amortised by the increase in synchronisation granularity due to the increased token size.

This case study shows that increasing synchronisation granularity has the effect of decreasing the overal communication time in both simulation and on the FPGA implementation. It is also shown that increasing the synchronisation granularity has the effect of amortising the temporal effect of the required extra poll in simulation.

## 4.3 JPEG Decoder Case Studies

A JPEG decoder is used to demonstrate a real-life application that can be modelled using the dataflow analysis technique. JPEG is the format of choice for the storage of photographic images as its compression techniques use knowledge of the human visual system to more heavily compress less important parts of the image. Many steps in the JPEG decoding process, as illustrated in Figure 4.11, are similar to those used to decode video formats such as MPEG.


Figure 4.11: JPEG decoding steps.

The JPEG decoding steps can be mapped onto different processors to create a functional partitioning of labour. The entire decoding process can be mapped

to multiple cores with each core decoding a different part of the image, creating a data partitioning of labour. The cores work independently of each other under a data partitioning of labour with as a consequence that all cores must perform the RLD step on the image. As a consequence all the cores need to read the entire JPEG from memory.

By decoding different images different workloads on the cores and the NoC can be achieved. e.g. An image with a relatively large amount of high frequency information will have a relatively large file size to be read by the cores over the NoC and will also take longer to decode during the RLD step.

The following two case studies both demonstrate a data partitioned JPEG decoder.

### 4.3.1 3 Core Parallel JPEG Decoder

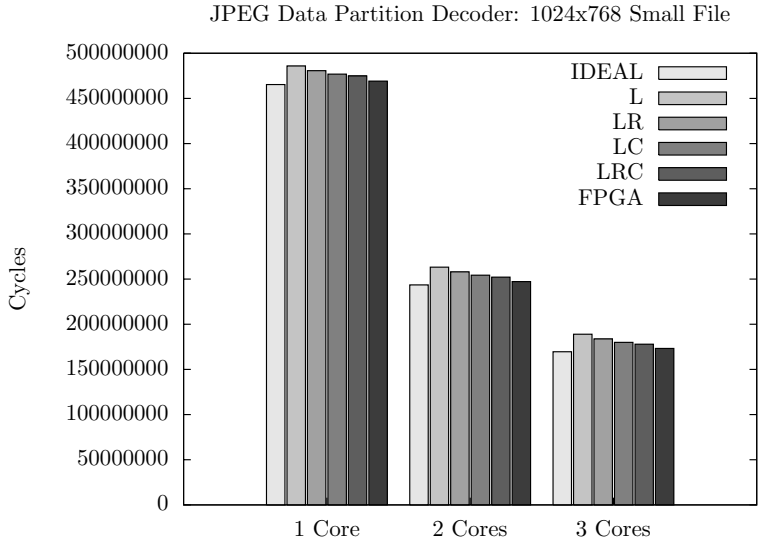
In this case study a JPEG decoder is mapped onto the system that is described in Figure 1.2a. From the connection map in Figure 1.2b it can be seen that only cores 1–3 have the appropriate connections between the host, shared memory and the frame buffer. The entire JPEG decoding algorithm is mapped onto each of the cores 1–3. The VLD step of the decoding algorithm can not be easily data partitioned. This results in all of the cores having to read the image in its entirety from shared memory. The image "blocks" used for the JPEG encoding are used as unit elements for sharing the workload among the cores. The cores follow a counting system, modulo the number of cores, to ascertain which "blocks" they are to decode.

Two JPEG images were used to test the system. Both of the JPEGs have the same dimensions of $1024 \times 768$. One of the JPEGs is more detailed and therefore is compressed to a relatively large file size. The other JPEG had much less detail and is therefore compressed to a relatively small file size. For each of these images simulations were performed using CSS and DSS calculated LR timings.

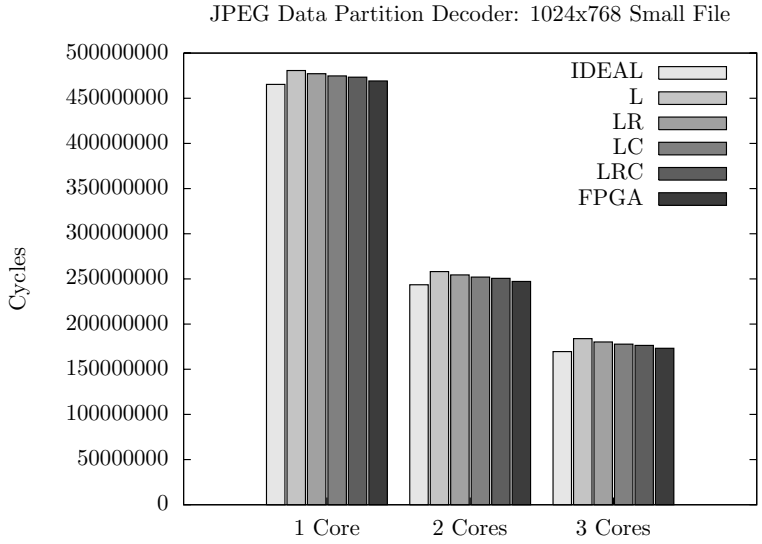
The delay due to NoC communication can be seen as the difference between the **IDEAL** and the **FPGA** results. The **IDEAL** simulation cycle accurately models the computation, for the JPEG decoding, but not the NoC communication. In Figure 4.12 the resultant bar charts, for both JPEG files, show that relatively little of the decoding time is due to communication over the NoC. This is to be expected as the computation involved in decoding a JPEG is relatively complex. The larger NoC time difference for the relatively large file, in comparison to the relatively smaller file, is simply due to the larger file needing to be read over the NoC from shared memory.

As is discovered, in Section 4.2.2, the shape of the resultant bar charts are similar in shape to the read bar chart in Figure 4.9a indicating that the delay caused by reading over the NoC dominates the writes over the NoC. This is not an obvious result as the compressed JPEG that is read from shared memory should be smaller than the decoded image written to the frame buffer. This appears to be another illustration of how much longer reads take in comparison to writes.

The resultant bar charts in Figure 4.12 show that the model can be used to perform conservative application-level performance analysis through simulation through simulation of an MPSoC.

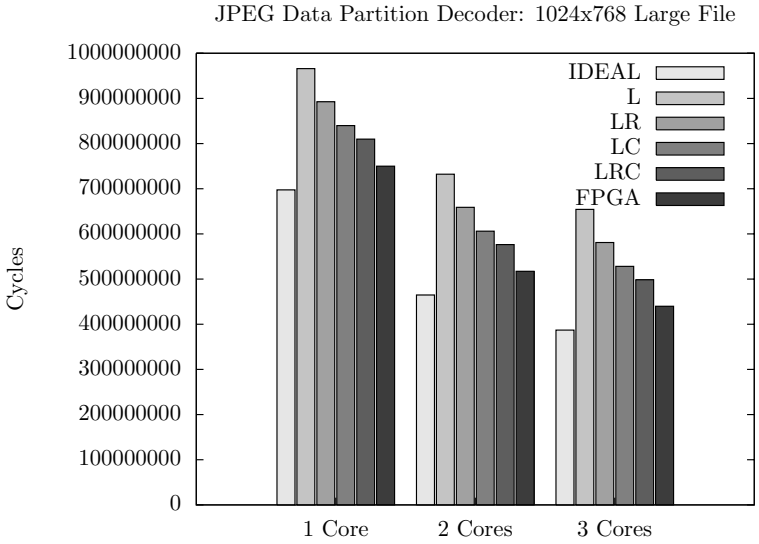



(a) Timings generated assuming Continuous Service Slots in the slot tables.

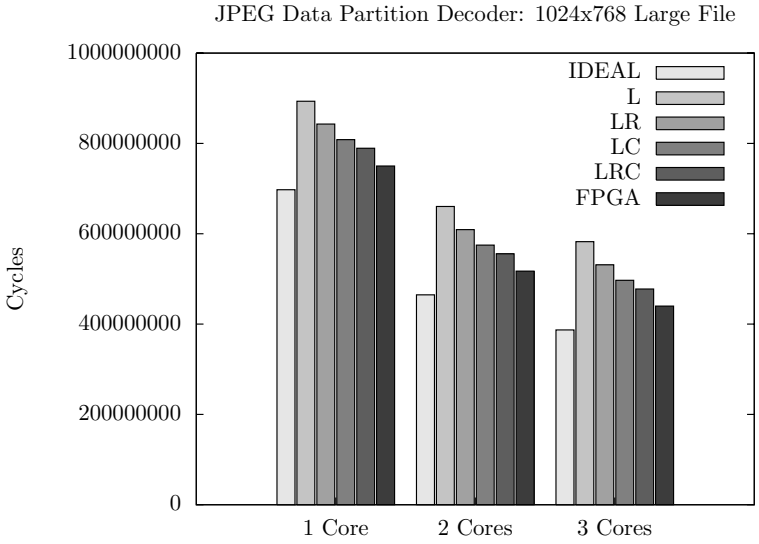



(b) Timings generated taking into account Distributed Service Slots.

Figure 4.12: Bar charts illustrating the conservative timing data for traces generated using a data parallel jpeg decoder, decoding a $1024 \times 768$ JPEG with a relatively small file size.




(c) Timings generated assuming Continuous Service Slots in the slot tables.




(d) Timings generated taking into account Distributed Service Slots.

Figure 4.12: Bar charts illustrating the conservative timing data for traces generated using a data parallel jpeg decoder, decoding a $1024 \times 768$ JPEG with a relatively small file size.

### 4.3.2 5 Core Parallel JPEG Decoder

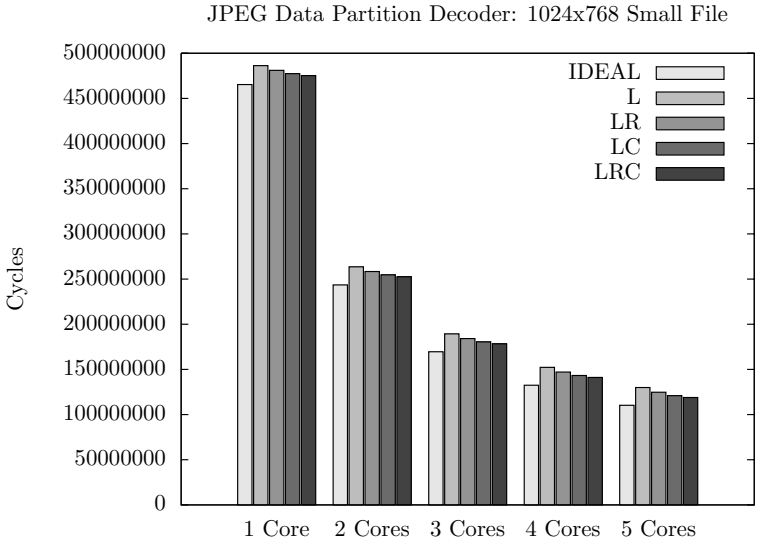
This case study, as in the case study in Section 4.3.1, maps a data partitioned JPEG decoder onto the system illustrated in Figure 1.2a. Instead of mapping it on Cores 1–3 it is mapped on Cores 1–5. The dashed lines in Figure 1.2b are connections that do not exist in reality but can be created by adding LR data for the connections, into the XML data file that contains the derived LR values for the connections. For this case study similar LR values to the relative connections to and from the other cores were input into the XML file. The LR values for the arbitration at the memory and the frame buffer were modified appropriately to take into account the extra connections.

This case study shows that a theoretical platform connection configuration can be created and tested as an alternative to creating and testing an FPGA implementation. Results from the JPEG decoding simulations, using the NoC model, are conservative compared to an equivalent real system.

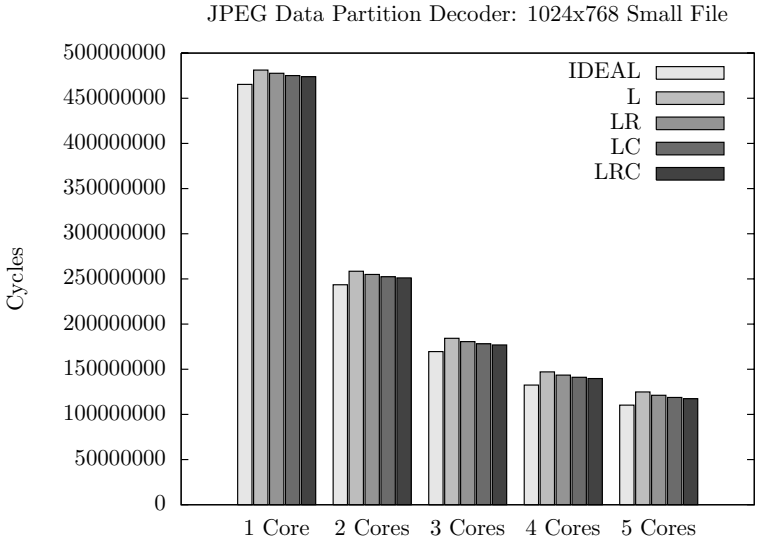
As in Section 4.3.1, two JPEG images are used for testing that have the same resolution of $1024 \times 768$ but have different compressed file sizes. Figure 4.13 illustrates the results decoding the images on simulation systems configured with the different NI models, as illustrated in Figure 2.6. The images were also decoded on a simulation system with ideal NoC timings and a cycle accurate simulation on an FPGA, for comparison. The bar charts' legends are explained in Section 4.2.1.

As in Section 4.3.1, the computation is the major contribution to the decoding time. Adding more cores has an exponential decrease of return, as is bore out in Figure 4.13. The **IDEAL** simulation stays approximately proportional to the simulation models that take account of the NoC timings for all numbers of cores used. This demonstrates that it is computation and not communication that is the limiting factor when the JPEG decoder is mapped to an increasing number of cores, e.g. all cores must perform the VLD step on the entire image. The contribution of the NoC to the decoding time has only marginally increased compared to the results for the 3 Core system in Section 4.3.1. The Æthereal NoC, as described in Section 2.2, is contention free. The marginal increase in delay due to the NoC is because of taking into account the contention at the memory arbitration.

This case study shows that with the addition of the NoC model, theoretical hardware platforms can be developed and simulated in software. The results of any simulation will be conservative compared to the real life implementation. This allows the platform based design approach to be carried out entirely in software, while still being able to provide performance guarantees.

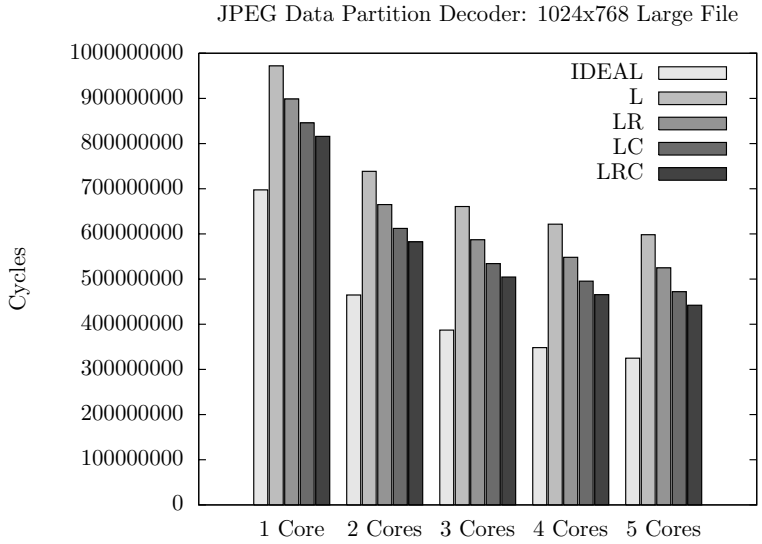



(a) Timings generated assuming Continuous Service Slots in the slot tables.

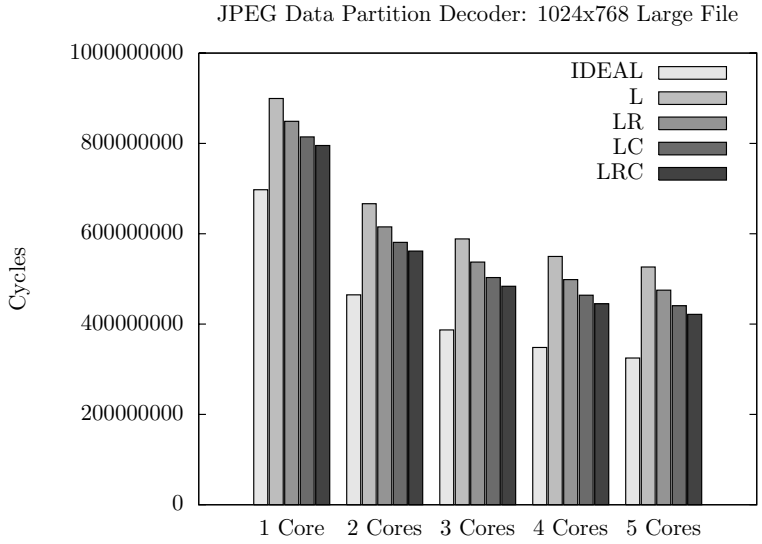



(b) Timings generated taking into account Distributed Service Slots.

Figure 4.13: Bar charts illustrating the conservative timing data for traces generated using a data parallel jpeg decoder, decoding a $1024 \times 768$ JPEG with a relatively small file size.




(c) Timings generated assuming Continuous Service Slots in the slot tables.




(d) Timings generated taking into account Distributed Service Slots.

Figure 4.13: Bar charts illustrating the conservative timing data for traces generated using a data parallel jpeg decoder, decoding a $1024 \times 768$ JPEG with a relatively large file size.

# Chapter 5

# Conclusions & Future Work

Real Time applications have temporal constraints to adhere to. In order to ensure compliance to the temporal constraints it is necessary to be able to conservatively predict the temporal behaviour of the application for its target hardware platform. It is possible to analytically calculate application-level conservative temporal bounds through the use of formal modelling techniques to model the application and hardware. Soft RT applications are often more difficult to formally model than Firm RT applications, as they may be input dependent and the programming models used less strict, while their temporal constraints are more lenient.

In this thesis, simulation is proposed as an alternative to formal modelling in order to facilitate application-level performance analysis. Using a predictable hardware platform as a starting point, an off-the-shelf MPSoC modelling and simulation framework from Silicon Hive is used to model the hardware platform. The processors are modelled using cycle accurate ISS, while the temporal behaviour of the interconnect, memory arbitration and memory access are encapsulated in the interconnect model.

In this thesis a method is contributed how Æthereal NoC connections can be independently modelled using a combination of cycle accurate and LR abstracted components. LR abstraction is applied to runtime arbitrated components in order to conservatively bound the temporal behaviour of the arbitration without having to apply an overly conservative static worst case timing. In this thesis it is shown how Latency and Rate values can be calculated for TDMA arbitration tables, using analytical methods described in [14]. It is shown that the Latency component of the LR values calculated using this method are overly conservative for TDMA arbitration tables that do not deliver service in a single continuous block of service slots. It is observed that TDMA tables can be classified as having either Continuous Service Slots (CSS) or Distributed Service Slots (DSS). In this thesis an algorithmic method to calculate LR values is contributed, that uses the observation that DSS tables are made up entirely of CSS sub-tables to apply the principles of the existent analytical LR value method to algorithmically derive tight conservative Latency values. It is also shown how models of varying degrees of abstraction can be used to model the producing NI components of the connection model using the derived LR values.

The thesis work also contributes a description of how the Æthereal NoC model can be implemented in a programming language non-specific manner.

The implementation described explains the techniques that are used to model all the independent NoC channels in a single thread, that may be necessary for some MPSoC modelling frameworks. The NoC simulation model is able to conservatively bound read and write transactions across the NoC. It is shown in this thesis that conservatively bound transactions do not guarantee that the application level is also conservatively bounded for inter-IP synchronisation. A technique is described that uses an augmented application-level communication API that incorporates measures to conservatively bound inter-IP synchronisation in simulation.

A further contribution is made in the evaluation of the simulation technique by applying it to artificial applications and a real-life application in the form of a JPEG decoder. Results are then compared with the timing results of the same applications executed on an FPGA implementation of the system. Through the use of artificial applications it is possible to bring to the surface underlying behaviour of the model. The Connection Test case study exercises a single Æthereal NoC connection by reading and writing an array from and too external memory respectively. This allows the temporal behaviour of the model for the two different transaction types to be studied. The Communication API case study results illustrate that the effect of synchronisation granularity on the applications temporal behaviour in both simulation and FPGA implementation. It is shown that increasing the synchronisation granularity decreases the runtime in both cases. Increasing the synchronisation granularity also has the effect of increasing the amortisation of the extra poll required for conservative application-level synchronisation in simulation.

The JPEG Decoder case study results provide the opportunity to evaluate the model when used to simulate a real-life application. The method used to decode a JPEG has much in common with the method to decode video frames, e.g. from a video format such as MPEG. The results show that the model can successfully conservatively bound the decoding of a JPEG image. A second JPEG Decoder case study is used to demonstrate that the NoC model parameters are modifiable enabling the exploration of NoC configurations in simulation.

In this thesis conservative simulation is proposed as an alternative to formal modelling for application-level performance analysis of an MPSoC. It is contributed how the Æthereal NoC connections can be modelled independently using a combination of cycle accurate components and LR server abstraction for runtime arbitrated components. A further contribution in this thesis is an algorithmic technique to derive tightly conservative Latency Rate values for the Æthereal TDMA arbitration tables used by the producing NIs. The description of the implementation of the model is also contributed along with the evaluation of the model through the simulation of multiple case study applications. The conclusions of which are that conservative simulation of an MPSoC is a feasible technique for conservative application-level performance analysis.

## 5.1 Future Work

The ISS for the Pearlray cores used in the hardware platform does not take memory arbitration time of the Pearlray's internal memory into account. This was circumvented in the work in this thesis by performing inter-core communication

via shared memory external to the cores. Silicon Hive processors definitions are modifiable using the TIM language. It may be possible to implement temporal accountability for Pearlray core's internal memory arbitration.

The derivation of Latency Rate values, in this thesis, is focussed on the calculation of conservative values. LR servers do not necessarily have to be conservative. If it is not necessary to provide temporal guarantees some form of average LR values could be derived for use instead. This could allow the model to provide expected timing estimations rather than just worst case timings.

Conservative buffer sizing should be possible on a per-trace basis through simulation of an application using the model. By setting the producing NI buffer on the request side to be infinitely large (or something more feasible) the maximum occupancy of the producing NI buffer on the request side during a simulation is the worst case size for the buffer, as all timings after and including the arbitration at the producing NI buffer, are worst case. In order to conservatively size the consuming NI buffer on the request side in a similar manner some form of best case LR timing could be used for the producing NI. All timings after and including the request side NI buffer are worst case, making the maximum occupancy of the consuming NI buffer during simulation the worst case buffer size.

It should be possible to provide worst case energy consumption for the NoC model on a per-trace basis. By assigning the NoC model components a watts-per-cycle value the conservative timing simulation could also generate the worst case energy consumption for the NoC. Using average LR timings, as suggested for timing analysis, would generate an energy consumption estimation.

An Embedded Systems Laboratory module, that is taught in the Technical Universities in both Eindhoven and Delft, uses the Silicon Hive SDK in combination with a predictable MPSoC hardware platform to teach students about the complexities of program mapping on MPSoCs. Timing analysis of the students efforts is carried out using the FPGA implementation. The Æthereal NoC model described in this thesis could be used to provide application-level performance analysis through simulation of the MPSoC.

# Bibliography


[1] L. Benini *et al.* Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[2] G. Bilsen *et al.* Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[3] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 1997.

[4] S. Dimitrios *et al.* Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Trans. Netw.*, 6(5):611–624, 1998.

[5] K Goossens *et al.* Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, September/October 2005.

[6] A. Hansson *et al.* Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 954–959, 2007.

[7] A. Hansson *et al.* Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–24, 2009.

[8] A. Hansson *et al.* Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques*, 2009.

[9] Matthias Krause *et al.* Combination of instruction set simulation and abstract rtos model execution for fast and accurate target software evaluation. In *CODES+ISSS'08*, pages 143–148, 2008.

[10] S. Mahadevan *et al.* Arts: a system-level framework for modeling MPSoC components and analysis of their causality. *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 480–483, Sept. 2005.

[11] A. Nieuwland *et al.* C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of the embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, October 2002.

[12] Silicon Hive. website. `http://www.siliconhive.com`.

[13] Silicon Hive. *SDK User Manual: Silicon Hive software development kit*, November 2006. Revision 1.

[14] Maarten Wiggers *et al.* Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *SCOPES*, pages 11–22, 2007.