Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Front-end for Composable Resource Sharing Using Latency-Rate Servers

Getachew Teshome Woldegebreal

CE-MS-2009-22

Abstract

In this thesis, the design and implementation of an efficient front-end for composable resource sharing is presented. With composable resource sharing, every application obtains a service that is not affected by interference from other applications that share the same resource. Since applications are shielded from interference, each one of them can be verified by simulation independently and integrated without reverification. This reduces the verification effort that would, otherwise, be tremendous.
Our solution is based of Latency-rate ($\mathcal{LR}$) servers, which are used to model service provided by a predictable resource. The front-end provides composable resource sharing when attached to a predictable resource. A series of tests have been carried out to verify that the front-end isolates applications from each other while sharing a resource.
The design has been synthesized for FPGA as well as ASIC on CMOS 90nm technology to estimate area and operating frequency.

Faculty of Electrical Engineering, Mathematics and Computer Science

Front-end for Composable Resource Sharing Using Latency-Rate Servers

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Getachew Teshome Woldegebreal
born in Motta, Ethiopia

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

Front-end for Composable Resource Sharing Using Latency-Rate Servers

by Getachew Teshome Woldegebreal

Abstract

In this thesis, the design and implementation of an efficient front-end for composable resource sharing is presented. With composable resource sharing, every application obtains a service that is not affected by interference from other applications that share the same resource. Since applications are shielded from interference, each one of them can be verified by simulation independently and integrated without reverification. This reduces the verification effort that would, otherwise, be tremendous.

Our solution is based of Latency-rate ($\mathcal{LR}$) servers, which are used to model service provided by a predictable resource. The front-end provides composable resource sharing when attached to a predictable resource. A series of tests have been carried out to verify that the front-end isolates applications from each other while sharing a resource.

The design has been synthesized for FPGA as well as ASIC 90nm technology to estimate area and operating frequency.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2009-22

Committee Members :

Advisor:	Prof. Kees Gooossens, CE, TU Delft
Advisor:	Benny Akesson, ES/ICS, TU Eindhoven
Member:	Prof. Kees Goossens, CE, TU Delft; NXP Semiconductors
Member:	Prof. A. Van Der Veen, CAS, TU Delft
Member:	Dr. Sorin Cotofana, CE, TU Delft

I dedicate this thesis to my beloved parents, my dad Ato Teshome and my mom W/ro Mitikie. They always devote themselves for the success of their children, usually giving more than what they afford.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

This thesis is the result of a project done at NXP Semiconductors, High Tech Campus in Eindhoven, under the supervision of Prof. Kees Goossens. I would like to thank Prof. Kees for offering me the opportunity to work on such an interesting project in a very conducive working environment. I had the privilege of getting his wonderful supervision regularly throughout the course of the project. It would be unfair to mention Benny Akesson just as a great supervisor. He was also like an elder brother who shows his youngest a new road everyday. I can't thank him enough.

All this wouldn't have been possible without the support that I got from Nuffic (Netherlands Organization for International Cooperation in Higher Education) to pursue my study at Delft University of Technology; I am very grateful. I am also grateful to Addis Ababa University (in Ethiopia) for the support and arrangement of my stay in the Netherlands.

Last but not least, my beloved family and friends had never been further than a phone call away. Their encouragement and advice is the single thing that kept me upright even at moments of my weakness.

Thank you God for giving me everything I needed to reach this point.

Getachew Teshome Woldegebreal
Delft, The Netherlands
August 26, 2009

1 Introduction

Time after time, embedded systems are increasing their presence in our daily life. From mobile phones to aviation, kitchen appliances to medical equipments and toy cars to space shuttles, we depend upon embedded systems to lead our daily life properly. Their application areas range from simple consumer applications to safety critical applications, in which malfunctioning may result in severe consequences, such as loss of life. To prevent mishaps from happening, devices are vigorously tested before putting them to actual use. As verification is part of the design cycle, it increases the time-to-market and introduces more expenses. Embedded systems in mission critical application domains enjoy the luxury of a huge budget to spend on testing and longer design cycle. In consumer electronics, in contrast, reducing time-to-market and design cost is determinant to the success of a product. Some trends in embedded systems design and market, however, tend to increase the complexity of system verification, thereby, raising verification effort, design cost and time-to-market.

We start this chapter by highlighting some trends in System-on-Chip (SoC) that are important to our work in Section 1.1. In section 1.2, statement of the problem is presented. Section 1.3 explains the context in which this work has been done. This introduction section is concluded with Section 1.4, where requirements from the solution are pointed out.

1.1 Trends in Embedded Systems

1.1.1 Embedded System Market

Due to the fast growth of the consumer electronics, the Embedded System market has attracted the attention of many manufacturers. A lot of them compete fiercely to manufacture devices that have good value in the market. This puts a lot of pressure on each manufacturer to be successful. The first key issue is coping with the dynamic market and being able to satisfy user demand that continues to rise day by day [15]. Equally important is a short time-to-market in releasing a product. Shorter time-to-market not only reduces design cost, but also gives the products good price in the market. Thus it is necessary for every manufacturer to reduce the design time of products.

1.1.2 Multiple functions per device

Driven by market convergence [19], between *Telecom*, *Consumer Electronics* and *Computers* , it has become a common practice to incorporate multiple functionalities into a single device. A white paper from the Economist Intelligence Unit [20], mentions iPhone as a successful converged device.

> *... game-changing iPhone, which neatly combines a mobile phone, handheld computer and a music player in one converged device.*

This practice, for one thing, adds more portability to the devices, and then it makes it possible to share some of the device components, such as processors and memories, among different applications thereby reducing the total cost.

In a device with multiple applications, the set of applications that are active changes from time to time. Each envisaged combination of applications that run simultaneously on a device is called a *use-case*. The number of use-cases grows exponentially with the number of applications included in the device. As each use-case must be verified, *the verification complexity also grows exponentially with number of applications.*

1.1.3 Real-time requirements

As new applications continue to enhance the capability of embedded systems, many of them are emerging with different real-time requirements. Based on their real-time requirements, applications are categorized into *Non Real-time* and *Real-time* applications. An application is said to be real-time if the usefulness of operations that comprise it depends not only upon its logical correctness, but also upon the time in which it is performed. In *non real-time* application, on the other hand, timing is not of concern as long as the operations are logically correct.

Real-time Applications are further categorized into *Soft Real-time* and *Hard Real-time*. In hard real-time applications, timing requirements are so strict that an operation is useless, or may even have negative consequences, if it is not completed within the deadline. Missing deadlines in hard real-time systems may lead to failure of the application and, even, damage the surrounding. One example of hard real-time applications is Fly-by-wire flight control, which uses electronic signals to measure a pilots input and to control the aircraft [7]. The flight control *must* receive the pilot's inputs and transmit actuation signals at a predetermined frequency. If it fails to meet the timing requirements, the aircraft can become instable and lead to a crash.

Soft real-time applications also have timing requirements. However, occasional deadline misses are tolerated with impact on the quality of application only. For instance, deadline misses in *Video Decoder* application lead to missing frames in display. But, the consequence is limited to degradation in application quality.

Timing requirements in real-time applications , obviously, add another dimension to system verification. Such systems require verification of timing behavior on top of the trivial functional verification and, hence, lead to higher verification effort.

1.1.4 Modular IP Design and IP Reuse

One of the emerging methodologies in a System-on-chip (SoC) design is combining pre-designed and pre-verified blocks - often called intellectual property (IP) blocks - on a chip to increase design productivity [15]. An IP block can be software or hardware component that is obtained from internal sources, or even from external vendors, in reusable form. With this design approach, an SoC is built from small blocks that have well-defined function and approach. The role of an SoC designer is, then, to integrate IP blocks on to a chip to implement a complex function. This design paradigm has benefit for IP vendors as well as users. For IP vendors, it gives the opportunity to design components that can be used by more clients so that the design cost is amortized. IP users, on the other hand, benefit from reduced design cost and short time-to-market.

Although reuse is a viable approach to tackle the unprecedented complexity of current SoCs, the associated verification problem is in some respects harder [15]. As mentioned in the *International Technology Road map for Semiconductors (ITRS)* [8], while design sizes have grown exponentially over time [16], theoretical verification complexity has been growing double exponentially [15]. If the entire SoC were to be tested at once without exploiting the structure of the building blocks, all possible verification states would have to be considered and the complexity would be tremendous. The solution is incremental verification, in which individual cores are verified once in isolation and then the integrated system is verified under the assumption that each core is correct. By doing so, the verification effort for each core is made reusable.

1.1.5 Resource Sharing

A SoC, usually, contains one or more IPs that can be used for multiple applications. In such occasions, sharing resources becomes the economical approach. Memory units and I/O devices are among resources that are often shared. Resource sharing is not, however, without problems. This implicit interaction between applications alters their temporal behaviors. In verifying real-time applications, it is necessary to conform that the interference due to resource sharing does not lead to violation of timing requirements. This, obviously, adds to verification complexity unless a mechanism is put in place to share resources between applications in an interference-free way.

1.2 Problem Statement

Each of the trends discussed in the Section 1.1 relate to the verification effort in SoC design. The market dynamics pushes manufacturers for new products and new features more often than ever. With the remaining trends tending to increase complexity of system verification, it becomes a big challenge to verify all requirements within reasonable time frame and budget. However challenging it might be, system verification is mandatory to ensure reliability of products and hence to sustain in the market.

The objective of this work is reducing system verification effort by removing implicit dependencies between applications that arise due to resource sharing. When applications do not have implicit dependence among them, they can be verified independently and integrated later without requiring re-verification.

Consider a system composed of n applications, with the i^{th} application having some measure of verification complexity (e.g., number of reachable states, number of functional coverage points) of v_i. If the integration leads to interference among the applications, it becomes necessary to consider the cross product [15] of all possible verification states.

This results in a verification complexity of

$$\prod_{i=1}^{n} v_i \tag{1.1}$$

which is exponential in n. The goal of this work is a resource sharing mechanism that shields applications from interference from other applications so that individual applications are verified once in isolation and only system-level verification is required after integration. The system-level verification is carried out under the assumption that each application behaves correctly as was verified independently. This leads to a lower overall verification complexity which amounts to

$$v_{sys} + \sum_{i=1}^{n} v_i \tag{1.2}$$

where v_{sys} represents the system-level verification effort. As can be seen in Equation (1.2), the total verification effort grows only linearly as the number of applications is increased. Another benefit of this design approach is incremental verification. The verification process can be started earlier with the available applications without having to wait for the whole application set.

1.3 Context

The context of this thesis is a SoC with multiple IP blocks interconnected with an on-chip interconnect. Some of the IPs are processing elements, such as a DSP or general purpose processor while others are resources; such as memory or VGA display that are to be used by applications. Communication between two IPs is carried out via requests and responses. One of the IPs (*the master*) generates request and the other IP (*the slave*) responds by sending back responses. A request contains the command component of a transaction and the data to be written (*payload data*) in case the request is a write operation. Critical elements such as *request address*, *request size* and *request type* are

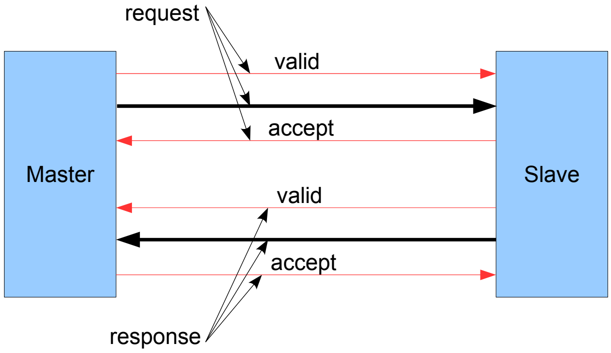


Figure 1.1: Components of a Transaction and Flow-Control

included in the command component. The response sent by the slave can be *read data* for read requests or information about the completion of a write operation for write requests. The components of a transaction and the direction of signals involved are shown in Figure 1.1. When a master sends a request, it informs the slave about the validity of the request by setting the *valid* flag high. The slave then acknowledges by setting the *accept* flag high if it is able to accept the request or indicates otherwise if it is unable. Similarly, when the slave returns responses, it sets the *valid* signal in the response path high. The master, then, indicates whether or not it is able to accept the response. These handshakes between two communicating parties represent the flow-control mechanism. The DTL protocol, which uses such flow-control mechanism and used in our implementation is discussed in detail in Section 5.1.1.

When there are multiple links between the master and the slave, the communication involves link-level flow-control at each hop. Figure 1.2 illustrates the link-level flow-control involved as a master and a slave communicate with multiple hops in between.

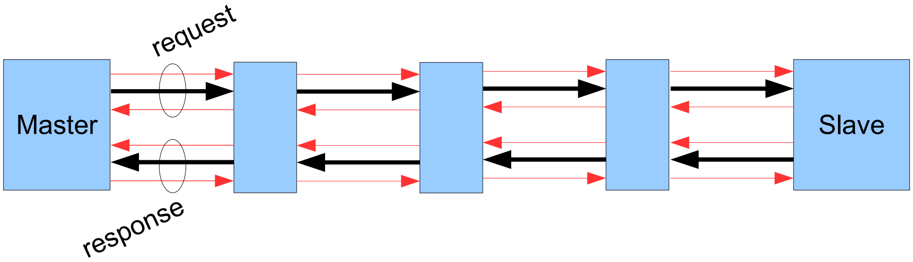


Figure 1.2: Link-level Flow-Control

As the number of IP blocks in a single SoC increases, it becomes difficult, if not impossible, to satisfy the communication needs using ad-hoc means. Interconnect infrastructures such as busses, switches and network on chips (NoCs) provide an elegant communication platform in a SoC [3]. Communicating IPs use the shared infrastructure to send request and responses. A virtual path in the interconnect that is used for communication between a master IP and a slave IP is referred to as a *connection*.

Figure 1.3 illustrates an example SoC with three processing IPS (ARM, DSP_0 and, DSP_1) and two resources ($Memory$ and VGA). The example also demonstrates that the memory is used by the three processing elements and hence represents a shared resource. When there is a shared resource in the system, a resource sharing front-end is required to arbitrate/ schedule access to the resource and that is the central issue in this thesis.

One or more applications, such as *JPEG decoder* or *Audio Filter* run on the system to bring about the required functionality. When an application is loaded, the tasks that comprise it, such as *color conversion* in JPEG decoder are mapped on to hardware IPs in the system.

Mapping of two applications, a *JPEG Decoder Application* and an *Audio Filter Application*, on to the system in Figure 1.3 is presented in Figure 1.4 as an example. The JPEG decoder application is comprised of three tasks [5] - *Variable Length DecodingVLD*, *Inverse Discrete Cosine Transform (IDCT)* and *Color Conversion (CC)*. Similarly, the Audio Decoder application has three tasks - *Analog-to-Digital Converter (ADC) Filter* and Digital-to-Analog Converter (DAC). Communication between tasks of an application

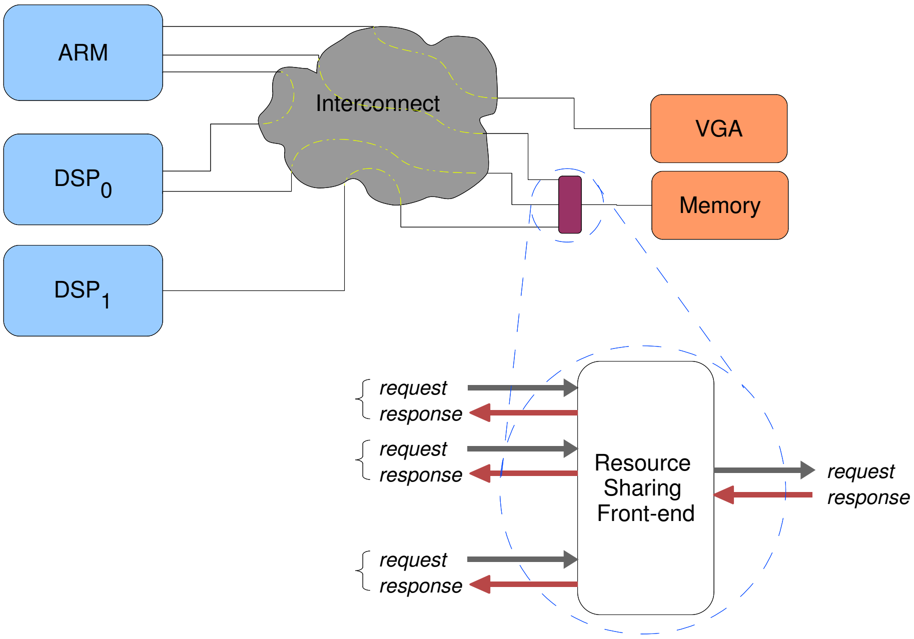


Figure 1.3: A System-on-Chip

is carried out through connections between IP blocks.

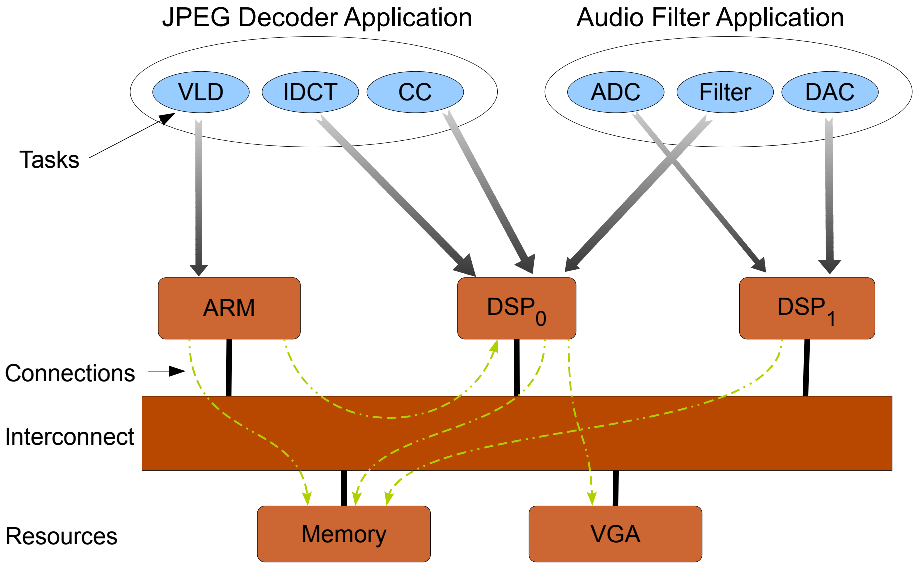


Figure 1.4: Mapping of an Application on to an SOC

A connection that terminates at a target port of a resource is refered to as a *requestors*. Each resource has a set of requestors, which correspond to a request path from different IPs. For instance, the Memory port in Figure 1.4 has three requestors that emanate from ARM, DSP_0 and DSP_1. The focus of this work is the elimination of implicit dependence between applications caused by interference while sharing a resource.

1.4 Requirements

The major requirement in the solution is isolating applications so that they can be verified independently. In addition, the design has to feature state-of-the-art design requirements, such as programmability and modular design. These requirements are briefly discussed next one by one.

1.4.1 Composability

A system is said to be composable if both the functional and temporal behaviors of every application are independent of how other applications behave. In a composable system dependence between applications is eliminated both in time and value domains [2]. This isolation is important because it enables incremental design and reduces complexity of system integration and verification [9]. Composability simplifies the verification process for the following three reasons [2].

- As applications are independent of each other, they can be designed, tested and verified in isolation. This reduces system simulation time and makes the verification process linear and non-circular.

- The verification process can be done incrementally and, can, hence be started even before the complete application set is ready.

- Moreover, composability contributes to IP protection. Since verification of IPs can be done in isolation, before the final integration, vendors do not have to share their IP.

In a situation where a resource is shared, availability of the resource for one application may depend on the behavior of other applications that use the same resource. If an entire system is to be composable, resource sharing also has to be composable.

1.4.2 Programmability

When applications are started and stopped at run time, they create different use-cases. Whenever there is use-case transition, requirements from applications are added to or removed from the system. To effectively satisfy such changing demands, the system has to be flexible. Flexibility is achieved by having some programmable parameters in the system that can be changed at run time.

1.4.3 Modular Design

Instead of realizing a solution as a monolith, building it out of smaller functional modules has benefits. First of all, the modules can be designed and modified independently without affecting other modules. Hence parts of the system can be changed while keeping the rest intact. Another benefit is reusability, which reduces design time and time-to-market. By creating individual blocks with specific functions, reusability of the design is maximized. To facilitate reusability modules should be created with standard interfaces to connect with other modules.

1.5 Contributions

The result of this thesis is a hardware block *(resource sharing front-end)* that eliminates interference between applications as they share a resource. The following mechanisms have been implemented to realize this function.

1. Mechanism to chop transactions into fixed size (Atomizer).
2. Mechanism to delay transactions and release them at a predefined time (Delay Block).
3. Hardware Implementation of the Credit-Controlled Static-Priority (CCSP) Arbiter [2].
4. Representation of time with circular cycle counter and associated mechanism to compare time stamps.
5. Synthesis and testing of the resulting hardware for FPGA and ASIC (only Synthesis).

The hardware overhead and the operating frequency of the resulting hardware has been assessed, too.

1.6 Organization of The Thesis

The rest of the thesis is organized into nine chapters. We start by looking at related works in Chapter 2. Then, in Chapter 3, we propose a solution to address the problem stated. Our approach towards the solution and the proposed system architecture are outlined there. The design of all architectural elements based on our proposed solution is presented in Chapter 4 followed by implementation details in chapter 5. Experiments and test results are presented in Chapter 6 followed by synthesis results in Chapter 7. Analysis on hardware cost and operating frequency of our design is made there. The thesis ends with conclusions in Chapter 8 and direction to future work in Chapter 9.

2 Related Works

A lot of work has been done with the aim of reducing system verification effort while building complex systems. In [10], worst-case resource allocation is enforced to each application during verification in order to account for maximal interference from others. If a given user-level performance goal is satisfied with the enforcement, then the system is considered to be able to perform equally well, or even better, during actual use with the enforcement removed. This approach can guarantee better performance, during deployment, only if the applications considered execute on the platform in a *performance monotonic* manner i.e. if applications are known to perform better with more service. However, in many cases, earlier service availability or higher service amount than the guaranteed value does not necessarily result in better system performance [4]. For instance, timing anomalies are observed in multi-processor systems with out-of-order processing [11]. Cases have been demonstrated in which a system performs better with cache miss than with cache hit. Hence a system verified with enforcement is not guaranteed to meet requirements during deployment, when service may be available earlier and/or with higher amount. Thus system verification requires system level simulation which can take tremendous effort when many applications are integrated to the system.

Composability has been proposed in many works to reduce system verification cost. In the automotive industry, for instance, the traditional way to achieve composability is by not sharing resources between applications. With this approach, systems are designed with *federated architectures*, where every function is served by a dedicated *Electronic Control Unit (ECU)* [12]. As the units are not shared between applications, obviously, there is no interference and hence the resulting system is composable. The cost of systems without resource sharing, however, is prohibitively high for consumer electronics. To circumvent the cost problem, *integrated architectures* are in use in the consumer electronics domain, and even in the automotive industry [14]. To ensure composability in integrated architectures, where resources are shared among applications, the resulting interference has to be eliminated.

Despite difference in approach, the works in [9, 6, 2] propose composable system to reduce verification cost in an integrated system. The Time-Triggered Architecture in [9] proposes a two-phased design methodology to achieve composability. During architecture design phase, components are specified in value and temporal domain. Communication at the interfaces of the shared resource is statically scheduled. After validating the constraints, the resulting components are used to build a composable system. As the individual components are pre-validated, the resulting system incurs less verification effort. This approach requires global notion of time and is limited to applications that can be scheduled statically.

The CoMPSoC platform proposed in [6] employs local Time Division Multiplexing (TDM) to bring about composability in a multi-processor SoC. With TDM, every application is guaranteed access to resources during allocated slots of time and hence is shielded from interference. With this approach, however, latency and bandwidth are coupled. As a result, applications with low bandwidth requirement cannot be guaranteed with low latency without overallocation of bandwidth. Many resources such as SDRAM are scarce and have to be shared efficiently.

The composable resource sharing presented in [2], which is implemented in this work, is based on latency-rate ($\mathcal{LR}$) servers. By using different arbiters in the class of ($\mathcal{LR}$) servers, it makes it possible to provide greater service differentiation than in TDM [2]. For instance, an arbiter in this class that decouples latency and bandwidth can be chosen to efficiently satisfy tight latency requirements. As this approach is the base for the work in this thesis, it will be discussed in more depth in Section 3.

Proposed Solution 3

3.1 Approach

When requestors share a resource, interference arises between them due to two major reasons - *Arbitration for service* and *effect on resource state.*

- **Arbitration:** When multiple requestors seek service from the same resource, arbitration is required to sequentialize the requesters. The result of the arbitration depends on the set of requests waiting to be served and the scheduling policy. With static-priority scheduling, for instance, the lowest priority requestor is deprived of service as long as there are other requestors with higher priority.
- **State of resource:** Once a request has been admitted to the shared resource, the actual service that it gets may depend on the state of the resource at the moment. We can consider a case with a Synchronous Dynamic RAM (SDRAM) to demonstrate such a dependence. Depending on the request served previously, the SDRAM may have to go through read/write switch, write/read switch or no switching at all. It may even require to refresh before serving the request at hand. That means the waiting time for a request depends on the nature of requests served previously. As there is a possibility for the previous request to be from another requestor, this situation entails interference between requestors.

The effect of interference is reflected on the timing of *responses* and *flow-control information* sent to each requestor. Figure 3.1 illustrates these paths of interference for two requestors sharing a resource. First of all, the time at which a response becomes ready depends on the time when the corresponding request is admitted to the resource. Then, it matters how fast the request is processed. The time of admission is determined by

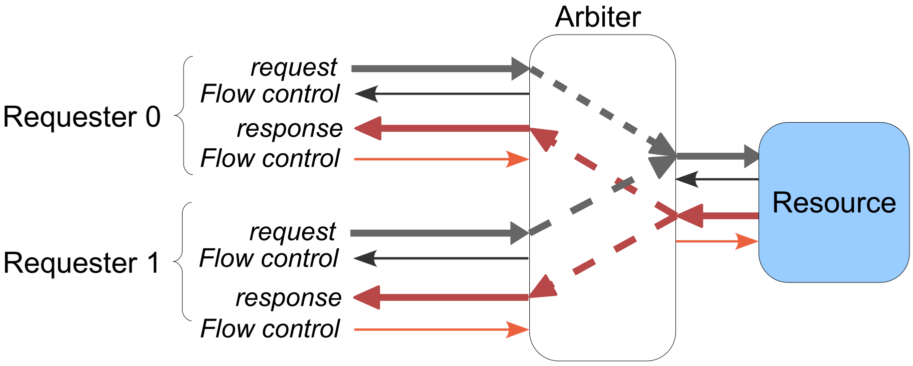


Figure 3.1: Paths of Interference During Resource Sharing

the scheduling decision, which is made considering requests from other requestors, as well. As a result, access to the resource is affected by the behavior of others. The state of the resource, which, among others, is affected by the nature of requests served previously, determines the time at which responses are made available. The other path of interference is through the flow-control mechanism. As shown in Figure 1.1, flow-control signals represent critical information about acceptance of requests and availability of resources. Requestors change their behavior in accordance with flow-control information about state of the system.

In summary, resource sharing creates implicit dependence between requestors and the resulting interference is reflected on the timing of responses and flow-control signals. If resource sharing is to be composable, the amount of interference that each requestor faces should not be affected by the behavior of other requestors.

We propose a resource sharing front-end based on the approach in [2] to eliminate fluctuation in the timing of the two events - release of responses and generation of flow-control information. Regardless of the actual time when responses are made available, they are always delayed and released at a later time that emulates maximum interference from other requestors, i.e., whether other applications are present or not and regardless of their actual behavior, our solution always emulates their worst-case interference. Similarly, flow-control signals are sent at a time that reflects maximum interference.

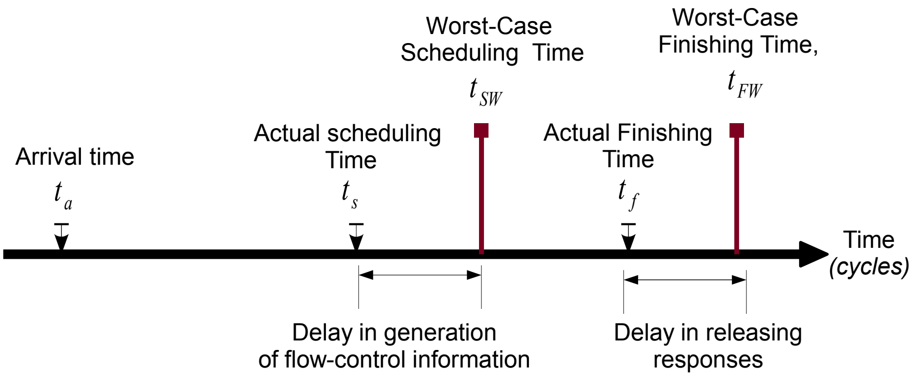


Figure 3.2: Timing of Events in the Resource Sharing Front-end

Figure 3.2 shows *arrival time(t_a)*, *actual scheduling time(t_s)*, *worst-case scheduling time(t_{SW})*, *actual finishing time(t_f)* and *worst-case finishing time(t_{FW})* for a request. The worst-case values (t_{SW} and t_{FW}) are computed considering the worst-case interference from all other requestors. From the figure, it can be seen that the request is scheduled before the worst-case scheduling time (t_{SW}) and the responses are made available at t_f, which is earlier than the worst-case finishing time (t_{FW}). However, in order to absorb any fluctuation caused by change in behavior of other requestors, generation of flow-control signals and release of responses is delayed until the *worst-case scheduling time(t_{SW})* and the *worst-case finishing time(t_{FW})*, respectively. The interval between (t_s) and (t_{SW}) represents delay in generation of flow-control information introduced by the resource sharing front-end to absorb fluctuation caused by changing behavior of other requestors. Similarly, the interval between (t_f) and (t_{FW}) represents the delay in releasing responses that is introduced by the front-end.

Since the generation of flow-control information and release of responses is always delayed until a predefined time, each requestor obtains service that does not fluctuate with the actual behavior of other requestors. This makes the system composable on the level of requestors, which is sufficient to be composable on the level of applications [2]. With this approach, requestors are prohibited from using any slack (unused capacity) that results from change in behavior of other requestors. It seems like waste not to utilize slack, but being able to shield applications from interference while providing a guaranteed service (both in time and value domains) pays off by reducing system verification cost.

Predictable Service

Our approach to bring about composability requires the resource to be predictable. A resource is said to be predictable when the service that it provides can be bounded by a known finite value. Arbiters in the class of latency rate ($\mathcal{LR}$) servers [17] are used to give guarantees on the service provided to each requestor.

With the $\mathcal{LR}$ model, service provided to each requestor is expressed in terms of two values - *service latency* and *service rate*. Service latency refers to the amount of time that a request has to wait in front of the resource before it is scheduled for service. Even though the actual time of scheduling for a request fluctuates depending on the availability of requests from other requestors, the state of the resource and the scheduling policy, $\mathcal{LR}$ servers provide an upper bound on this waiting time, which is *Maximum Service Latency* (Θ). The service rate, on the other hand, stands for the rate, such as the throughput of a memory, at which a request is served after admission to the resource. Once more, the actual service rate provided to a requestor can fluctuate depending on the momentary state of the resource. However, with $\mathcal{LR}$ servers, the amount is lower bounded with the *Allocated Service Rate* (ρ') which is the minimum service rate that is always reserved for a requestor [17].

The service curve in Figure 3.3 shows requested service along with provided service and composable service. As can be seen from the curves, requests are served at an earlier time that the worst-case scheduling time and at a rate higher than the allocated service rate. But the composable service, which is represented by the bounds (Θ) and (ρ'), is what every requestor obtains in our approach.

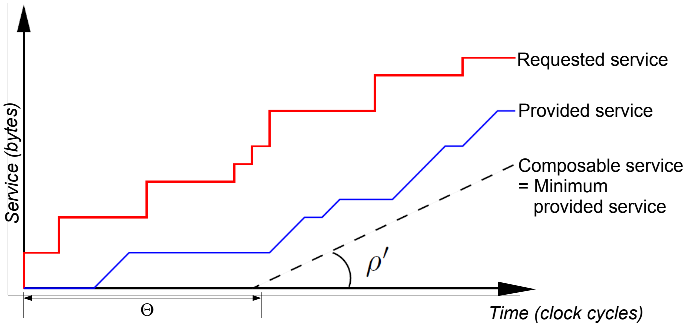


Figure 3.3: Service Curve with latency-rate ($\mathcal{LR}$) model

As mentioned above, predictability of the resource is a pre-requisite, in our approach, to achieve composability. Given a predictable resource, there are additional requirements in order to ensure service guarantees. Either requests should have a known maximum size or preemption should be allowed after a certain time. Restricting the size of requests limits robustness of the solution. Such a restriction has implication on the domain of applications that can be supported. The later approach, on the other hand, either limits the solution to preemptive schedulers such as TDM or complicates the arbiter. Our solution employs a third way that allows usage of any arbiter in the class of $\mathcal{LR}$ servers while maintaining robustness [2]. Requests can be of varying size but they are split into smaller pieces before they are scheduled.

3.2 Architecture

The front-end in our design, which is to be placed in front of the shared resource, is presented in Figure 3.4. It comprises four major functional blocks - *Arbiter, Request Bus,*

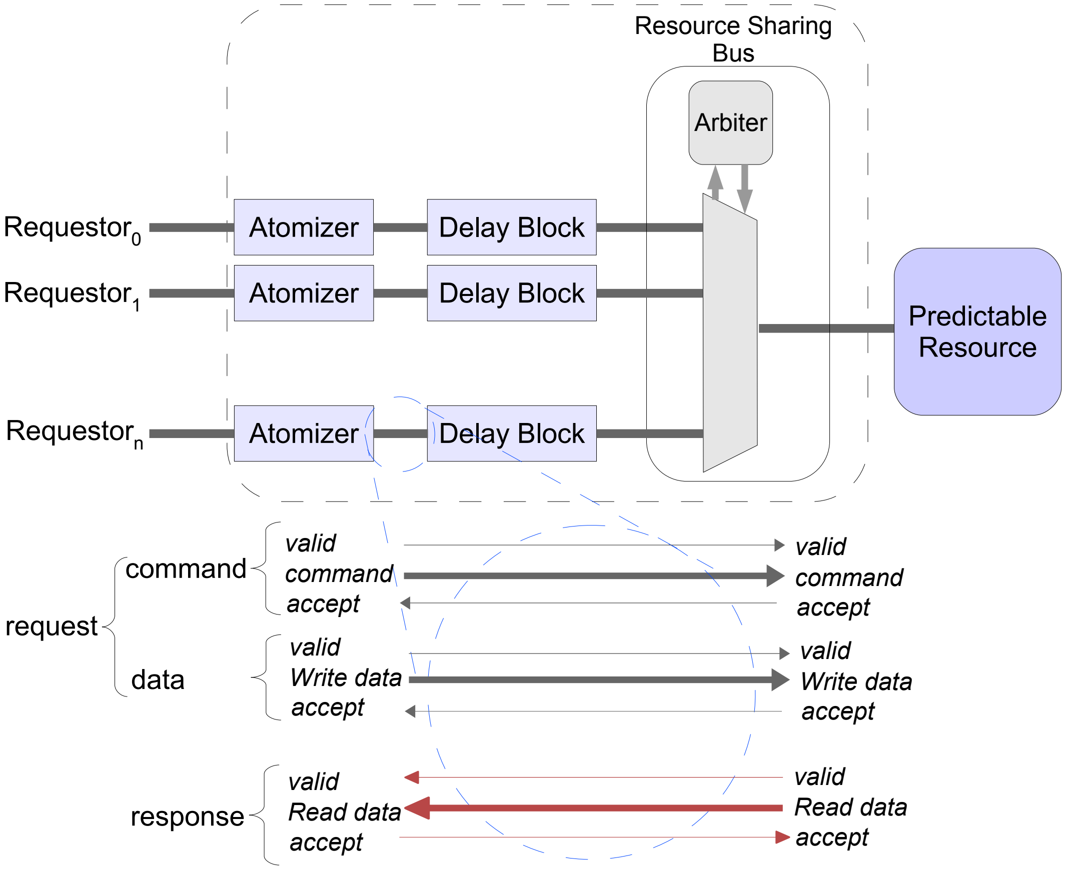


Figure 3.4: The Proposed Resource Sharing Front-end.

Delay Block and *Atomizer*. The link between blocks is composed of *request, response* and *flow-control* components. One of the links is shown in detail to make the request and response components visible. Requests contain command component and data to be written and responses contain the data read. In each direction a pair of *valid* and *accept* signals is used for flow-control.

As mentioned in Section 3.1, composability is achieved, in our solution, by delaying responses and flow-control signals to emulate worst-case interference at all times. Therefore, a hardware block, called *Delay Block* is dedicated to each requestor for the purpose. Another important unit of the front-end is the Atomizer. It is responsible for chopping requests into fixed size so that a variety of arbiters in the class of $\mathcal{LR}$ servers can be used without imposing restriction on the size of request.

In the following chapters, the design (Chapter 4) and implementation (Chapter 5) of the entire frontend are presented.

Design 4

In this chapter, the design of the entire front-end is presented. The design of the Atomizer is presented in Section 4.1 and that of the Delay Block in Section 4.2. The design of the arbiter and the Resource Sharing Bus, which contains the arbiter, is presented in Sections 4.3 and 4.4, respectively. Finally, the configuration mechanism for the front-end is presented in Section 4.5.

4.1 Atomizer

The function of the Atomizer is to homogenize requests before they enter the rest of the front-end. It chops requests into smaller pieces called *atoms.* The Atomizer has the size of atoms as a generic parameter to be chosen based on the resource. It is determined by the access granularity of the shared resource. For instance, with a 32-bit static RAM (SRAM), the size of an atom is chosen to be a word (4 bytes).

Chopping requests brings additional benefits to the design. Since all requests are made to have similar size as they leave the atomizer, subsequent units of the front-end (*Delay Block, Resource Sharing Bus* and *Arbiter*) can be simpler and more efficient. The major operations in the Atomizer are discussed in Section 4.1.1, and then the architecture is presented in Section 4.1.2.

4.1.1 Functions of the Atomizer

Chopping Requests:

This process is responsible for chopping requests and feeding them to the Delay Block. Whenever the Atomizer receives a valid request and the Delay Block is ready to accept requests, this process starts chopping incoming requests. The chopped requests are, then, sent to the Delay Block one after the other. Requests contain command component and data to be written in case of a write request. The command component holds crucial information about size and address of the request. Hence, chopping a request involves attaching new request size and appropriate start address to the command component of each atom. The first atom has its start address the same as that of the original request. The start address of subsequent atoms is, then, offset from that of their predecessor by an amount proportional to the *Chopped Request Size.* In case of a write request, a portion of the payload data is sent with each chopped request. Figure 4.1 illustrates a request as it enters the Atomizer and the corresponding atoms produced after chopping.

If the request being chopped is expected to produce response, the size is recorded in *Request Size Buffer* to be used later in the process of merging responses. For instance, a read request is turned into a series of atomic read requests and the atomizer expects

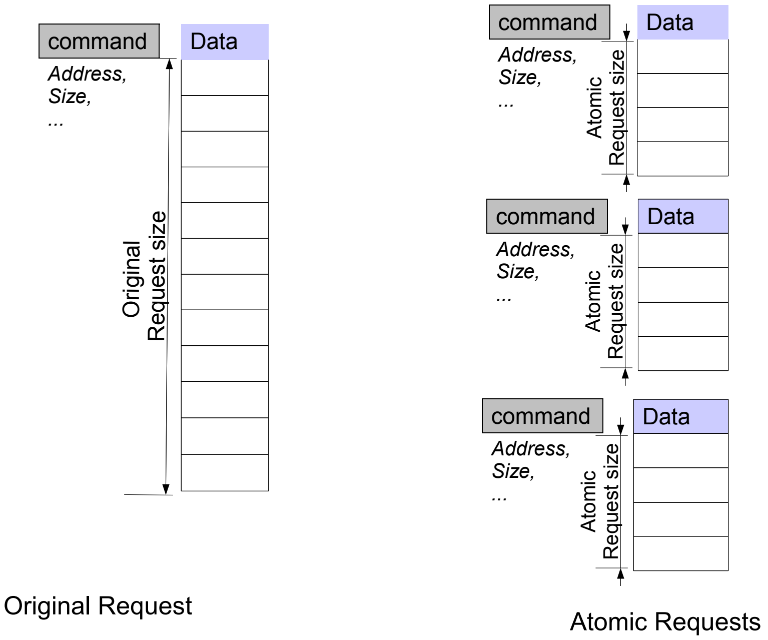


Figure 4.1: Requests Chopped by the Atomizer

responses corresponding to each atom to arrive one after the other. Hence, the request size recorded here is required to count responses and identify the ones that belong to each request.

Merging Response:

Since requests delivered to the resource are chopped, the resulting responses are also chopped alike. The requestors, however, expect responses in accordance with the original requests sent. Hence, responses are merged before leaving the front-end. This process of the Atomizer is responsible for merging responses so that they match the size of the original request. The original request sizes recorded in *Request Size Buffer* are used to count and merge responses that belong to the same request. Merging read responses, for instance, is equivalent to removing markers (indicators of the last response) from the individual pieces and generating one for the whole group of responses that belong to the same request. Figure 4.2 demonstrates an example in which responses of size 4 are merged in to responses of size 8, 12, 1, ..., according do request sizes recorded by the chopping process.

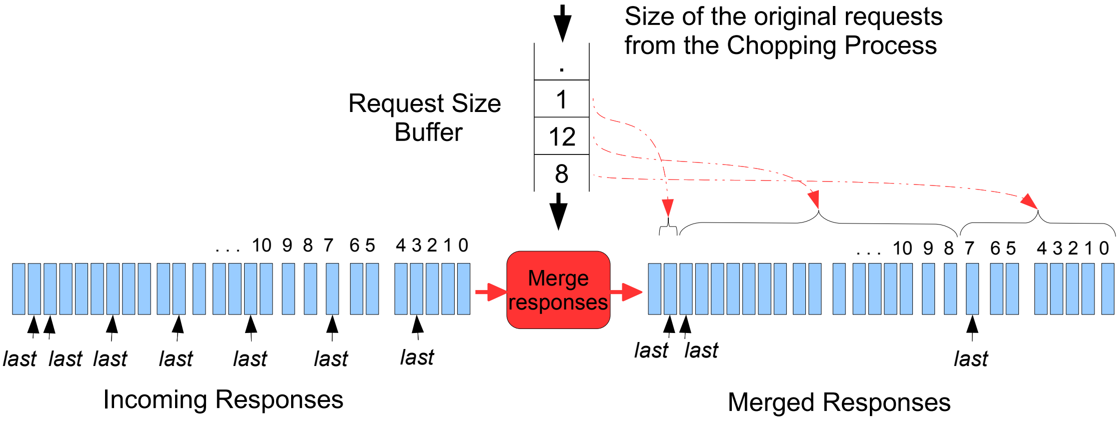


Figure 4.2: Merging of Responses in the Atomizer

4.1.2 Architecture

As shown in Figure 4.3, the Atomizer comprises two major processes - one that chops requests and another that merges responses. FIFOs are used for buffering requests to be chopped.

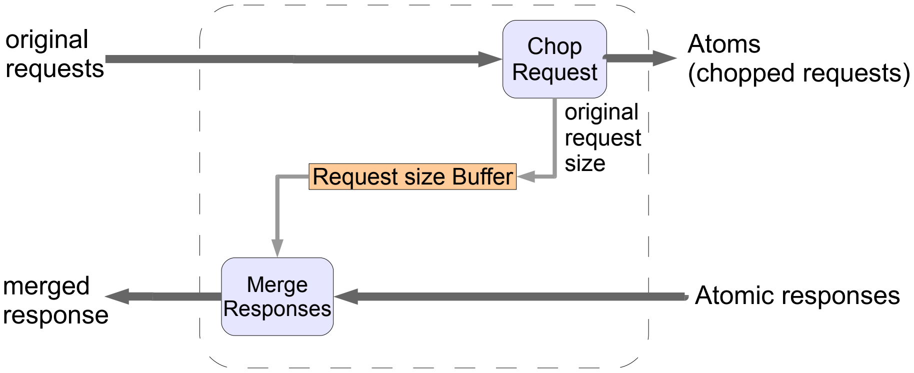


Figure 4.3: Architecture of the Atomizer

4.2 Delay Block

The Delay Block is the core unit of our approach to bring about composability. It absorbs fluctuations in the service provided to each requestor (*latency* and *bandwidth*). As pointed out in Section 3.1, interference between requestors is reflected on the timing of the following two events:

1. Time at which flow-control signals are sent when requests are accepted by the resource.

2. Time, after acceptance of a request, at which the corresponding responses are sent back to the requestor.

The time at which a request is accepted by the resource fluctuates depending on activity of other requestors and the state of the resource. This determines generation of flow-control signals. The time when responses become available depends not only on the time of scheduling but also on the actual service rate offered by the resource.

To hide the fluctuation in timing of the two events, the Delay Block always delays responses until the worst-case finishing time of requests, which reflects worst-case interference from other requestors. Similarly, flow-control information about acceptance of requests is generated based on the worst-case scheduling time. Figure 4.4 illustrates the delay processes along the response and flow-control paths. Computation of time stamps that are used in the timing of these events is presented in detail in Section 4.2.2.

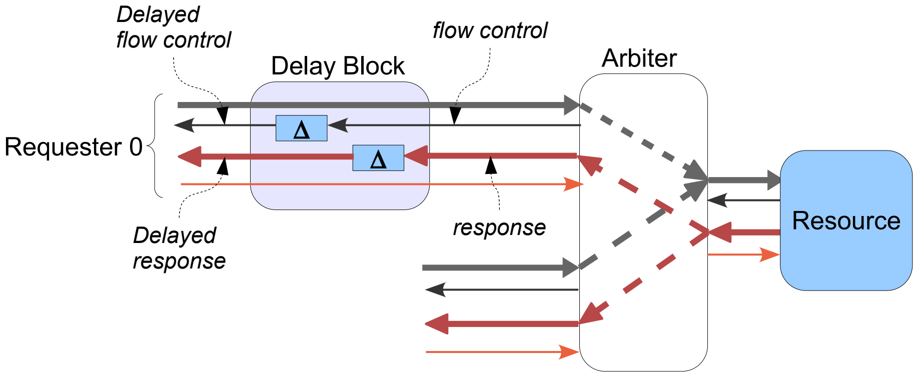


Figure 4.4: Delay Processes in the Response and flow-control Paths.

In summary, the Delay Block ensures that the scheduling time of a request and release time of the corresponding response are not affected by interference from other requestors, which is required in composable resource sharing.

4.2.1 Parameters

Before describing the operation of the Delay Block, it is important to define two crucial parameters used in the timing of events - *Service Latency* (Θ) and *Completion Latency* (λ). The value of these parameters is determined by the service guarantees provided by the $\mathcal{LR}$ servers.

1. **Service Latency (Θ):** This represents the worst-case amount of time that a request waits in front of the resource before it is scheduled for service [2]. The bound depends on the priority that the source requestor has at the shared resource.

2. **Completion Latency (λ):** This the worst-case amount of time that the resource needs to serve a unit-sized request. The value depends on the service rate allocated to each requestor, viz. $\lambda = 1/\rho'$ [2].

4.2.2 Timing of Events

In this section, the timing of some interesting events in the front-end is discussed. The time line in Figure 4.5 illustrates the most outstanding points in time as a request passes through the front-end to get service from a predictable resource.

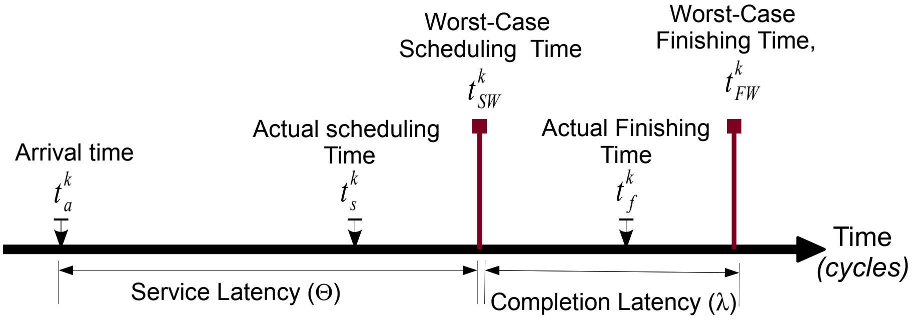


Figure 4.5: Important Events in serving a request

- **Arrival Time (t^k_a):** is the time when the k^{th} request has, fully, arrived in the Delay Block and can be scheduled. This requires two conditions to be fulfilled.

  - The request, along with the associated payload data, if any, should be buffered in the Delay Block.
  - The Delay Block should have enough space for responses, if any, corresponding to the request at hand.

  When these conditions are fulfilled, the time stamp is recorded as *Arrival Time* for the request at hand.

- **Actual Scheduling Time (t^k_s):** Once a request has fully arrived in front of the resource, the time when it is admitted depends on the actual state of the resource and , possibly, arbitration among other contending requests. This time stamp represents the actual time when a request is accepted by the resource for service.

- **Worst-case Scheduling Time (t^k_{SW}):** This refers to the worst-case time when the k^{th} request is accepted by the resource. The actual scheduling time depends on state of the resource and availability of other requests. This time stamp, however, is computed based on Θ and the arrival time. To bring about composability, this worst-case time is recorded as scheduling time for requests and is used for timing of flow-control information. The value of this time stamp is computed according to Equation (4.1) in Section 4.2.3. Note that it depends on the worst-case, and not actual, behavior of other requestors.

- **Actual Finishing Time (t_f^k):** This is the time at which the resource finishes serving the k^{th} request and returns the corresponding responses, if any. The finishing time depends on actual scheduling time of a request and the actual service rate. Hence, the actual behavior of other requestors and the state of the resource affects its value.
- **Worst-case Finishing Time (t_{FW}^k):** This time stamp represents worst-case finishing time for a request. Actual finishing time of a request depends on the actual time when it has been admitted to the resource and the actual rate at which it is served. To absorb fluctuations in actual values, this finishing time is computed based on the worst-case scheduling time and the allocated service rate. Despite the actual finishing time of a request, responses are, always, released at this worst-case finishing time. Hence, this time stamp can also be referred to as *Response Release Time*. The time stamp is computed according to Equation (4.2) in Section 4.2.3.

4.2.3 Computation of Time Stamps

The arrival of a request is validated when it has fully arrived in the Delay Block and space has been reserved in the response buffer for the corresponding responses, if any. This is the time that is taken as arrival time (t_a) for the request. (The reason behind reserving space for responses upon the arrival of requests is explained in Section 3.2.) The worst-case scheduling time (t_{SW}) and worst-case finishing time (t_{FW}) are, then, computed based on this arrival time. The worst-case scheduling time for a request is either *Service Latency (*Θ*)* cycles after its arrival time or at the worst-case finishing time of its predecessor, whichever is later [2].

$$t_{SW}^k = MAX(t_a^k + \Theta, t_{FW}^{(k-1)}) \tag{4.1}$$

The second term in the MAX equation (Equation (4.1)) is the result of self-interference because the scheduling time, in this case, is determined by the worst-case finishing time of a previous request from the same requestor.

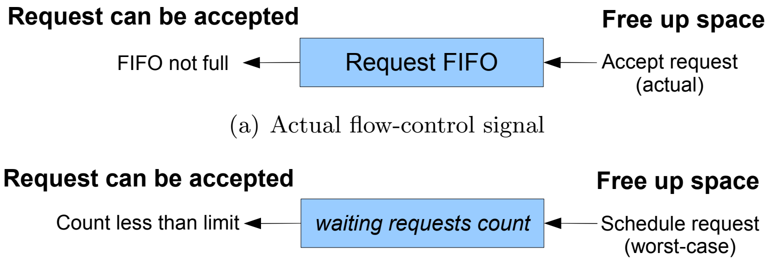
The worst-case finishing time for responses is always computed as λ cycles after the worst-case scheduling time of the corresponding request [2].

$$t_{FW}^k = t_{SW}^k + \lambda \tag{4.2}$$

4.2.4 Flow-Control

In addition to its effect on the time when responses become available, interference between requestors leads to fluctuation in the timing of flow-control signals. The flow-control signals refer to acceptance of requests and responses into the Delay Block. A new request can be admitted into the Delay Block if there is space in the *request buffer*. Space is freed up in the *request buffer* when a request is accepted by the resource.

Therefore, the flow-control signal, which tells availability of space in the *request buffer* depends on the actual scheduling process and hence on other requestors. To eliminate such interference, flow-control signals, in our approach, are generated based on worst-case scheduling. Figure 4.6 compares generation of the actual flow-control signal and that of the composable flow-control signal.



(a) Actual flow-control signal

(b) Composable flow-control signal

Figure 4.6: Generation of flow-control signal (a),(b)

The front-end is also responsible for preventing a misbehaving requestor from stalling the resource, so that it does not affect the service provided to the remaining requestors. The resource is stalled when a requestor sends many requests to the resource and then does not accept the responses. To prevent this, the front-end performs validation of requests before presenting them to the resource. A request is validated when it has fully arrived in the Delay Block, including associated payload, and if there is guarantee that responses will be accepted when they are made available by the resource. The guarantee is made by reserving space, in the response buffer, for potential response, before sending the request to the resource. If there is no space in the response buffer, the request is not validated and forwarded to the resource until a previous response leaves the Delay Block and space is freed up. Since each request is validated after reserving space for its responses, it is always guaranteed that responses are always accepted to the Delay Block as soon as they are made available by the resource.

4.2.5 Architecture

The architecture in Figure 4.7 shows major processes and buffers that comprise the Delay Block. The Delay Block has four concurrent processes that are responsible for *receiving requests, sending requests, receiving responses* and *sending responses* to and from connected blocks. It has additional processes that manage timing of events and record relevant time stamps. One of the processes, *Compute Time Stamps*, is responsible for the computation of worst-case scheduling time and worst-case finishing time of requests based on the service bounds. Another process generates flow-control information based on worst-case scheduling of requests. The *Schedule Request* process, shown in Figure 4.7, takes care of the scheduling event that takes place at the worst-case scheduling time of requests. Similarly, the *Release Response* process handles releasing of responses at the worst-case finishing time.

The Delay Block contains four buffers. The *Request Buffer*, as the name indicates, holds requests in the Delay Block until they are scheduled and sent to the resource.

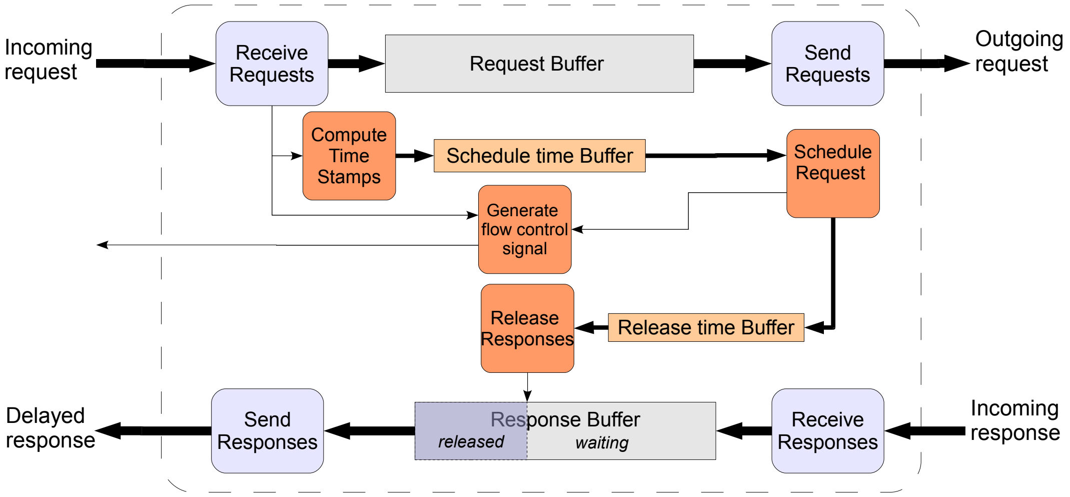


Figure 4.7: Architecture of the Delay Block

The *Response Buffer*, on the other hand, holds responses until their release time is due and they leave the Delay Block. The remaining two buffers - *Scheduling Time Buffer* and *Release Time Buffer* hold time stamps to be used for the timing of Scheduling and Releasing events, respectively. Each of the units and their implementation are detailed in Section 5.2.2.

4.3 Arbiter

In the proposed resource sharing front-end, any arbiter in the class of $\mathcal{LR}$ servers can be used. One such arbiter is the Credit-Controlled Static-Priority Arbiter (CCSP) [1]. One difference is that the arbiter is now the resource sharing front-end and hence can be used with any predictable resource. A few changes have also been made in the implementation. This will be explained while discuss the implementation in Section 5.2.3. The design of the CCSP arbiter has been revised with an approach different from the one used in a previous implementation [18]. , here, so that it fits well in the resouce sharing bus. The CCSP arbiter combines rate regulation and priorities to decouple guarantee on latency and rate. This is required, because some applications are latency sensitive and others are latency tolerant. By decoupling the two requirements, the CCSP arbiter allows providing low-latency service to low-bandwidth requests without overallocation. The parameters, mechanism and architecture of CCSP are presented in Sections 4.3.1, 4.3.2 and 4.3.3, respectively.

4.3.1 CCSP Parameters

The arbitration process in CCSP relies on three crucial parameters. These are the *Allocated Service Rate (ρ')* required for rate regulation, *Priority* for the static-priority scheduling and *Allocated Burstiness (σ')* to control burst of service.

- **Priority (p):** of a requestor, as the name implies, indicates the priority given to a requestor when it is in contention with other requestors for a resource. Every requestor is given a unique priority value.

- **Allocated Service Rate (ρ'):** represents the amount of service that a requestor is entitled for every service cycle. Every service cycle, its budget is upgraded by this amount. The allocated service rate is expressed as fraction of the resource capacity allocated to a requestor. Hence, it always has value between 0 and 1.

- **Allocated Burstiness (σ'):** represents the maximum burst of service that each requestor can get, i.e. the maximum number of times that a requestor can be served in succession without having to upgrade its budget. The initial budget that each requestor obtains at start up is based on this parameter.

4.3.2 Mechanism

CCSP maintains a budget per requestor in order to regulate service provided to each one. This protects low priority requestors from starvation. Because of the regulation, requestors are always guaranteed that they can get their allocated service amount. Every service cycle, all requestors are entitled for service that amounts to their allocated rate. This is reflected by an upgrade to their budget value. Each requestor, then, pays from its budget for every service it gets i.e. whenever it is scheduled. Requestors that do not have enough budget have to wait and accumulate enough budget to be eligible for service.

After identifying requestors that have enough budget to be considered for service, one of them is selected based on priority. The scheduling is based on static priority and the requestor that has the highest priority is scheduled. After every scheduling event, the budget of all requestors is updated in accordance with the scheduling result.

4.3.3 Architecture

The architecture in Figure 4.8 shows the three major functional units of the CCSP arbiter - *Rate Regulation*, *static-priority scheduling* and *Budget Management*.

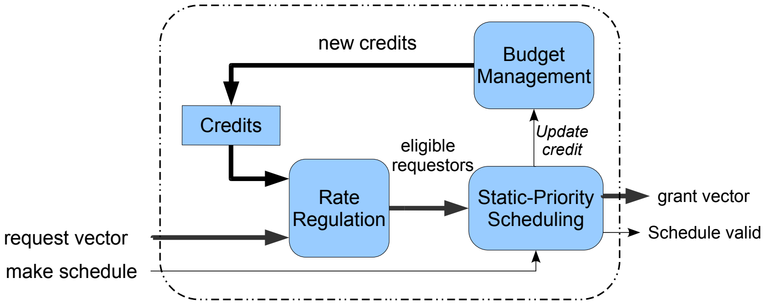


Figure 4.8: Architecture of CCSP Arbiter

4.3.3.1 Rate Regulation

The CCSP arbiter regulates service by watching the budget status of each requestor. This stage of the arbiter identifies requestors that are eligible for service based on their budget status and passes them to the next arbitration stage. To be eligible, requestors are required to have budget amount that exceeds a certain threshold. The threshold is also determined based on the service rate allocated to each requestor. The pseudocode of Algorithm 4.1 shows the simple process of identifying eligible requestors.

Algorithm 4.1 Eligibility Check in the CCSP Arbiter

```
GIVEN: R requestors with corresponding budget amount
OUTPUT: Eligibility Mask of requestors.
for each requestor r do
  if budget[r] > Budget Threshold[r] then
    Eligibility (r) ← TRUE;
  else
    Eligibility (r) ← FALSE;
  end if
end for
```

4.3.3.2 Static-Priority Scheduling

This stage of the CCSP carries out static-priority scheduling on those requestors that have been identified by the previous state as *eligible*. The result of the scheduling, *schedule mask* is sent to request multiplexer so that the scheduled request is handed over to the resource. Static-priority scheduling is a simple process of selecting the highest priority requestor.

Algorithm 4.2 Static-Priority Scheduling in the CCSP Arbiter

```
GIVEN: R requestors with corresponding priority values PRIORITY.
OUTPUT: Scheduled Mask which indicates the requestor that has been scheduled.
VARIABLE scheduled ← FALSE ;
for each requestor r in decreasing order of priority do
  if Eligibility(r) = '1' and scheduled = FALSE then
    schedule mask(r) ← '1'; scheduled ← TRUE;
  else
    $schedulemask(r) \leftarrow' 0'$;
  end if
end for
```

4.3.3.3 Budget Management

The budget for each requestor is updated according to the result of the scheduling. After each scheduling event, which takes place every service cycle, the budget manager upgrades the budget of each requestor by an amount equal to the allocated service rate. And for the requestor that is scheduled, if any, budget value is deducted by 1 because it has to pay for the service it has obtained. There is, however, a limit on the maximum budget that can be accumulated by requestors when they do not have valid request and that limit is equal to the *initial budget*. This is enforced in order to limit the burstiness of requestors to their allocated amount (σ').

Algorithm 4.3 Budget Management in the CCSP Arbiter

```
GIVEN: Current Budget of Each Requestor, Result of the Scheduling Decision.
OUTPUT: New Budget for Each Requestor.
Every Service Cycle
for each requestor r do
  Budget[r] ← Budget[r] + Allocated Rate [r];
  if schedule mask (r) = TRUE then
    Budget[r] ← Budget[r] - 1;
  else
    if requestor r has no request pending then
      //reset budget to initial amount
      Budget[r] ← Initial Budget[r];
    end if
  end if
end for
```

4.4 Resource Sharing Bus

This bus contains a timer, an arbiter and another unit that is responsible for multiplexing requests and demultiplexing responses. Among requests that are waiting for service, the request multiplexer sends one to the resource based on the scheduling decision made by the arbiter. A new scheduling decision is produced periodically provided that the resource is available for service. When responses are ready, the response demultiplexer in this bus returns them to the correct requestor.

The major units inside the resource sharing bus are shown in Figure 4.9.

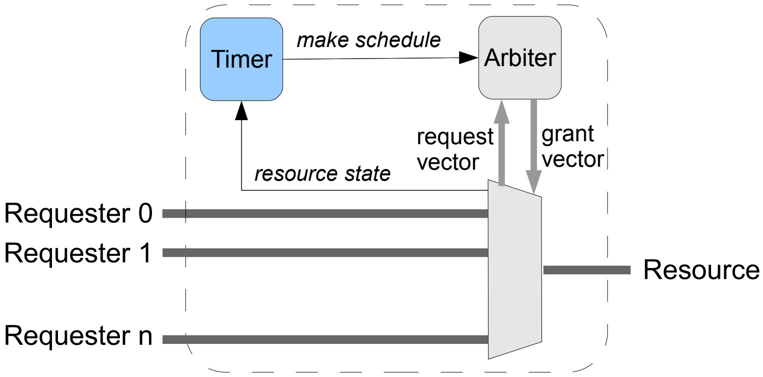


Figure 4.9: Architecture of Resource Sharing Bus

4.4.1 Timer

The timer controls the time at which new scheduling decision is made. It counts cycles up to a value called *Service Cycle* and initiates new scheduling decision. *Service Cycle* is the minimum number of cycles required to serve an atom. Making the scheduling decision at a later time may cause the resource to sit idle in case the previous request was served faster. A scheduling decision made earlier, on the other hand, does not reflect an up-to-date decision as higher priority requests may arrive in the meantime. Therefore, making the scheduling decision at this moment ensures effective utilization of the resource while keeping the scheduling decision as up-to-date as possible. In fact, a new request is scheduled only if the previously scheduled request has been accepted and the resource is ready for another one. Upon scheduling a request, the timer restarts counting for the next scheduling event.

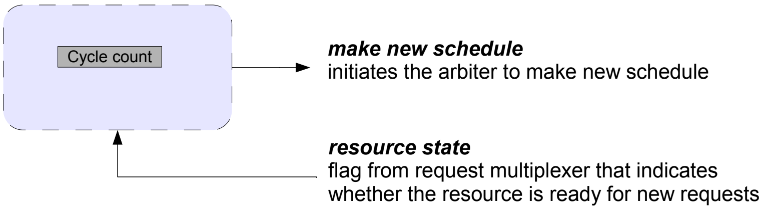

Cycle count
make new schedule
initiates the arbiter to make new schedule
resource state
flag from request multiplexer that indicates
whether the resource is ready for new requests

Figure 4.10: Service Timer

4.4.2 Request Multiplexer and Response Demultiplexer

The resource sharing bus has a request multiplexer and a response demultiplexer. Decoupling the request and response phases makes it possible to pipeline requests and utilize the resource 100When a scheduling decision is made by the arbiter, the request multiplexer selects a request accordingly and presents it to the resource. In the meantime, the *resource state* is indicated as busy and the timer is prevented from initiating new scheduling decision. New schedule can be made only when the request at hand has been fully accepted by the resource. The bus has a parallel function that takes care of responses. When the resource finishes serving a request, it is the responsibility of this unit to return the resulting responses to the correct requestor. To serve this purpose, the request multiplexer records scheduling history, which is the order in which requestors are scheduled. Responses are expected in the same order in which the corresponding requests are sent. Hence request reordering by the resource is not supported.

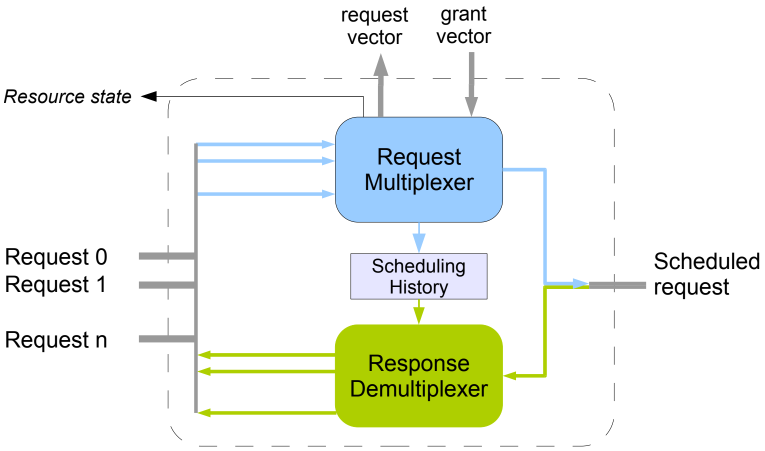


Figure 4.11: Multiplexing Requests and Demultiplexing Responses

4.5 Configuration

The proposed front-end is comprised of stand alone functional blocks connected with each other using standard communication protocol. This makes it possible to eliminate a block from the design when the associated feature is not required. Once a block is included in the front-end, it can be instantiated with proper settings so that it fits well with the requirements. This represents design time configuration of the individual functional blocks. The third phase of configuration is at run time. In a SoC with multiple functionalities, the set of applications that are active simultaneously changes from time to time. This change of use case results in variable service requirements. To satisfy such dynamic requirements, some parameters in the front-end are made configurable at runtime. The three phases of configuration - Feature selection, design time configuration and Run time Configuration - are discussed in Sections 4.5.1, 4.5.2 and 4.5.3, respectively.

4.5.1 Front-end Feature Selection

Depending on the features required in the design, some of the components of the front-end can be eliminated in order to save hardware.

4.5.1.1 Atomizer:

As mentioned above, the role of the Atomizer is to chop big requests in to unit size before they enter the Delay Block. If a requestor is known, at design time, to produce requests not larger than an atom, chopping requests becomes unnecessary and hence the Atomizer can be removed for that requestor.

4.5.1.2 Delay Block:

The Delay Block is required only when composable service is required. There is a situation where some requestors in the design require composable service while others do not. In such situations, Delay Block should be added only to those requestors that need composability. There is also a possibility that a requestor needs composability under some use cases and does not in others. In such cases the Delay Block is kept and whenever composability is not required, it is disabled by setting its delay parameters to zero. With delay parameters set to zero, the Delay Block releases flow-control signals and responses at the actual time when they are produced.

4.5.1.3 Arbiter:

The arbiter that resides in the Resource Sharing Bus can be anything in the class of $\mathcal{LR}-$servers. Hence, the most suitable arbiter that satisfies the needs is chosen at design time.

4.5.2 Design Time Configuration (Block Customization):

Once a decision has been made on which features to include in the front-end, the required blocks are instantiated with appropriate design parameters. These parameters are determined by the layout of the system, the resource attached and the communication protocol used between system components. The following are some of the front-end parameters that are configurable during instantiation.

4.5.2.1 Number of requestors:

When layout of the system and communication requirements are identified, the number of requestors to a given resource can be known. This determines then number of target ports that the Resource Sharing Bus should have. The arbiter is also instantiated properly according to number of requestors sharing the resource.

4.5.2.2 Chopped Request Size

This is the preferred size of requests that enter the resource. For instance, efficiency of read/write operations in an SDRAM can be maximized by choosing an optimal request size. This parameter is required in the Atomizer. As the resource is known at design time, this parameter is used while instantiating the Atomizer and the bus.

4.5.2.3 Service Cycle

Service cycle, according to the definition in Section 4.4.1, depends on the resource attached and the size of requests that enter the resource. Thus, as soon as properties of the resource and size of chopped requests are known, the *service cycle* can be set in the Resource Sharing Bus.

4.5.2.4 Communication parameters

Blocks in the front-end are connected to each other via standard communication protocols, such as *Device Transaction Level (DTL) Protocol* [13]. The width of each signal, such as *data width* and *address width*, are set at design time.

4.5.2.5 Buffer Sizes

Generally, big buffers help to achieve more throughput amid a bursty traffic. To save cost, however, all buffers are sized to a minimal value that can satisfy application requirements. The size of buffers inside each block of the front-end are generic parameters that can be set during instantiation of the blocks. The values are automatically computed using data flow models based on the service requirements of each requestor.

4.5.3 Run-time Configuration

The CCSP arbiter and the Delay Block are programmable components of the front-end. Each of these has parameters that can be set at run time and each unit starts functioning only after all parameters have been programmed. The CCSP Arbiter requires three parameters to be set for each requestor - *Priority (p)*, *Allocated Service Rate (ρ')* and *Allocated Burstiness (σ').* Similarly, each Delay Block has two programmable parameters

-*Service Latency* (Θ) and *Completion Latency* (λ) - that need to be set before starting operation.

4.5.3.1 Configuration Infrastructure

We use the configuration infrastructure presented in [5], *pages 60-62* for run time configuration of our front-end. The configuration values are sent by an IP, usually called the *host*, that is responsible for controlling the SoC. A bus with fixed-address decoding, *configuration bus*, takes the responsibility of delivering configuration data to the intended component. The configuration port of each programmable IP has unique address known by the controlling IP at design time. Thus configuring a unit done transparently as a write operation to the address corresponding to the required configuration port. Figure 4.12 shows how programmable components of the front-end are connected to the configuration bus.

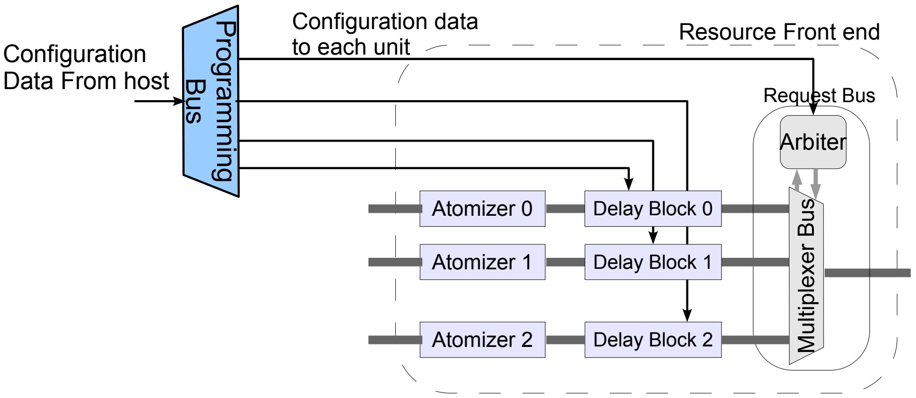


Figure 4.12: Configuration Infrastructure

Implementation 5

In this chapter, we discuss the implementation of the resource sharing front-end based on the design presented in Section 4. Discussion is made on mechanisms employed to realize the aforementioned design in hardware. We begin with Section 5.1, where preliminary implementation issues are discussed. In Section 5.2, the implementation of each functional block in the front-end is discussed. Then, in Section 5.3, implementation of the configuration mechanism is presented. The chapter is concluded by showing how our design is integrated to the **Æthereal Design Flow** to automate the design process.

5.1 Preliminaries

5.1.1 Communication Protocol

To make the individual blocks of the front-end reusable, each one is implemented with standard interfaces for communication with other blocks. The DTL Protocol [13] has been used in this implementation of the front-end.

DTL is a point-to-point protocol that connects an initiator port of an IP to the target port of another IP. Initiator is the device that initiates transactions. The initiator drives commands and associated data, if any, to be written to the target. The initiator may receive read data as response from the target. The target is connected to the other end of a DTL connection. It receives command and potential write data from the initiator. For read transactions, it responds by sending back read data to the initiator. DTL is a synchronous protocol with all signal transitions synchronized to rising edge of the clock.

In the complete specification of the DTL protocol, the signals are categorized in to five groups - *System Group*, *Command Group*, *Write Group*, *Read Group*, *Write Buffer Management Group* and *Error Abort Group.* For our implementation, the features provided by the first four groups are sufficient and only these signals have been used. Table 5.1 defines the DTL signals that have been used in our implementation.

Figure 5.1 shows the direction of all the signals during communication between an initiator and a target. Grouping of the signals in to *command*, *payload* and *response* is also shown. Transaction is started when the initiator sets the value of signals in the command group. Validity of the command is indicated by the *dtl_cmd_valid* signal. The transfer of command is completed when the target accepts it by setting the *dtl_cmd_accept* signal. If it is a write request, transfer of the associated payload data follows. In the same manner as with command group, validity of signals in this group is indicated by the *dtl_wr_valid* signal. A data word is transferred at every clock cycle when both *dtl_wr_valid* and *dtl_wr_accept* are asserted. The transfer of payload data, and hence the command, is completed when the last word of the payload data (indicated by *dtl_wr_last*) is transferred. Signals in the *Command* group and *Write* group constitute a request.

Table 5.1: Definition of Relevant Signals in the DTL Protocol

Name	Driver	Description
System Group		
clk	Input to initiator and target	This is the main clock for communication. All other signals are synchronized to this clock signal.
rst_n	Input to initiator and target	This active low signal is used to initialize the DTL interface.
Command Group		
cmd_valid	Initiator Command Valid	This signal is used to indicate that the signals in the DTL Command group are valid. This is driven high to transfer a new command from the initiator to the target.
cmd_accept	Target Command Accept	This signal is used to indicate that the signals in the DTL Command group have been accepted by a target. Command transfer is completed when *cmd_valid* and *cmd_accept* are high simultaneously .
cmd_addr	Command Address	This represents the byte address associated to the active command.
cmd_read	Initiator Command Read Operation	This signal is used to indicate whether the active transaction is a *read* or *write* operation. When high, it indicates read operation otherwise a write operation.
cmd_block_size	Command Block Size	Indicates the number of elements to be transferred over the *wr_data* or *rd_data* lines.
Write Group		
wr_valid	Write Valid	This signal is used to indicate that the signals in the DTL Write group are valid. This is driven high to transfer new write data (and a mask) from the initiator to the target.
wr_accept	Target Write Accept	This signal is used to indicate that the data and control signals in the Write group have been accepted by a target. Transfer of a data element is completed when *wr_valid* and *wr_accept* are high simultaneously .
wr_data	Write Data	These bits represent the actual data sent to the target.
wr_mask	Write Data Byte Mask	The bits are used to indicate which bytes should be written during a write data transfer. Each bit corresponds to a byte in *wr_data* group and only those bytes that have the corresponding bit high are written.
wr_last	Write Data Last	Indicates that the current data transferred on the *wr_data* lines is the last of the current transaction.
Read Group		
rd_valid	Read Data Valid	This signal indicates that the signals in the DTL Read group are valid. This signal is driven high to transfer new data read from the target.
rd_accept	Initiator Read Accept	This signal is used to indicate that the data and status signals in the DTL Read group have been accepted by an initiator. Read data is transferred when *rd_valid* and *rd_accept* are high simultaneously .
rd_data	Read Data	These lines hold the data read from the target.
rd_last	Read Last	Indicates that the current data is the last one read from the target.

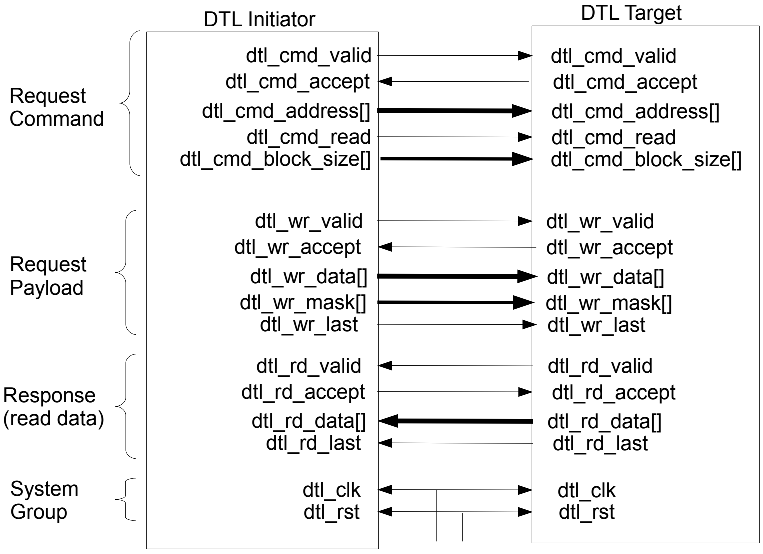


Figure 5.1: Direction and Grouping of DTL Signals

The *Read* group represents data read from the resource, which is response for a read request. The target presents read data to the initiator by setting signals in the read group. The read data is valid when the signal *dtl_rd_valid* is set. The initiator, then, accepts the response by setting the signal *dtl_wr_accept*. The last word of a response is indicated by *dtl_wr_last*.

The DTL protocol supports default accept for *command, write* and *read* groups. If the target is able to accept any command, it sets the *dtl_cmd_accept* signal high even before the command is presented to it. The *dtl_wr_accept* signal can also be set before the data arrives if the target is able to accept the payload. Similarly, if the initiator is able to accept read data, it sets *dtl_rd_accept* even before receiving the data. This feature is required to achieve 100implementation.

5.1.2 Representation of Time

Time, in the Delay Block, is represented as count of clock cycles. The cycle count is global within the Delay Block and is used by all units. When a request arrives, the corresponding time stamps are computed in cycles and recorded. Then, events like scheduling of requests and releasing of responses are fired up when the cycle count reaches the corresponding time stamp.

With real hardware implementation, however, there is a limit on the number of bits that can be used for counting and hence the cycle count can not be incremented indefinitely. One solution is using a circular counter that wraps around to zero after reaching the maximum allowable value. Wrapping, however, makes it difficult to compare two time values. For instance, with N bits, the cycle counter can count from 0 to (2^N-1) and back to 0 producing the sequence (0, 1, 2,..., 2^N-1, 0, 1,...). In this sequence, it is difficult to say whether 2^N-1 is before or after 0. 2^N-1 can be considered as a time value that is 2^N-1 cycles later than 0 or it can be considered to be 1 cycle earlier than 0. An extra bit is used in our implementation to avoid this ambiguity, viz. *MOD*-2^{N+1} circular counter, with *N+1* bits, is required to handle time range of [0, 2^N-1].

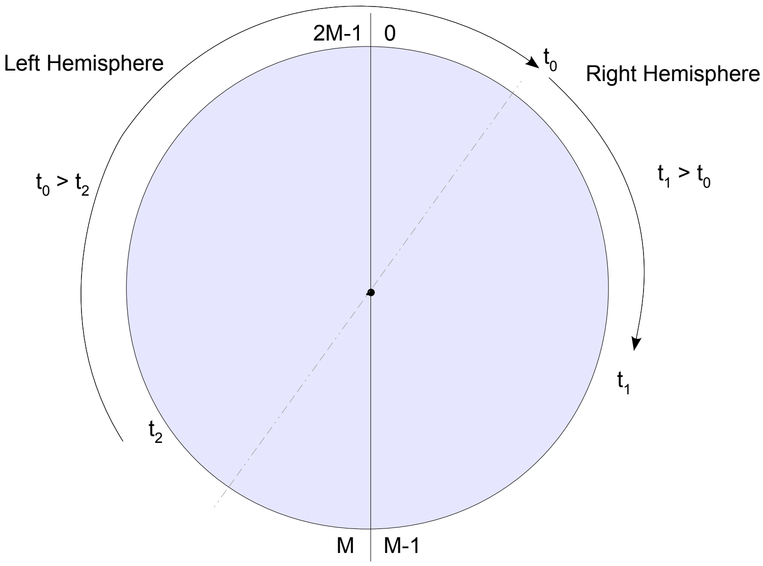


Figure 5.2: MOD-2M Circular Counter, where M=2^N

The maximum span between two time values that can be handled by this counter is M = 2^N-1. With this constraint in mind, the order between any two points in time is determined by the *shortest clock-wise* arrow between them. The time value at the arrow tip is considered to be later than the one at the other tip. Figure 5.2 illustrates counting in a circular counter and comparison between time values. For example, the three arrows indicate that t_0 is earlier than t_1 but later than t_2.

It should be emphasized that for this representation of time to be valid, the time gap between any two related events should not exceed 2^N. For instance, in the Delay Block the actual cycle count between arrival of request and the release of the corresponding response should be within this limit. Thus the number of bits required to represent cycle count is determined by the longest possible interval between two related events.

Comparison is done by taking the signed difference between two time values and checking the two most significant bits of the result. For instance, to check is a certain time is due we compare the time stamp with the current time. Consider two time values x and y. With our representation either of them can have value in the range [0, 2M-1]. Hence, the difference *(x-y)* lies in the range [-2M+1, 2M-1]. This range can be broken down to four disjoint sub-ranges - [-2M+1, -M-1], [-M, -1], [0, M-1] and [M, 2M-1]. Then the range where *(x-y)* lies tells which of the two time values is earlier. The meaning and implication of these four sub ranges is explained in Table 5.2. To find out the sub range in which x-y lies, we look at the two most significant bits (MSB) of the difference. With 2's complement notation for signed numbers, the mapping between the values of the two MSBs and the sub ranges is shown in Table 5.3.

Table 5.2: Comparison of Two Time Values x and y based on their difference *(x-y)*

Range for $(x - y)$	Position of x and y	Result of Comparison
[-2M+1, -M-1]	x lies in the right hemisphere and y in left hemisphere	x is later than y
[M+1, 2M-1]	x lies in the left hemisphere and y in the right hemisphere	x is earlier than y
[-M+1, -1]	Multiple cases possible	x is earlier than y
[1, M-1][a]	Multiple cases possible	x is later than y

[a] Obviously, x-y = *0* implies that x and y are equal.

Table 5.3: Identifying the sub range for *(x-y)*

The two MSB's of *(x-y)*	sub range
"10"	[-2M+1, -M-1]
"01"	[M, 2M-1]
"11"	[-M, -1]
"00"	[1, M-1]

5.2 Functional Blocks

As outlined in Chapter 4, the front-end has four major functional blocks. Based on the design presented there, the actual implementation is presented next. For each block, the representation of parameters and realization of functions are detailed. Among others, algorithms and Finite State Machines (FSMs) that are used in the various processes are explained.

5.2.1 Atomizer

5.2.1.1 Units of the Atomizer

As outlined in the design, Section 4.1, the Atomizer has two major tasks - Chopping requests and Merging responses. The implementation of the two units that carry out these task is discussed next.

Unit: Request Chopper

The FSM in Figure 5.3 shows the states through which the chopper advances while chopping and sending requests. The Chopper starts from *WAITING* state and stays there until it receives a request to be chopped. Depending on the request received, it can move to either *CHOP READ REQUEST* state or *CHOP WRITE REQUEST*. Chopping a read request involves sending commands with new headers (Header implies signals in the command group). Whereas in chopping write requests, part of the payload is sent following each chopped command. The chopper enters the *SEND PAYLOAD* state to send part of the payload data that belongs to each chopped command.

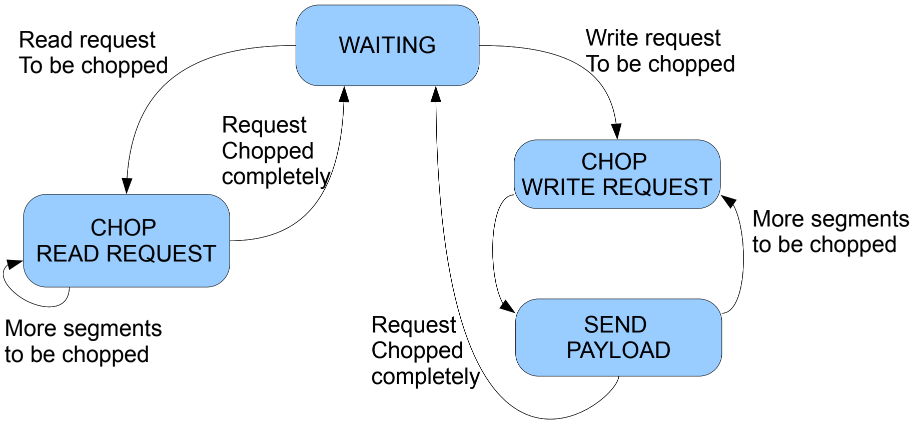


Figure 5.3: FSM for Chopping Requests

Request Size Buffer: Whenever the atomizer chops a request, it records the original size of requests so that it can be used for identifying responses later. With the DTL protocol, which we used in this implementation, the size corresponds to the *dtl_cmd_block_size* signal. Hence the width of the buffer is determined by the dtl specification. We used 5-bits to represent request sizes and hence the width of *Request Size Buffer* is 5 bits.

Unit: Response Merger

As described in the specification the DTL protocol, a group of read responses that correspond to the same request are identified by by a marker (*dtl_rd_last*) to the last response in the group. Merging multiple read responses is, thus, done by removing the last marker from all responses but the last one in the merged group. The following pseudocode represents the merging process.

Algorithm 5.1 Merging Responses in the Atomizer

```
GIVEN : Atomic Responses from the Delay Block.
OUTPUT : Merged Response.
-count ← 0;
while (TRUE) do
  - Read request size at the front of Request Size Buffer
  if valid response is received then
    - Remove Last marker, if there is any.
    - count ← count + atomic_size
    if count = request size then
      - Add Last marker to the response at hand.
      - count ← 0;
    end if
    - Send the response.
  end if
end while
```

5.2.2 Delay Block

5.2.2.1 Representation of Delay Block Parameters

Both Service Latency and Completion Latency are represented with cycle counts. However, these two values are not integer numbers. For instance, *completion latency* is derived from the allocated service rate (ρ'), viz. $\lambda = 1/\rho'$. As the resulting value is a rational number, representation in cycles requires rounding of $1/\rho'$ to the nearest integer value. Such approximation errors, however, accumulate and affect the service provided. As explained in [2], rounding up ($\lceil 1/\rho' \rceil$) leads to a worst-case finishing time that is too pessimistic and leads to a service rate that is less than ρ'. Rounding down ($\lfloor 1/\rho' \rfloor$), on the other hand, leads to a worst-case finishing time that is earlier than the correct value. As this error accumulates, it can reach a point where responses are unavailable at the computed worst-case finishing time. This leads to a non-composable behavior and needs to be solved. Figure 5.4 shows the consequence of approximating the completion latency. The two dotted lines ($\lceil 1/\rho' \rceil$ and $\lfloor 1/\rho' \rfloor$) represent the service curves that result from rounding up and rounding down $1/\rho'$ respectively.

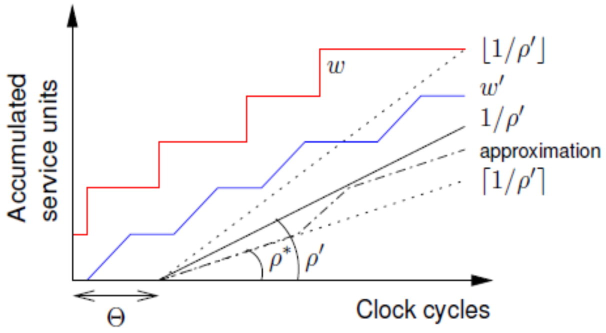


Figure 5.4: Accumulation of Error due to Approximation of λ

The approach we used to solve this problem is by representing the completion latency as a formal rational number.

$$\lambda = \lambda_I \frac{\lambda_n}{\lambda_d} \tag{5.1}$$

where λ_I stands for integer part of λ and λ_n and λ_d represent numerator and denominator of the fractional part, respectively. The error that results from approximating service latency (Θ), however, does not accumulate (Equation (4.1)). Hence, it can be rounded up to the next integer. The mechanism that we used to approximate λ is listed as Algorithm 5.4 in Section 5.2.2.4.

In the end,we require four integer values to represent the two Delay Block parameters : Θ, λ_I, λ_n and λ_d. Hereafter, the symbol λ is used to refer to the value of the completion latency as a whole and its components are referred using λ_I, λ_n *and* λ_d, accordingly. With the new representation of completion latency as formal rational number, the accuracy is determined by the number of bits used for each value.

- **Service Latency (Θ):** The value of service latency for each requestor is calculated based on the total number of requestors that share the resource, priority of the requestor, and its service requirement *(bandwidth and burstiness)*. Hence, the number of bits required to represent Θ is decided at design time when these characteristics are known.

- **Completion Latency Integer Part (λ_I):** As explained in the design section, completion latency stands for the maximum amount of time, in cycles, that a resource takes to serve a unit-sized request considering service at the allocated rate (ρ'). The number of bits required to represent this parameter is determined at design time, when properties of the resource are known.
- **Completion Latency Denominator Part (λ_d):** With the formal representation of λ_d, accuracy of the fractional part is determined by the number of bits used to represent the denominator and the numerator. With D-bits used to represent the denominator, the precision that we can have is $\frac{1}{2^D}$.
- **Completion Latency Numerator Part (λ_n):** Since λ is represented as a formal rational number, the range of values that λ_n can have is [0, λ_d-1). Hence, the number of bits required is the same as that for λ_d i.e. *D-bits.*

The number of bits required to represent the above parameters are generic parameters in our design. Hence they can be set at design time when the required information regarding requestors and the resource are known.

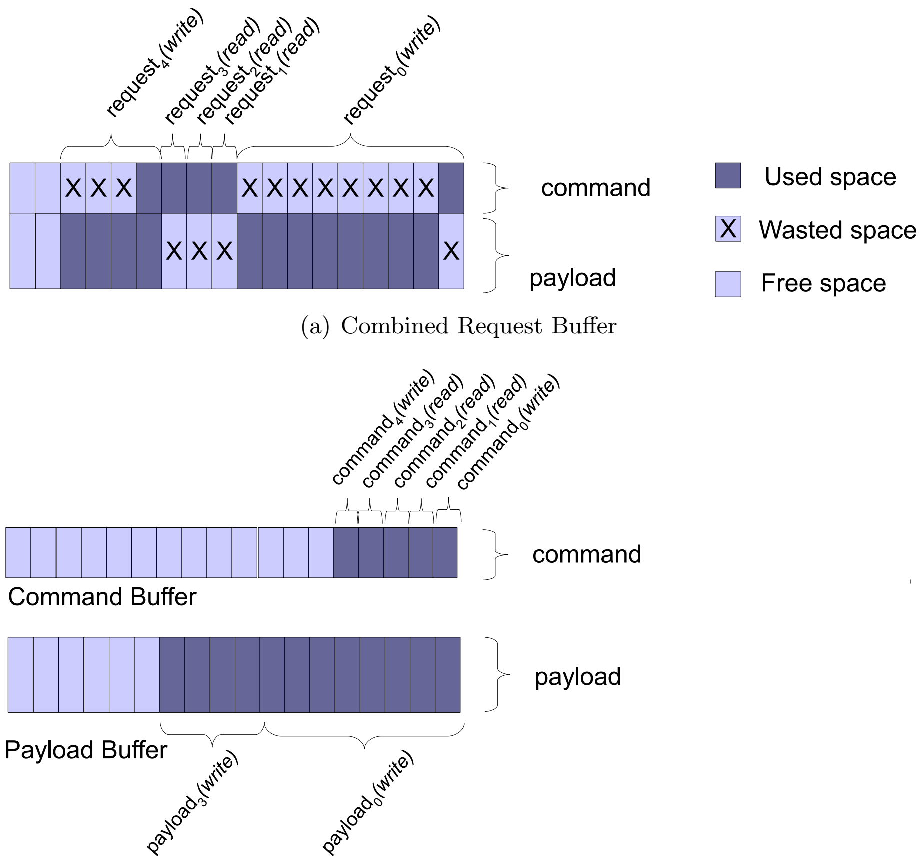
The accuracy (number of bits required to represent λ_n and λ_d) is crucial not only for the sake of hardware cost but also for its impact on operating frequency of the Delay Block. As the critical path of our design is through the process that computes time stamps, shown in Figure 7.2, the number of bits used here affects the performance of our design. Hence the number of bits is chosen for optimal precision and operating frequency.

5.2.2.2 Buffers

FIFOs have been used to store requests, responses and time stamps in sequence. Requests are buffered as they enter the Delay Block and responses are buffered until they are released. Two time stamps are tracked in our design - *worst-case scheduling time* and *worst-case finishing time.* All buffers are implemented as synchronous FIFOs. The implementation of each one is detailed next.

Request Buffer

With the DTL protocol, as explained in Section 5.1.1, a request is composed of *command* component and *payload* component. The Delay Block requires both the command and the payload, if any, components of a request to arrive before considering it as valid (Section 4.2.3). Hence both components need to be buffered in the Delay Block as they arrive. Storing both *command* and *payload* in a single wide buffer results in wastage of storage space. Because when requests without payload are buffered, the portion corresponding to payload is unused and hence unused. Moreover, the command portion is used only for the header of each request. Otherwise it is wasted (Figure 5.5(a)). To solve this problem we use two separate FIFO buffers in parallel - one for the command component and another for the payload. Figure 5.5(b), illustrates the benefit of this



(a) Combined Request Buffer

(b) Parallel FIFO Buffers for Command and Payload Parts

Figure 5.5: Request Buffer (a),(b)

approach. Command and Payload portions are used only as needed and hence no space is wasted. The depth of the buffers is determined based on throughput requirements of requests.

Response Buffer

This is a buffer that holds responses until they are released and leave the Delay Block. Responses for read requests correspond to signals in the *Read Data* group of the DTL protocol. For write requests, signal about the completion of a request is considered as response. Hence the response buffer provides space for both read data and write responses. While the width of this FIFO buffer depends on the DTL specification, the depth is chosen in such a way that throughput requirements are met.

Scheduling Time Buffer

Time stamp for scheduling event is recorded for each request that is received and validated in the Delay Block. Entries are removed from this FIFO buffer whenever the worst-case scheduling time that they represent is due. As a result the status of this buffer emulates worst-case scheduling of requests. As explained in Section 5.1.2, time stamps represent cycle count. The width of this buffer is hence equal to the number of bits chosen to represent cycle count. The depth is, once more, chosen based on throughput requirements.

Finishing Time Buffer
Entries in this FIFO represent time stamps at which responses should be released. Hence this FIFO buffer has the same width as that of *Scheduling Time Buffer.*

5.2.2.3 Counters

The Delay Block maintains a few counters to track the status of requests in the Delay Block.

1. **Full requests count ($n_{full\ requests}$):** represents the actual number of requests that have been fully received by the Delay Block. For read request, acceptance of the header (command part) suffices. For a write request, all the associated payload data has to be received as well. In other words, this counter represents the number of requests that are validated and are waiting to be scheduled for service.

2. **Waiting requests count ($n_{requests\ waiting}$):** is count of requests that would have been waiting in the Delay Block, if every request were to be accepted at the worst-case scheduling time. The actual number of requests that are waiting, in the Delay Block, to be scheduled is less as most requests are scheduled before the worst-case time. This counter reflects the worst-case filling of the *Request Buffers.*

3. **Released responses count ($n_{released}$):** represents the number of responses that have their release time due but have not left the Delay Block. For instance, congestion in the communication path can cause responses to stay in the Delay Block after the worst-case finishing time.

4. **Free response space ($n_{response\ space}$):** represents free space available in the response buffer. For every request that produces response, space is reserved in the response buffer its arrival is validated. And space is freed up as responses leave the Delay Block.

In addition to the above counters, the following two have been included to be used in assertion statements, explained later in Section 5.2.2.5.

1. **Accepted requests count ($n_{accepted}$):** counts requests that have left the Delay Block before their worst-case scheduling time. A request belongs to this group during the time between actual scheduling and worst-case scheduling.

2. **Waiting responses count ($n_{waiting\ responses}$):** represents the number of responses that are waiting to be released.These are responses that are withheld by the Delay Block until their worst-case finishing time is due.

The management of these counters is explained during the discussion of the various processes in Section 5.2.2.4. Later on, in Figure 5.10, the counters and their managing processes are illustrated.

5.2.2.4 Units of the Delay Block

The implementation of each unit of the Delay Block, shown in Figure 4.7, is discussed next.

Unit: Receive Requests

This process is responsible for getting requests into the Delay Block. Whenever there is a valid request and the Delay Block has space to accommodate it, this process stores the request and the associated payload data, to the *request buffer*. After the full request arrives, it has to be validated before getting service. Figure 5.6 illustrates the FSM used to receive and validate requests.

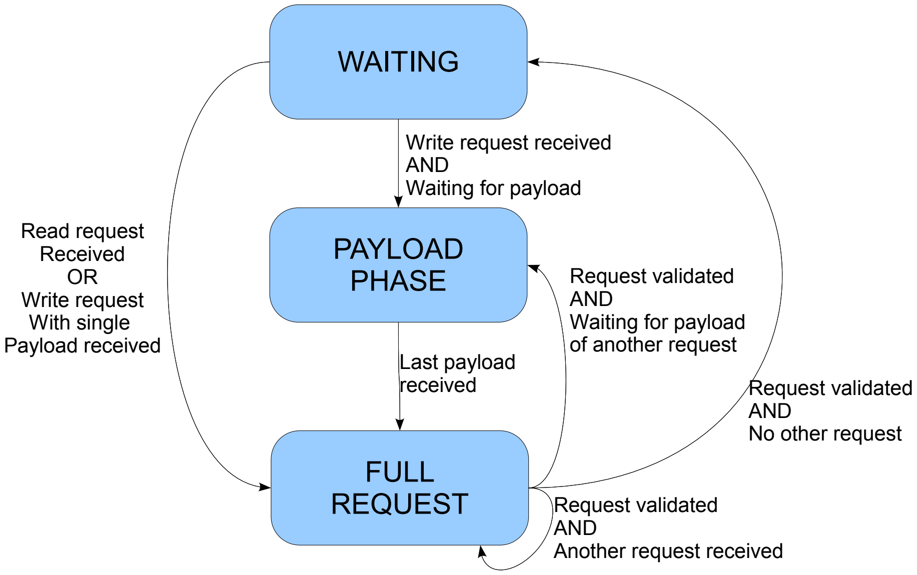

WAITING
Write request received AND Waiting for payload
PAYLOAD PHASE
Read request Received OR Write request With single Payload received
Request validated AND Waiting for payload of another request
Last payload received
Request validated AND No other request
FULL REQUEST
Request validated AND Another request received

Figure 5.6: FSM for Receiving and Validating Requests

Arrival of a request is validated when two conditions are fulfilled :

1. The request along with all its payload, if it has any, should be in the Delay Block.
2. There should be enough space to be reserved in the *response buffer* for all responses that the request produces.

Upon validating a request, computation of the corresponding time stamps is initiated and the counters $n_{full\ requests}$ and $n_{requests\ waiting}$ are incremented. Figure 5.7 shows the conditions required to validate arrival of a request and the operations performed upon validation. The check for response space is done by looking at the value of the counter $n_{response\ space}$. If there is enough free space, reservation is made by deducting the required amount from the counter.

The pseudocode in Algorithm 5.2 lists down operations performed when a request is validated. The time at which a request is validated is recorded as its arrival time (t_a^k).

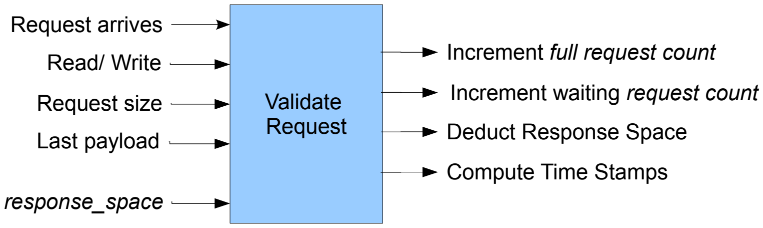


Figure 5.7: Validating Arrival of Requests in the Delay Block

Algorithm 5.2 Validating Arrival of a Request in the Delay Block

```
while (TRUE) do
  if a request arrives in the Delay Block AND
  $n_{requests\ waiting} < Depth\ of\ Request\ Buffer$ then
    if It is a $READ$ request then
      if $n_{response\ space} \geq request\ size$ then
        // There is free space for responses
        $n_{full\ requests} \leftarrow n_{full\ requests} + 1$ ;
        $n_{requests\ waiting} \leftarrow n_{requests\ waiting} + 1$ ;
        $compute\ time\ stamps \leftarrow' 1'$ ;
        $n_{response\ space} \leftarrow n_{response\ space}$ - $request\ size$ ;
        $t_a^k \leftarrow current\ time$ ;
      end if
    else
      if Last Payload is received then
        $n_{full\ requests} \leftarrow n_{full\ requests} + 1$ ;
        $n_{requests\ waiting} \leftarrow n_{requests\ waiting} + 1$ ;
        $compute\ time\ stamps \leftarrow' 1'$ ;
        $t_a^k \leftarrow current\ time$ ;
      end if
    end if
  end if
end while
```

Unit: Compute Time Stamps

When computation of time stamps is initiated by the request validator, worst-case scheduling time (t_{SW}^k) is computed for the request at hand. The computation is based on arrival time of the request(t_a^k), service latency (Θ) and the worst-case finishing time of the previous request (t_{FW}^{k-1}). The other time stamp, *worst-case finishing time*, is computed by adding λ to the corresponding *worst-case scheduling time*. The computation of the worst-case scheduling time is described by the pseudocode of Algorithm 5.3. At the end of the computation, t_{SW}^k is recorded in the *Scheduling Time Buffer*, as illustrated in Figure 5.8.

In Figure 5.8, two different types of adders are shown. The first one, which computes $t_a^k + \Theta$, is a simple integer adder because t_a^k is integer and the rounded up value of

Algorithm 5.3 Compute Worst-case Scheduling Time for a Request

```
When the $k^{th}$ request is validated
if $(t_a^k + \Theta) > t_{FW}^{k-1}$ then
    $t_{SW}^k = t_a^k + \Theta$;
else
    $t_{SW}^k = t_{FW}^{k-1}$
end if
```

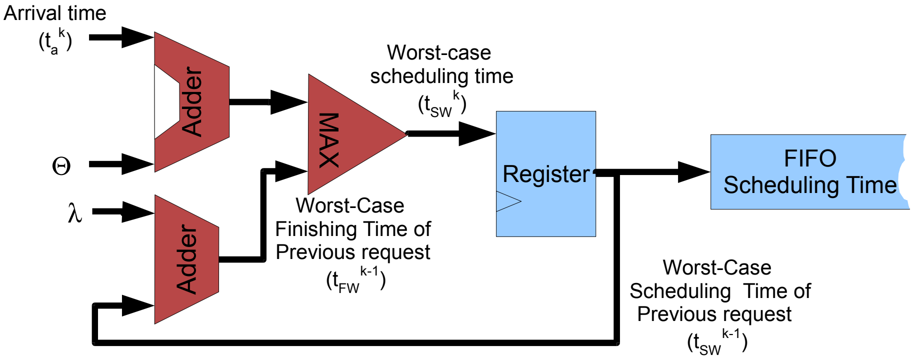


Figure 5.8: Computation of Worst-Case Scheduling Time

service latency (Θ) is used. The second adder, which computes $t_{SW}^{k-1} + \lambda$, on the other hand, performs the addition using the approximation mechanism in Algorithm 5.4 The mechanism uses a credit counter, c_r, to switch between using rounded up and rounded down values of the completion latency (λ). When requests arrive before the worst-case

Algorithm 5.4 Compute Worst-case Finishing Time for a Request

```
When the $k^{th}$ request is validated
if $(t_a^k + \Theta) > t_{FW}^{k-1}$ then
    $c_r \leftarrow 0$;
end if
if $(c_r < \lambda_d - \lambda_n)$ then
    // Use the rounded up value of $\lambda$ i.e. $\lceil 1/\rho' \rceil$
    $c_r \leftarrow c_r + \lambda_n$;
    $t_{FW}^k \leftarrow MAX((t_a^k + \Theta),\ t_{FW}^{k-1}) + \lceil 1/\rho' \rceil$
else
    // Use the rounded down value of $\lambda$ i.e. $\lfloor 1/\rho' \rfloor$
    $c_r \leftarrow c_r + \lambda_n - \lambda_d$;
    $t_{FW}^k \leftarrow MAX((t_a^k + \Theta),\ t_{FW}^{k-1}) + \lfloor 1/\rho' \rfloor$
end if
```

finishing time of their predecessor, it represents a busy period. During a busy period, the worst-case scheduling and finishing time stamps are determined by the worst-case finishing time of their predecessor. With this approximation mechanism, the credit counter (c_r) is set to zero at the start of a busy period. Then, as long as it is a busy period, the mechanism alternates between the rounded up and rounded down value of λ for use in computing the worst-case finishing time. Whenever the value of the counter c_r is above $(\lambda_d - \lambda_n)$, the rounded down value of λ is used; otherwise the rounded up value is used. The resulting approximation is conservative and guarantees that the deviation from the exact value is always less than a clock cycle.

Although the value of the worst-case finishing time (t^k_{FW}) is known upon arrival of the request, it is not recorded to the *Finishing Time Buffer* until the worst case scheduling time is due. This helps to reduce the size of the finishing time buffer. It is shown in 5.9 when the worst-case finishing time of requests is recorded. When the worst-case scheduling time of a request is due, the worst-case finishing time is recomputed by adding λ and recorded to the *Finishing Time Buffer*.

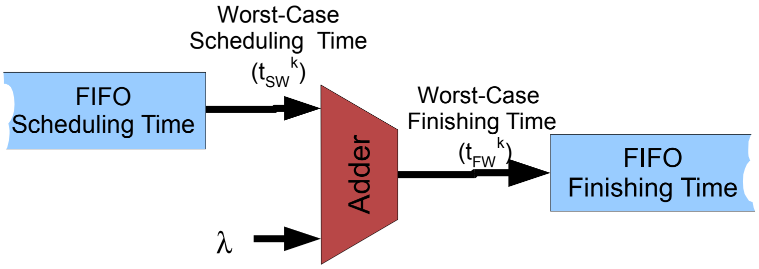


Figure 5.9: Computation of Worst-Case Finishing Time

Unit: Send Requests

Whenever there is a validated request residing in the Request Buffer, the *Send Request* process presents it to the arbiter and waits until it is scheduled for service. When the command is accepted, the entry is removed from the *Request buffer* and sending payload data, if the request has any, follows. For every request that leaves the Delay Block, the counter $n_{accepted}$ is incremented by one. $n_{accepted}$ is decremented only when the corresponding worst-case scheduling time is due.

Unit: Handle Scheduling Event

This process encompasses all operations to be executed during a scheduling event. The process watches the *Scheduling Time* buffer and when the cycle count reaches the time stamp at the front, it invokes the operations listed by the pseudocode in Algorithm 5.5. At every scheduling event, the counter $n_{accepted}$, which represents the number of requests scheduled before the worst-case scheduling time, is decremented. $n_{requests\ waiting}$, worst-case count of requests waiting in the request buffer of the Delay Block , is also decremented at each scheduling event. If the request scheduled is expected to produce response, this process computes the worst-case finishing time and records the time stamp in the *Finishing Time Buffer*.

Unit: Receive Responses

When the resource finishes serving a request, it returns the resulting responses, if there are any. This process is responsible for storing responses to the *Response Buffer*, where

Algorithm 5.5 Operations During a Scheduling Event

```
while (TRUE) do
  if $current\ time = t^k_{SW}$ then
    - Remove the time stamp $t^k_{SW}$;
    $n_{accepted} \leftarrow n_{accepted} - 1$;
    $n_{requests\ waiting} \leftarrow n_{requests\ waiting} - 1$;
    if $t^k_{SW}$ corresponds to a READ request then
      $t^k_{FW} \leftarrow t^k_{SW} + \lambda$;
      - Record $t^k_{FW}$ to Finishing Time Buffer;
    end if
  end if
end while
```

they stay until they are released. In our design, space is reserved for responses right at the arrival of the corresponding requests. As a result, responses are always guaranteed to be accepted by the Delay Block. Whenever a response is received, $n_{waiting\ responses}$ (the number of responses waiting to be released) is incremented.

Unit: Release Responses

Like the scheduling event, the process of releasing responses is started when the cycle count reaches the time stamp at the front of the *Finishing Time Buffer.* During each release event, a response in the *Response Buffer* is marked as released. The marking is performed by incrementing $n_{released}$, which is the count of responses that have their release time due and are waiting to leave the Delay Block. At the same time, $n_{waiting\ responses}$ is decremented.

Unit: Send Responses

Responses in the Delay Block that have their release time due, are bound to be sent to the requestor as soon as it is possible. Whenever there is a released response waiting in *Response Buffer* and the requestor is able to accept , the response is sent by this process. For every response sent, response space is freed up for upcoming requests by incrementing the counter $n_{response\ space}$. The counter $n_{released}$, which represents the number of responses waiting to leave the Delay Block, is also decremented.

The various counters maintained in the Delay Block and those processes that manage the counters are shown in Figure 5.10. Each counter has two sides, labeled '+' and '-'. The side labeled '+' is connected to the process that increments its value. Similarly, the side labeled '-' is connected to the process that decrements its value.

5.2.2.5 Assertions

In order for the Delay Block to bring about composability, predictable service should be offered at all times. As long as the service is predictable, the following two assertions should be valid. Namely,

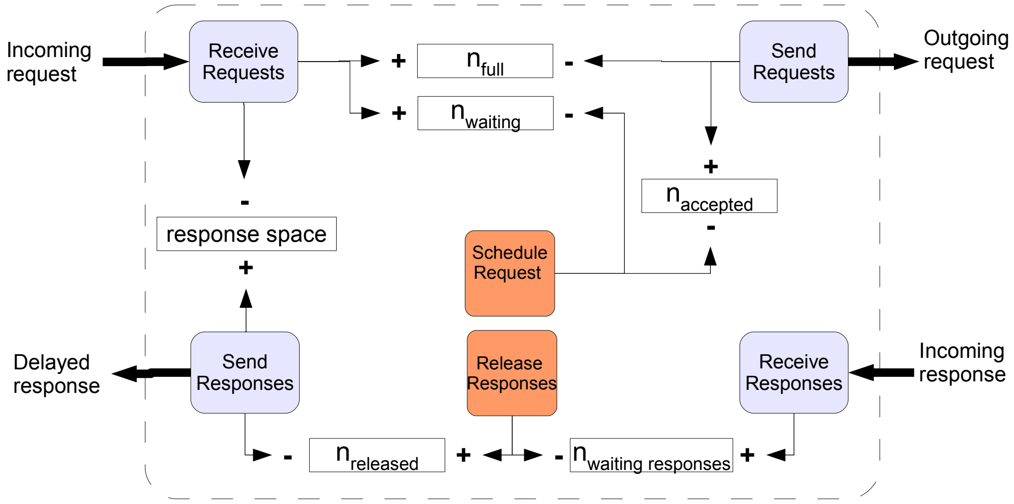


Figure 5.10: Counters in the Delay Block and Their Management

1. **During a scheduling event, $n_{accepted}$ should be greater than zero.** Since a scheduling event takes place at the worst-case scheduling time of a request, the corresponding request must have been scheduled by that time.

2. **During a release event, $n_{waiting\ responses}$ should be greater than zero.** Release event is started at the worst-case finishing time of requests, which determines the release time for responses. Hence, the response must be available in the Delay Block to be released at this time.

5.2.2.6 Flow-Control

The Delay Block uses the counter $n_{requestswaiting}$ to emulate the filling of *request buffer* under worst-case scheduling. This counter is incremented when a request is received by the Delay Block, however, decrementing is done at the worst-case scheduling time of the request, independent of the actual scheduling event (which should be earlier than the worst-case scheduling time).

The flow-control signal in the response path can also create interference unless it is properly handled. If a requestor is allowed to send requests to the resource without accepting the corresponding responses, it can stall the resource and affect the service provided to other requestors. In our approach, as explained in Section 3.2, space is reserved for every response right at validation of the corresponding request. If space cannot be reserved, the request is not validated for service and hence not presented to the resource. This is similar to assuming worst-case filling of the *response buffer*. Therefore, every request that has been scheduled for service is guaranteed that its responses are accepted by the Delay Block as soon as they are available. This ensures that the resource never stalls due to a requestor that does not accept responses when it should.

5.2.3 CCSP Arbiter

As presented in Section 4.3, the scheduling process in the CCSP arbiter is comprised of three major units. The implementation of the units that are responsible for each of the operations are discussed next. We begin by presenting how each parameter of the CCSP is represented.

5.2.3.1 Representation of CCSP Parameters

The *Priority* of the individual requestors can be represented by an integer value. The budget management is implemented based on the credit mechanism proposed in [1]. The budget that each requestor has is represented with an integer value called *credit*. The credit value of a requestor is incremented when its budget is upgraded and decremented when it pays for service. The amount by which the credit value of a requestor is incremented, during upgrade, and decremented, during service, depends on *allocated service rate (ρ')* of the requestor. With the credit mechanism, ρ' is represented as a formal fraction using two integer values, as in Equation 5.2

$$\rho' = \frac{numerator(n)}{denominator(d)}. \tag{5.2}$$

With this representation, the credit of each requestor is incremented by *numerator (n)* every *Service Cycle* and each service costs a credit amount of *denominator (d)*.

The other parameter, *allocated burstiness (σ')* is represented by the initial amount of credit given to each requestor. This leaves the CCSP with four discrete parameters per requestor - *priority*, *numerator*, *denominator* and *initial credit.*

The four parameters of the CCSP arbiter are listed next.

- **Priority (p_i):** The priority of each requestor is represented as an integer value ranging from 0 to *R-1*; where R is the number of requestors to be arbitrated. 0 corresponds to the highest priority value and *R-1* to the lowest.

- **Denominator (d_i):** is an integer value that corresponds to the denominator when the allocated service rate (ρ') is represented as formal fraction. See Equation (5.2).

- **Numerator (n_i):** is again an integer value that corresponds to the numerator when ρ' is represented as formal fraction. See Equation (5.2). The value of n_i ranges from 0 to d_i*-1*.

- **Initial Credit (cr_i):** The amount of credit that each request gets at start up is determined by its *allocated burstiness (σ_i')* and *denominator (d_i)*. The initial credit represents a credit amount that is enough for a requestor to pay for σ_i' successive services. Since d_i is the amount paid for each service, the initial credit value is computed as the product of σ_i' and d_i.

$$cr_i = \lceil \sigma_i' * d_i \rceil \tag{5.3}$$

5.2.3.2 Rate Regulation

Rate regulation is performed by preventing those requestors that do not have enough credit from getting service [1]. Based on the representation of ρ' as in Section 5.2.3.1, a requestor is eligible for service only if it is able to pay a credit amount of d_i up on scheduling. Considering the credit amount of n_i that each requestor gets every service cycle, the credit threshold for eligibility becomes $d_i - n_i$.

An array of flags, *eligibility mask[R]*, is used to indicate eligibility of requestors. An entry of '1' in the array indicates that the corresponding requestor has enough credits to pay and is eligible for service; An entry of '0' indicates otherwise. The pseudocode in Algorithm 5.6 how eligibility of requestors is checked.

Algorithm 5.6 Check Eligibility of Requestors in the CCSP Arbiter

```
for each requestor $r$ do
  if $credit[r] \geq (d_i - n_i)$ then
    eligibility mask[r] ← '1';
  else
    eligibility mask[r] ← '0';
  end if
end for
```

5.2.3.3 Static-Priority Scheduling

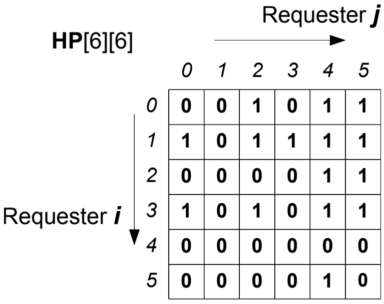
At this stage, one of the eligible requestors is selected based on priority. A requestor is scheduled iff it has the highest priority among all eligible requestors that have valid request. The scheduler uses an $R \times R$ square matrix to indicate, for each requestor, whether there is a request from any other requestor that is eligible and has higher priority.

In a previous implementation [18] of the CCSP arbiter, comparison between priority of requestors was done at every scheduling decision, increasing the complexity of the arbitration process. However, since the priority relationship between requestors does not change after configuration, the computation can be done once during configuration and stored. Then after the scheduler reads priority relationship as a bit value from the matrix and combine it with the *eligibility mask (Algorithm 5.6)* to make scheduling decision.

Given a pair of requestors - $requestor_i$ and $requestor_j$, the cell $HP_{i,j}$ in the matrix is set to '1' iff $requestor_j$ has higher priority than $requestor_i$ and is waiting for service. Figure 5.11 shows an example matrix of priority relationship between six requestors. For example, by looking at the row HP[2], it can be seen that $requestor_4$ and $requestor_5$ have higher priority than $requestor_2$ and both are waiting for service.

The pseudocode in Algorithm 5.7 illustrates how the priority matrix is computed for R requestors.

When a row in the priority matrix is filled with all 0's, it indicates that every other requestor with higher priority either does not have valid request or has run out of credit.


Requester *j*
HP[6][6]
Requester *i*

	0	1	2	3	4	5
0	0	0	1	0	1	1
1	1	0	1	1	1	1
2	0	0	0	0	1	1
3	1	0	1	0	1	1
4	0	0	0	0	0	0
5	0	0	0	0	1	0

Figure 5.11: An Example Priority Matrix for 6 requestors.

Algorithm 5.7 Priority Relationship Among requestors

```
for requestor i in 0 TO R-1 do
  for requestor j in 0 TO R-1 do
    if $priority_j > priority_i$ AND valid request[j]='1' AND eligibility mask[j]='1' then
      HP[i][j] ← '1';
    else
      HP[i][j] ← '0';
    end if
  end for
end for
```

The scheduler, then, makes scheduling decision by combining this *priority matrix* with the *request vector*. For a given requestor to be scheduled, it is required to have a valid request and the row in *HP* that corresponds to it should be filled with all 0's. The output of the scheduler is the *grant vector* that indicates the requestor that has been scheduled for service. It is the requestor that corresponds to an entry in the *grant vector* with value '1', if any, that gets scheduled for service. Algorithm 5.8 shows how scheduling decision is made for a given set of requests. Scheduling is triggered by a signal (*make schedule*) from the service timer in the resource sharing bus. (See Section 4.4.1.)

5.2.3.4 Credit Management

Every service cycle, the credit value of each requestor has to be updated based on the scheduling decision made. Every requestor gains credit amount of n_i and the one that has been scheduled pays credit amount of d_i. To protect low priority requestors from starvation, requestors without valid requests are forbidden from accumulating more than the *initial credit* value [1]. The pseudocode in Algorithm 5.9 describes the credit updating process, which is triggered by *update credit* signal every service cycle.

5.2.4 Resource Sharing Bus

Three major operations are performed in the Resource Sharing Bus.

1. Making a scheduling decision to identify the request that should be served next,

Algorithm 5.8 Static-Priority Scheduling in the CCSP Arbiter

```
GIVEN: Priority Matrix HP[R][R], request vector request[R]
OUTPUT: Scheduling Result grant[R]
done ← '0';
while TRUE do
  if make schedule = '1' then
    for requestors i in TO R-1 do
      if HP[i]= 0 AND request[i]='1' and done='0' then
        grant[i] ← '1';
        done ← '1';
      else
        grant[i] ← '0';
      end if
    end for
    return grant;
    update credit ← '1';
  end if
end while
```

Algorithm 5.9 Credit Management in the CCSP Arbiter

```
GIVEN: Previous credit value of each requestor credit[R],
request vector request[R],
grant vector grant[R]
OUTPUT: Updated Credit of each requestor (credit[R])
if update credit = '1' then
  for requestors i in 0 TO R-1 do
    if grant[i]= '1' then
      credit[i] ← credit[i] + $n_i$ - $d_i$;
    else
      credit[i] ← credit[i] + $n_i$;
      if request[i]= '0' AND credit[i] > initial credit[i] then
        credit[i] ← initial credit[i];
      end if
    end if
  end for
end if
```

2. Sending the scheduled request to the resource, and

3. Returning responses to the correct requestor when they are ready.

The whole operation is started when the *Service Timer* sends a signal to the arbiter requesting new scheduling decision. The *Service Timer* is implemented as a *saturating counter* that counts up to *Service Cycle (See Sec. 4.4.1)*. The counter is reset to zero after every scheduling decision. The pseudocode in Algorithm 5.10 shows the counting mechanism of the timer. Both the *cycle count* and *Service Cycle* are represented as integers. The number of bits required to represent them is determined at design time, based on the minimum time that the resource takes to serve an atom.

Algorithm 5.10 Service Timer in the Resource Sharing Bus

```
while TRUE do
  if cycle count < SERVICE CYCLE then
    cycle count ← cycle count + 1;
  else
    if resource is ready then
      - Schedule new request;
      - cycle count ← 0;
    end if
  end if
end while
```

5.2.4.1 Request Multiplexer

Figure 4.11 shows the major units in the Resource Sharing Bus that are responsible for sending requests to the resource and returning responses to the respective requestor. The *Request Multiplexer* applies scheduling decision made by the arbiter to select one of the requests that are waiting for access to the resource. Whenever the multiplexer receives a valid schedule, i.e. a non-zero *grant vector*, it takes a request from the selected requestor and presents it to the resource. The request multiplexer is also responsible for notifying state of the resource to the timer. Operation of the request multiplexer is based on the FSM in Figure 5.12.

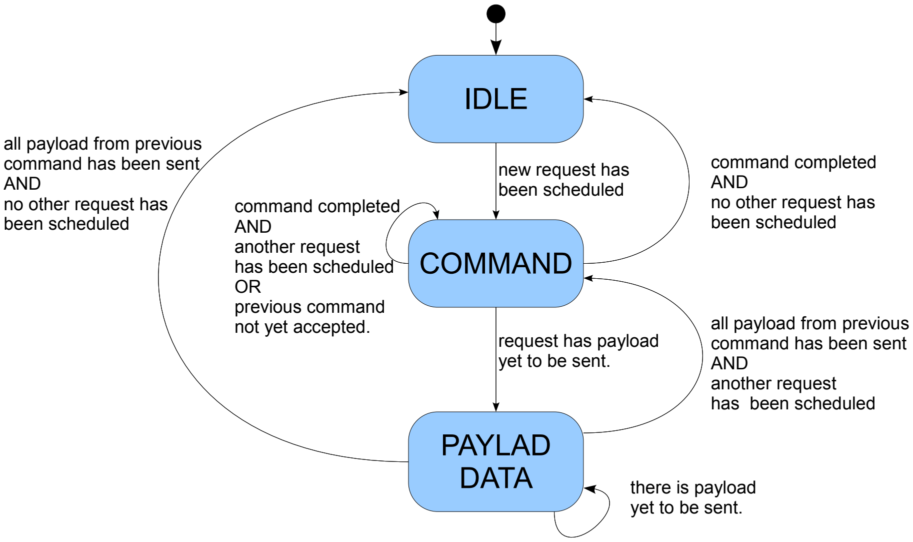

IDLE
COMMAND
PAYLAD DATA
new request has been scheduled
request has payload yet to be sent.
all payload from previous command has been sent AND no other request has been scheduled
command completed AND another request has been scheduled OR previous command not yet accepted.
command completed AND no other request has been scheduled
all payload from previous command has been sent AND another request has been scheduled
there is payload yet to be sent.

Figure 5.12: FSM for multiplexing requests

IDLE state is the initial state of the multiplexer. The FSM enters to the *IDLE* state when the multiplexer has finished sending the previously scheduled request and has not received another valid schedule. While in *IDLE* state, the multiplexer simply waits for a valid scheduling decision to come from the arbiter. With the DTL protocol, which is used in our implementation, a request is comprised of command component and payload component. Thus a request is said to be accepted completely when both the command component and the payload data have been sent to the resource. The multiplexer stays in *COMMAND* state until command part of the request at hand is received by the resource. For every read request sent, the multiplexer records the source requestor in a FIFO (*Scheduling History*) for later use to decide destination of responses. In case of a write request, the multiplexer has to go to *PAYLOAD* state and send the payload data before considering further requests. When the multiplexer finishes sending the request at hand and receives new scheduling decision fromt the arbiter, it progresses to the *COMMAND* state and starts sending.

5.2.4.2 Response Demultiplexer

Upon serving read requests, the resource produces read data as response. The *Response Demultiplexer* unit, then returns the response received to the requestor that made the request. This unit uses entries recorded in the *Scheduling History FIFO* to identify the source requestor. All segments of the response are then sent to the same requestor until a marker indicates the last segment. With the DTL protocol, the last segment of a read response is indicated by the signal *dtl_rd_last*, see Table 5.1. Operation of the response demultiplexer can be described by the pseudocode in Algorithm 5.11.

Algorithm 5.11 Demultiplexing Responses in the Resource Sharing Bus

```
while TRUE do
  if new response is received then
    - Read the entry at the front of the Scheduling History FIFO and determine the
    source requestor, $requestor_i$;
    repeat
      - Receive segment of the response;
      - Send it to $requestor_i$;
    until Last segment is received
  end if
end while
```

5.3 Configuration

As mentioned in Section 4.5, our resource sharing front end is configurable in three phases - *Feature Selection*, *Design Time Configuration* and *Run time Configuration*. Sections 5.3.1, 5.3.2 and 5.3.3 explain the implementation of each, in order.

5.3.1 Feature Selection

Figure 3.4 in Section 3.2 (Proposed Solution) shows the complete front-end with all blocks in place. A Delay Block and an Atomizer are included per requestor in the front-end. Occasionally, the blocks become unnecessary for some of the requestors. For instance, Atomizer is not required for a requestor that always generates unit-sized requests. Similarly, Delay Block is not necessary for a requestor that does not require composable service. Hence, some hardware can be saved by eliminating unnecessary blocks. Another choice that can be made at design time is the arbiter to be included in the *Resource Sharing Bus*. For instance, CCSP Arbiter can be included if tight resource allocation is required.

Figure 5.13 illustrates an example instance of the front-end for four requestors with different requirements. In this example, $Requestor_0$ requires both Atomizer and Delay Block. $Requestor_1$ produces unit-sized requests and hence does not require Atomizer. $Requestor_2$ does not need composable service and hence requires no Delay Block. $Requestor_3$ requires neither the Delay Block nor the Atomizer because it produces unit-sized requests and does not require composable service.

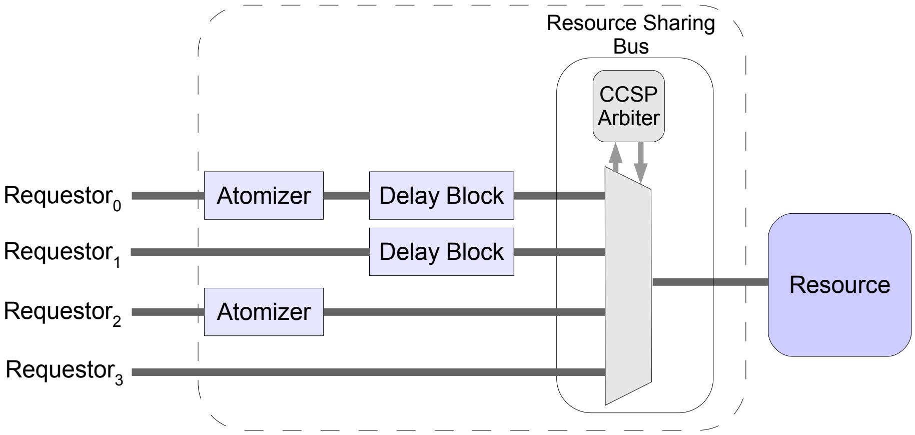


Figure 5.13: An Example Instance of The Front-end

5.3.2 Design Time Configuration

Each functional block in the front-end has generic parameters that makes it customizable. Buffer sizes and width of communication links are among the parameters that can be set during instantiation of the blocks. All blocks have the width of the various DTL signals as configurable at design time. *Data Width* and *Address Width*, which specify the number of bits used for data and address communication, respectively, are among values that are set during instantiation of the front-end. Additional generic parameters that are specific to each block are listed below.

5.3.2.1 Atomizer

The **Size of Chopped Requests** is a generic parameter for the Atomizer. It represents the size into which requests are chopped. The size of the resulting atoms (chopped requests) is chosen based on the resource attached. For a memory resource, for instance, *unit size* corresponds to the access granularity.

5.3.2.2 Delay Block

- **Width of Time Stamps:** This specifies the number of bits used to represent time. As explained in Section 5.1.2, time is represented as count of clock cycles. The number of bits required to represent time is determined by the longest interval expected between arrival of a request and release of the corresponding response. This number is set at design time to a value that is appropriate for each requestor.

- **Depth of Buffers:** The Delay Block has four major buffers that are used to store requests, responses and time stamps. The depth of each of these buffers is computed, based on latency and bandwidth requirements of the requestor, and set at design time. These buffers are *Request Buffer*, *Response Buffer*, *Scheduling Time Buffer* and *Finishing Time Buffer.*

5.3.2.3 Arbiter

The first parameter that is set during the instantiation of the arbiter is the number of requestors to be arbitrated. The arbiter is dimensioned accordingly. The number of bits required to represent the following programmable parameters of CCSP arbiter are also set during design instantiation.

- **Priority:** The number of bits used to represent *priority* depends on the maximum number of requestors that are expected to share the same resource. In a system with R requestors, P=$\log_2 R$ bits are required. In our system a maximum of 16 requestors is envisaged per resource and hence p_i is 4-bits wide.

- **Denominator:** The number of bits required to represent *denominator* depends on the granularity needed in service allocation. If D bits are used to represent *denominator*), the minimum granularity of service allocation becomes $1/2^D$. In our system , a service granularity of $\frac{1}{64}$ has been chosen and hence *denominator* is 6-bits wide.

- **Numerator:** Since the value of *numerator* ranges from 0 to *R-1*, the number of bits required to represent *numerator* and *denominator* is the same.

- **Initial Credit:** As explained in Section 5.2.3.1, *initialcredit* is computed as the product of *allocated burstiness* (σ') and *denominator*. Hence, the number of bits required for its representation is the sum of bits used for the two values. We used 10 bits to represent credits in our design.

5.3.2.4 Resource Sharing Bus

This bus is also dimensioned according to the number of requestors, which is known at design time. In addition, the bus has the following two as design time parameters.

- **Service Cycle:** After knowing the resource, the minimum number of cycles required to serve a unit-sized request is determined at design time.
- **Width of Cycle Count:** The service timer in side the bus counts cycles, for proper timing of scheduling decisions. The number of bits required to represent cycle counts is set at design time based on the value of *Service Cycle*.
- **Arbiter Type:** As shown in the architecture of the front-end, the Resource Sharing Bus contains an arbiter inside. The type of arbiter, to be instantiated is a generic parameter for the bus.

5.3.3 Run Time Configuration

The Delay Block and the Arbiter are the programmable components of the front-end. The configuration values for these blocks are computed based on the service requirements of each of the requestors. Given a resource with net bandwidth of $Bandwidth_{net}$, the allocated service rate for each requestor (ρ'_i) is computed as the fraction of the total bandwidth that is allocated to each one. The burstiness (σ'_i) allocated to each requestor is also needed for the computation.

$$\rho'_i = \frac{Bandwidth_i}{Bandwidth_{net}} \tag{5.4}$$

5.3.3.1 Computation of Configuration Parameters

Arbiter

The configuration values for the arbiter are computed based on the service allocated to each requestor. For CCSP, four parameters (*priority, numerator, denominator* and *initial credit*) are computed using Equations (5.5) and (5.6) and (5.7).

$$\frac{numerator_i}{denominator_i} = \rho'_i \tag{5.5}$$

$$initial\ credit(cr_i) = \sigma'_i * denominator_i \tag{5.6}$$

$$priority(p_i) = priority\ of\ requestor\ r_i \tag{5.7}$$

Delay Block The value of service latency (Θ) for each requestor is computed based on the priority, allocated service rate and allocated burstiness of requestors. With CCSP arbiter, service latency (Θ) is computed for each requestor according to Equation (5.8), as presented in [2];

$$\Theta_i^{ccsp} = \left\lceil \frac{\sum_{\forall r_j \in R_{r_i}^+} \rho'_j}{1 - \sum_{\forall r_j \in R_{r_i}^+} \rho'_j} \right\rceil \tag{5.8}$$

$R_{r_i}^+$ denotes the set of requestors with higher priority than requestor r_i. The completion latency (λ) is computed simply as the reciprocal of the allocated rate (ρ') for each requestor. In fact, as mentioned in Section 5.2.2.1, λ is a rational number and has integer part (λ_I) as well as fractional part (λ_n/λ_d).

$$\lambda'_i = \frac{1}{\rho'_i} \tag{5.9}$$

From equations (5.8) and (5.9), the four programmable parameters of the Delay Block are obtained.

5.3.3.2 Address Mapping

The configuration infrastructure presented in Section 4.5.3.1 is used to send configuration values to the right unit. After configuration data reaches a programmable unit, the value has to be assigned to the correct parameter. The intended parameter is identified based on the configuration address. The address mapping used to identify parameters in the Delay Block and the CCSP is presented next.

Delay Block

The four parameters, Θ, λ_I, λ_n and λ_d, are sent to each Delay Block one by one. Two bits in the configuration address (3^{rd} and 4^{th} bits) are used to identify which parameter the value corresponds to. The address mapping for configuration of the Delay Block is shown in Figure 5.14.

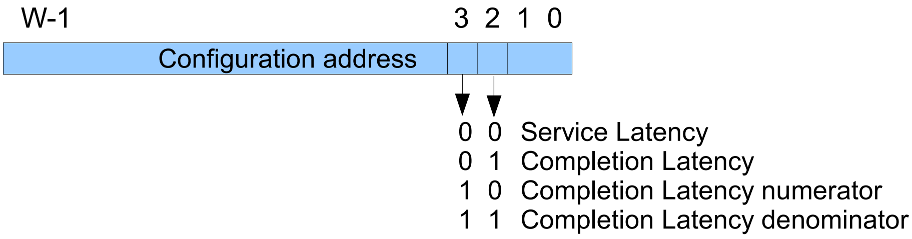


Figure 5.14: Address Mapping for Configuration of the Delay Block

5.3.3.3 CCSP Arbiter

The CCSP arbiter has four programmable parameters for each requestor that is arbitrated. Unlike the Delay Block, however, all four parameters that belong to the same requestor are packed in a single word and received at once. Thus, when the CCSP receives a configuration data, it checks the configuration address to identify which requestor the values are intended for. In our system, a maximum of sixteen requestors is envisaged per resource. Therefore four bits in the configuration address are required to choose among requestors. *Bit-7* through *bit-4* are used here.

After identifying the intended requestor, the value of each parameter is extracted from the configuration data received. The first four bits represent priority value for the selected requestor. The remaining bits are allocated as 10-bits for *initial credit*, 6-bits for *numerator* and 6-bits for *denominator*. Figure 5.15 shows how the four parameters are extracted from the configuration data received.

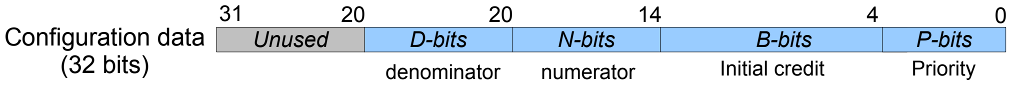


Figure 5.15: Mapping of Configuration Data to CCSP Parameters

The pseudocode in Algorithm 5.12 describes how the CCSP parameters are extracted for a requestor.

Algorithm 5.12 Configuration of CCSP Parameters

```
GIVEN: Configuration Data (Data[]) and Configuration Address (Address[])
while TRUE do
  if configuration command is received then
    $r \leftarrow f(Address(7\ downto\ 4))$; //identify the intended requestor
    . $Priority(r) \leftarrow Data(3\ downto\ 0)$;
    . $numerator(r) \leftarrow Data(13\ downto\ 4)$;
    . $denominator(r) \leftarrow Data(19\ downto\ 14)$;
    . $initial\ credit(r) \leftarrow Data(25\ downto\ 20)$;
  end if
end while
```

5.4 Design Flow and Automation

To automate, instantiation of the front-end, our design, is integrated into the *Æthereal Design Flow, which is used at NXP Semiconductors.* The design flow is presented in depth in [5]. With the *Æthereal Design Flow*, the topology and communication of IPs is specified using two *xml* files : *architecture.xml* and *communication.xml.*

5.4.1 Architecture Specification

In architecture.xml, all IPs in the architecture are listed. For each IP, name, type and ports are specified. The name of each IP has to be unique within the architecture. The type, for instance *Memory*, indicates what type of IP to be instantiated by the design flow. The individual ports also have a unique name within the IP. Each port can be either an *initiator* or a *target* and has additional parameters such as address range and protocol.

Our front-end is attached to the target port of an IP, which represents the shared resource. Hence, whether the resource sharing front-end is to be instantiated for a target port or not is specified in the architecture specification. When the parameter *delay* is set to ”1” in the specification of an IP target port, the front-end is instantiated for that port of the shared resource. The type of arbiter to be used in the front-end is also specified as a parameter to the target port of the shared resource. The following lines from *architecture.xml*, for instance, indicate that a front-end should be instantiated to the target port.

```
...
  <ip id="IP_B" type="Memory" >
    < port id="port_1" type="target" protocol="MMIO_DTL" >
        <parameter ... other parameters ... >
         <parameter id="delay" type="string" value="1">
         <parameter id="arbiter" type="string" value="CCSP">
    </port>
  </ip>
```

The complete architecture specification for an example system is shown in Appendix A.1.

5.4.2 Specification of Communication Requirements

After specifying the architecture, the communication between IPs is specified in the other *xml* file : *communication.xml.* In the communication specification, connections between ports of communicating IPs are specified. Each connection has an initiator and a target port, which represent its end points. A connection can be used for write and/or read operation for which bandwidth and latency requirements are specified.

If, in the architecture specification, a target port of an IP is specified with the parameter *delay* set to ”1”, then all connections that terminate at that port will be connected through the resource sharing front-end. As mentioned in Section 4.5.1.2, however, the

Delay Block in the front-end is activated only if composability is required for that connection (requestor). During instantiation, the Delay Block is activated by setting the parameter *delay* to ”1” for all connections that need composability. The following lines, taken from *communication.xml*, specify a connection from $port_0$ of IP_A to $port_1$ of IP_B. Since the parameter *delay* is set to ”1” for the connection, the Delay Block on that path is activated.

```
...
  <connection id=0 >
    <initiator ip='$IP_A$' port=$port_0$ >
    <target ip='$IP_B$' port=$port_1$ >
    <read bw=100 latency=500 >
    <parameter id=”delay” type=”string” value=”1”>
  </port>
```

The complete *xml* file, where communication requirements of an example system are specified, is found in Appendix A.2.

Experiments and Results 6

As pointed out in Section 1.4, the major requirement for this front-end is to bring about composable resource sharing between requestors when it is placed in front of a predictable resource. To verify this requirement, two test platforms have been devised - one for simulation and another one for FPGA. In Section 6.1 the testbench used for simulation and the test results are presented. In Section 6.2, the test on FPGA is discussed.

6.1 Simulation

6.1.1 Testbench

The system in Figure 6.1 was used to test composability in resource sharing using the front-end designed. The whole system is 100The system contains four traffic generators (*Core 0*, *Core 1*, *Core 2* and *Core 3*) that share a resource. There is additional traffic generator, *Configurator*, that is responsible for configuring the entire front-end. The Configurator is connected to each programmable component of the front-end, namely, Delay Blocks and the CCSP arbiter via a configuration bus. The shared resource is an SRAM with 32 bits data width that offers a net bandwidth of 800 MB/s.

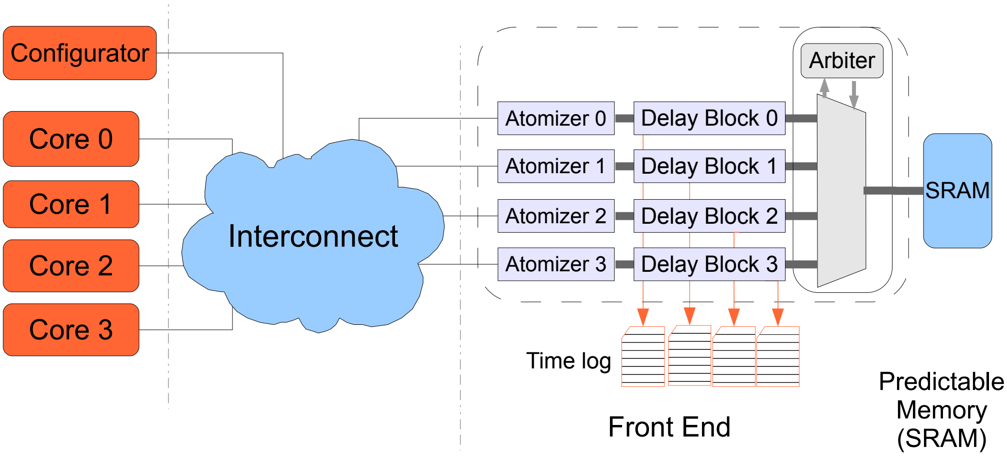


Figure 6.1: Testbench For The Resource Sharing Front-end

Before any operation, the *Configurator* sends configuration data to each block to set all programmable parameters. After configuring the entire front-end, the cores start sending requests. The traffic generated by each core can be adjusted to the required data rate and request size.

To help in analyzing the timing behavior of transactions, a process in the Delay Block writes relevant time stamps into a text file for offline analysis. The five time stamps mentioned in Section 4.2.2, are recorded for every request. These are *Request Arrival Time (t_a), Actual Scheduling Time (t_s), Worst-case Scheduling Time (t_{SW}), Actual Finishing Time (t_f)* and *Worst-case Finishing Time (t_{FW})*. The log created for each requestor looks like the one in Table 6.1. All time stamps are in clock cycles. According to the log, the arrival of the sixth request *(k=6)* to the Delay Block is validated at *cycle 525*. This represents the time when the request has been fully buffered in the Delay Block and free space is reserved for the corresponding responses. The request is then accepted by the resource at *cycle 639*, however, flow-control information is generated based on the worst-case scheduling time (*cycle 653*). Similarly, the reource finishes serving the request at *cycle 652* and responses will be available at that time. However, the Delay Block withholds them until *cycle 716*, its worst-case finishing time, is due.

Table 6.1: Time Stamp for Requests From Core 0

Request ID (k)	t_a^k	t_s^k	t_{SW}^k	t_f^k	t_{FW}^k
...					
4	523	524	529	527	590
5	524	588	590	589	653
6	525	639	653	652	716
7	526	712	716	715	779
...					

The test procedure is started with a given set of requestors and the service requirement for each described in terms of (*Bandwidth* and *latency*). Based on the requirements, the values of all programmable parameters in the front-end are computed. These are four programmable parameters (*Service Latency (Θ_i)* and *Completion Latency* $(\lambda_I, \lambda_n, \lambda_d)_i$) for each Delay Block and four parameters (*priority (p_i)*, *numerator (n_i)*, *denominator (d_i)* and *initial credit (cr_i)*), per requestor, for the CCSP Arbiter. See Section 5.2.2.1 and 5.2.3.1 for description of the parameters. The configurator then sends these parameters to each of the programmable blocks. After configuring the front-end the cores are turned on and they start sending requests. Time log is recorded for each of the requestors for offline analysis.

6.1.2 Use Case

We considered a use case comprised of four requestors - r_0, r_1, r_2 and r_3 that correspond to *Core 0*, *Core 1*, *Core 2* and *Core 3* in Figure 6.1. The service requirement for each requestor is shown in Table 6.2.

The service requirements for each of the requestors, expressed in terms of resource accesses, and the corresponding requestor parameters are shown in Table 6.3. Equations (5.4), (5.8) and (5.9), presented in [1], were used for the computation. Since we are using atomizers, all requests are of unit size when they reach the arbiter hence the value of *burstiness (σ')* will be 1 for all requestors. Regarding priority, all requestors are assigned priority in descending order - the highest priority (*priority=0*) for *Core 0* and lowest priority *(priority=3)* for *Core 3.*

Table 6.2: Service requirements of the four requestors in the use case

Requestor	Read/ Write	Bandwidth (MB/s)	Request Size (Bytes)	latency
r_0	Read	1	32	0
r_1	Read	100	4	0
r_2	Read	200	8	0
r_3	Write	40	4	0

Table 6.3: Service requirements of the four requestors and configuration parameters

Requestor	priority (p_i)	σ'	ρ'	Θ	$1/rho'$
r_0	0	1	0.00125	4	63
r_1	1	1	0.125	5	8
r_2	2	1	0.25	6	4
r_3	3	1	0.05	8	20

The corresponding discrete parameters to be used for configuring each of the Delay Blocks and the CCSP Arbiter are given in Table 6.4 Each of these parameters are sent to the respective blocks by the configurator and then the cores start sending requests.

Table 6.4: Configuration Values for the Front-end

Requestor	**Delay Block**		**CCSP Arbiter Parameters**			
	Θ_i	λ_i	priority (p_i)	numerator (n_i)	denominator (d_i)	Initial Credit (cr_i)
r_0	4	63	0	1	63	63
r_1	5	8	1	7	56	56
r_2	6	4	2	15	60	60
r_3	8	20	3	3	60	60

6.1.3 Predictability Test

With our approach, predictability of the resource is a pre-requisite to bring about composable resource sharing. Hence, predictability test has been carried out to verify that the service bounds always hold. As mentioned in Section 3.1, service offered to a requestor is said to be predictable iff the service latency can be upper bounded and the service rate can be lower bounded by finite values. The time stamps t_s and t_f reflect the actual service that is provided to a requestor. The time stamps t_{SW} and t_{FW}, on the other hand, are based on the service bounds (Θ and ρ') and hence reflect the worst-case service provided to a requestor.

With a predictable service, every request is expected to be scheduled before its worst-case scheduling time and responses, if any, are expected to be ready before the worst-case finishing time of the corresponding request. In this test, the values of time stamps *arrival time (t_a)*, *actual scheduling time (t_s)*, *worst-case scheduling time (t_{SW})*, *actual finishing time (t_f)* and *worst-case finishing time (t_{FW})* are logged for each requestor. According to the definition of a predictable resource, requests should be scheduled before thier worst-case scheduling time and should be finished before their worst-case finishing time. Hence, for every reqest, the time stamps t_s and t_f are expected to be earlier than t_{SW} and t_{FW}, respectively.

The plot in Figure 6.2, shows the timing of events for the first 20 requests from requestor r_2. From the plot, it can be seen that the curve for t_{SW} always lies above

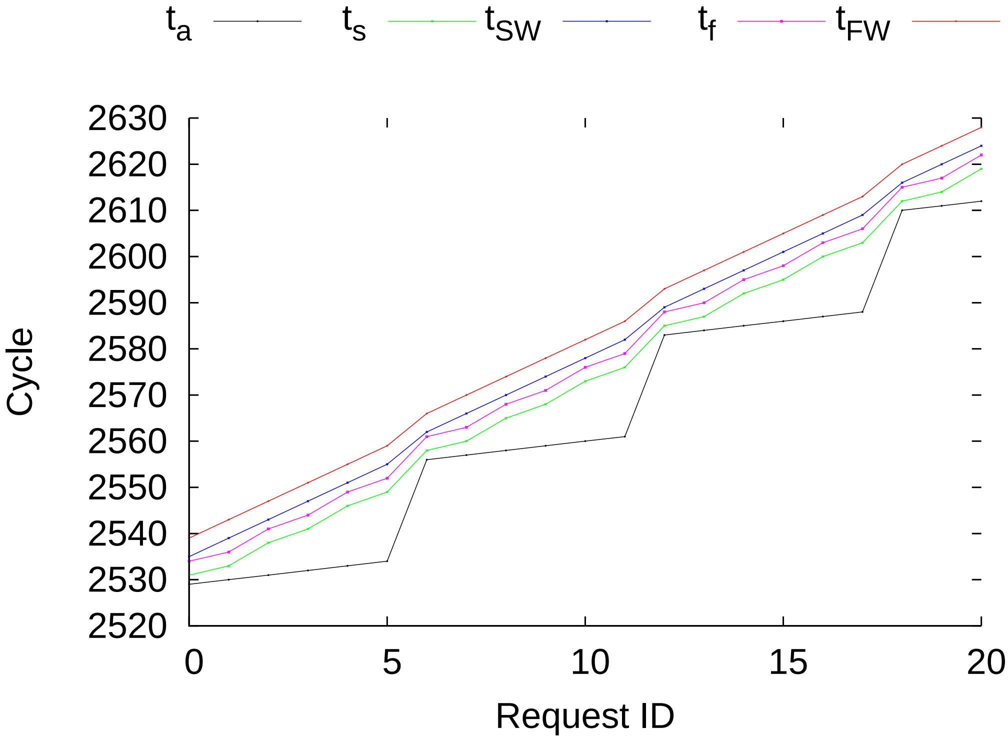


Figure 6.2: Timing of Events

the curve for t_s and the curve for t_{FW} always lies above the corresponding curve for t_f. Similar results have been obtained for all the remaining requestor. Hence, it can be concluded that the service provided to each requestor is bounded by the worst-case amounts computed based on Θ and λ.

6.1.4 Composability Test

As mentioned in Section 3.1, our approach to bring about composability is eliminating the dependence, on other requestors, of the time when responses and flow-control signals are released. With shared resource, the actual values of scheduling and finishing times depend on the requests made by others. With our approach, we disregard the actual timing of these events and consider the worst-case values to release responses and flow-control signals. More precisely, flow-control signals about acceptance of requests are generated based on the worst-case scheduling time (t^k_{SW}) and responses are released at the worst-case finishing time (t^k_{FW}). By doing so, requestors are always made to observe the same latency and rate in service, which is not affected by the actual behavior of other requestors.

6.1.4.1 Test Procedure

The test for composability is done by checking whether the service that a requestor obtains is affected by traffic from other requestors. With this test, we compare the service that a requestor gets when it is activated alone with what it gets when other requestors are also added. In the test, we measure the end-to-end latency, throughput, status of buffers in the front-end and the aforementioned time stamps to quantify the service that each requestor obtains.

i. **End-to-end latency:** The time between issuing a request and getting the corresponding response.

ii. **Throughput:** The actual data rate at which a requestor writes to or reads from the resource.

iii. **Status of buffers in the front-end:** The flow-control signal that determines whether new requests can be accepted from a requestor is generated based on the number of pending requests in the front-end (in the *Request Buffer* of the Delay Block) and availablity of space in the response buffer. The value of the two counters ($n_{requests\ waiting}$ and $n_{response\ space}$) by the time each request reaches the Delay Block is recorded for each requestor.

iv. **Time stamps:** For every request, the Delay Block records *Arrival Time (t_a)*, *Actual Scheduling Time (t_s)*, *Worst-case Scheduling Time (t_{SW})*, *Actual Finishing Time (t_f)* and *Worst-case Finishing Time (t_{FW})*. These values show the timing of events for each request in detail.

For each requestor, the measurements were made under the following two situations.

1. **Isolated:** The requestor is activated alone to use the resource. All other requests that share the resource are turned off. The measurements obtained here indicate the service that each requestor obtains without any interference from other requestors.

2. **Composed:** All requestors are activated together and they all make requests to the resource. The measurements obtained under this situation indicate the service provided to each requestor when it is sharing the resource with others.

By comparing measurements obtained under the above two situations, it can be checked whether the service that a requestor gets is affected by the activity of other requestors that share the same resource. The test results are presented next for each requestor followed by conclusion from each result.

6.1.4.2 Test Result - Average Throughput and End-to-End Latency

Table 6.5 shows the throughput and the end-to-end latency of each requestor under the two situations - *isolated* and *composed*.

Table 6.5: Comparison of Service offered to requestors - *Isolated Vs. Composed*

Requestor	**Throughput (MB/s)**		**End-to-End Latency (Cycles)**	
	Isolated	*Composed*	*Isolated*	*Composed*
r_0	1	1	565	565
r_1	98.7	98.7	185	185
r_2	197.4	197.4	184	184
r_3	37.9	37.9	23	23

Conclusion: The result of the bandwidth test, which is shown in Table 6.5, indicates that the net bandwidth offered to each requestor is the same under the two situations. Similarly, end-to-end latency of a requestor is not affected by activity of other requestors. This implies each requestor obtains the same service regardless of the traffic from other requestors and hence the resource sharing is composable.

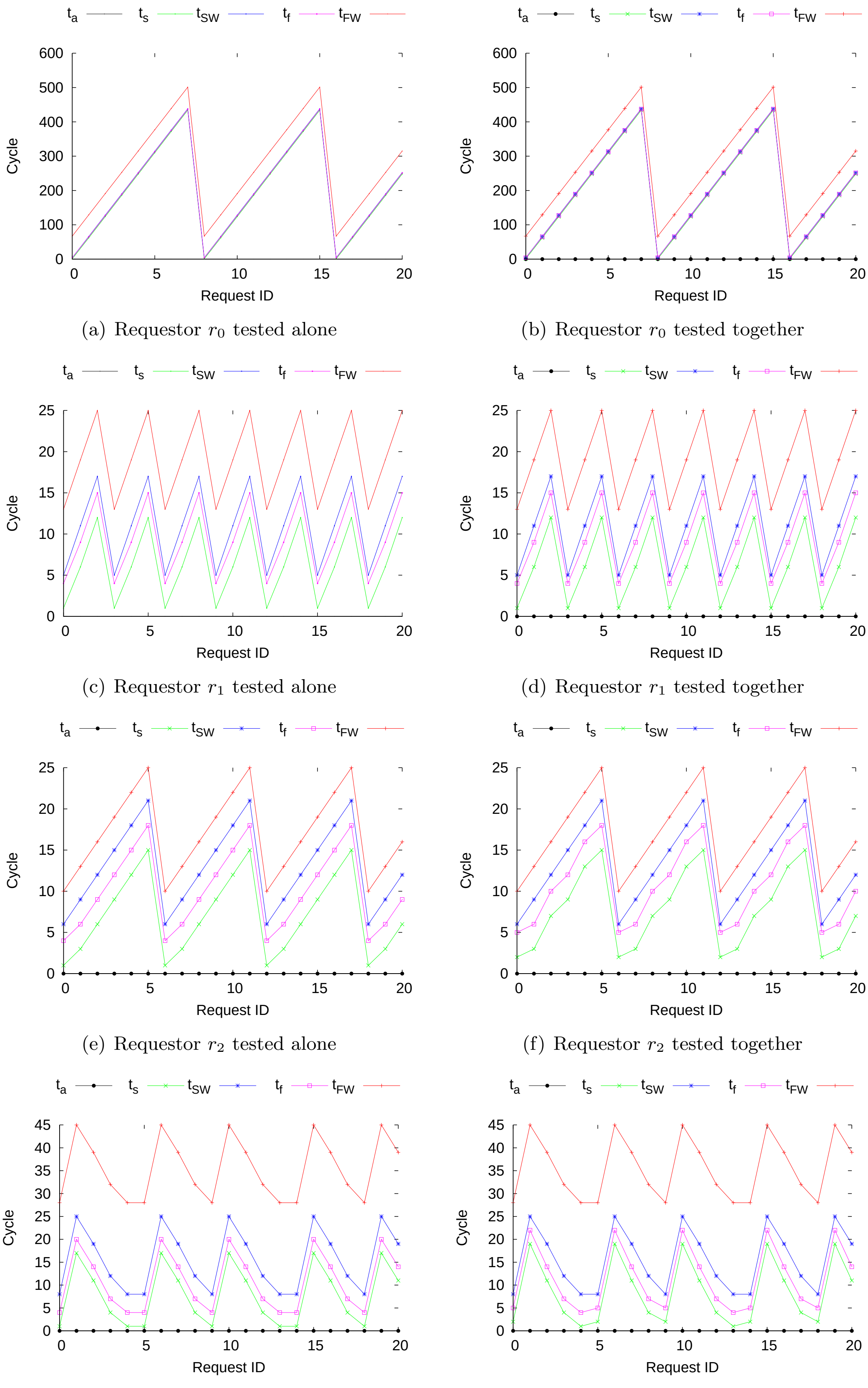
6.1.4.3 *Test Result - Detailed Timing of Events*

Interference to a requestor is reflected on the time at which its requests are accepted by the resource and the time it takes to get back the responses. When a requestor is not shielded from interference, it observes fluctuation in the timing of the two events (*release of responses* and *flow-control signals*) as other requestors change their behavior. If the resource sharing is to be composable, the two time stamps (t^k_{SW} and t^k_{FW}) of every requestor should be indifferent of the traffic from other requestors. The test here shows which of the time stamps recorded for a request are affected by interference.

Figure 6.3(e), for instance, shows the timing of events for the first 20 requests from requestor r_2, when it is the only requestor sending requests to the shared resource. For each request, the time stamps t_a, t_s, t_f, t_{SW} and t_{FW} are plotted. To make the plot easier to visualize, t^k_a is taken as reference for the other events. Hence, the curve for t^k_a overlaps with the horizontal axis and the remaining time stamps represent the number of cycles after t^k_a. Figure 6.3(f) shows the timing of events for the first 20 requests from the same requestor, but this time composed with the other three requestors. In the same manner, the remaining pairs of figures (Figures 6.3(a) and 6.3(b)), (Figures 6.3(c) and 6.3(d)), and (Figures 6.3(g) and 6.3(h)) show the timing of events for the respective requestors when tested in isolation and when composed with others, respectively.

Conclusion:

When we compare the pair of time stamps for a requestor under the two situation, we observe that some of the time stamps change due to addition of other requestors. The difference between the pair of graphs is not obvious for the high priority requestors (r_0 and r_1), but it is fairly visible for the low priority requestors (r_2 and r_3). This is due to



(a) Requestor r_0 tested alone

(b) Requestor r_0 tested together

(c) Requestor r_1 tested alone

(d) Requestor r_1 tested together

(e) Requestor r_2 tested alone

(f) Requestor r_2 tested together

(g) Requestor r_3 tested alone

(h) Requestor r_2 tested together

Figure 6.3: Timing of Events for Each Requestor

the simple fact that low priority requestors are affected most while sharing resources. In Figure 6.4, the difference between the time stamps under the two situations is plotted for requestor r_3. For instance, t_s represents the scheduling times of requests from requestor

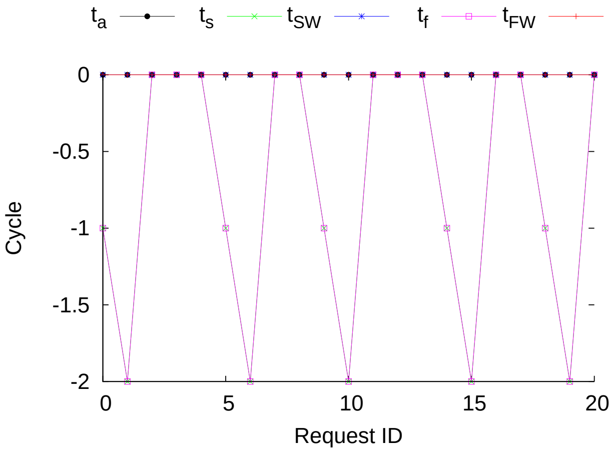


Figure 6.4: Comparison

r_3 when tested alone minus what they will have when other requestors are added. It can be seen from Figure 6.4 that the difference is *zero* for all time stamps except *actual scheduling time* and *actual finishinging time.* By comparing the pair of plots for each requestor, the following observations are made

- When other requestors start sharing the resource, the actual scheduling time (t_s) and actual finishing time (t_f) values of a requestor are affected.
- The time stamps t_{SW}, which is used to generate flow-control signals, and t_{FW}, which determines the time when responses are released remain the same irrespective of whether other requestors are activated.

In the plots above, only the time stamps for the first twenty requestors are shown. But the comparison has been made for all requests generated in the test and the results are the same.

As explained in Section 3.1, interference from other requestors, if any, is manifested by variation in timing of flow-control signals and responses. Since the time stamps t_{SW} and t_{FW} are the same under the two situations, we can conclude that the latency and rate of service that each requestor gets is not affected by the behavior of other requestors.

6.1.4.4 *Test Result - Flow-Control*

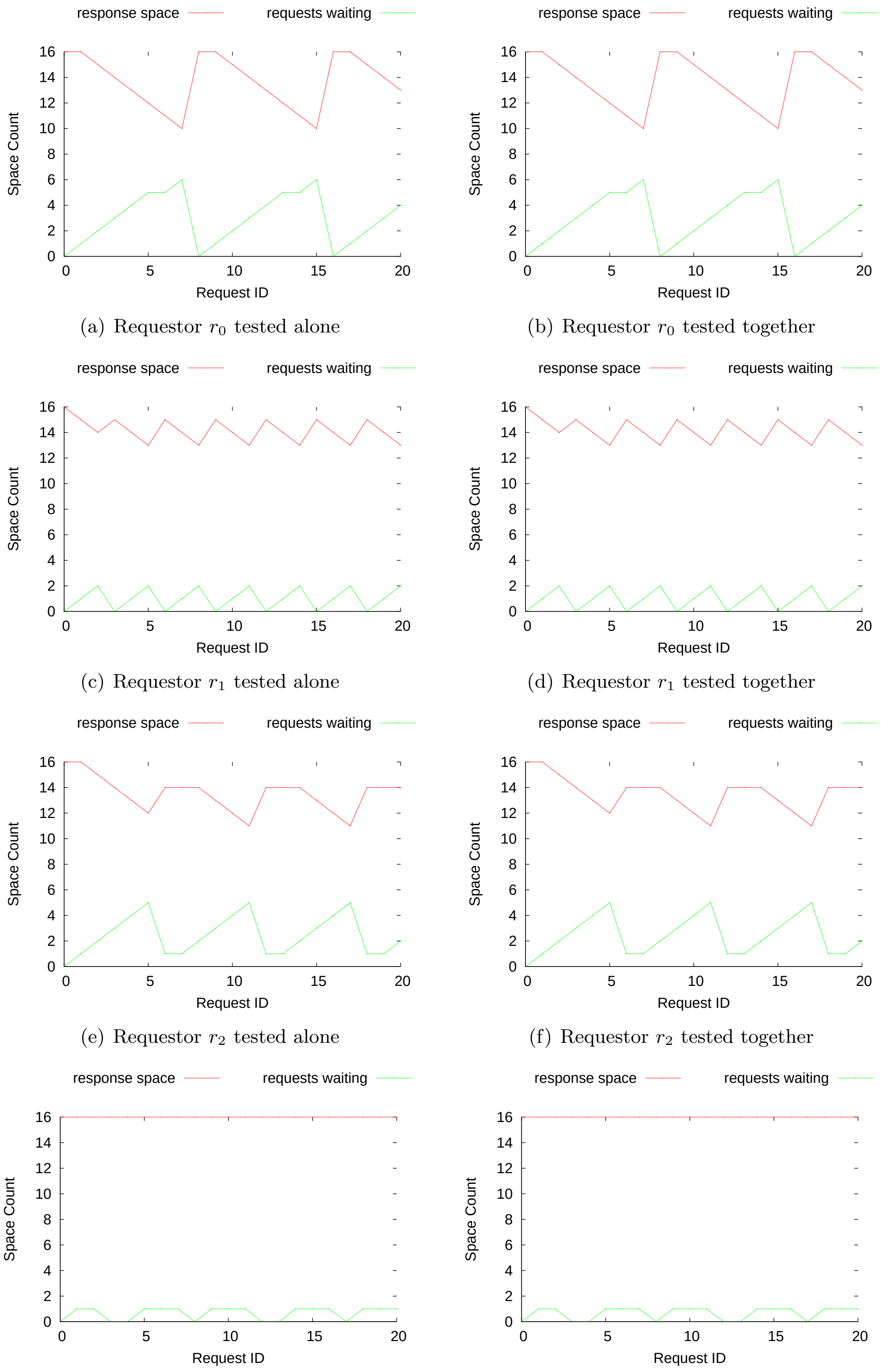
Another check to be made while testing composability is on the timing of flow-control signals. Flow-control signals indicate whether a request can enter the front-end and advance to the resource. As mentioned in Section 5.2.2.6, the Delay Block generates flow-control signal to *accept* requests only if there is space left in the request buffer for additional requests and space can be reserved in the response buffer for its responses. Thus, the timing of flow-control signals can be derived from the value of the counters

$n_{requests\ waiting}$ and $n_{response\ space}$. To see if traffic from other requestors affects flow-control to each requestor, we compare the values of the two counters ($n_{requests\ waiting}$ and $n_{response\ space}$) under the two situations.

Figure 6.5(a) indicates the status of the two buffers (*Request Buffer* and *Response Buffer*) when requestor r_0 is tested in isolation and Figure 6.5(b) indicates the status when all requestors are activated. In a similar manner, the remaining pairs of plots (Figures 6.5(c) and 6.5(d), 6.5(e)and 6.5(f), 6.5(g) and 6.5(h) correspond to requestors r_1, r_2 and r_3, respectively.

Conclusion:

For each requestor, the filling of buffers does not change with traffic from other requestors. Hence, each requestor observes the same flow-control whether or not other requestors are added.



(a) Requestor r_0 tested alone

(b) Requestor r_0 tested together

(c) Requestor r_1 tested alone

(d) Requestor r_1 tested together

(e) Requestor r_2 tested alone

(f) Requestor r_2 tested together

(g) Requestor r_3 tested alone

(h) Requestor r_3 tested together

Figure 6.5: Buffer State of Requestors

6.1.5 Impact on Performance

Since the front-end delays every response to the worst-case finishing time, it is expected to increase the average end-to-end latency. Moreover, the front-end has four pipeline stages in the request path and two pipeline stages in the response path which add to the end-to-end latency. As can be seen in Table 6.6, all requestors have longer end-to-end latency with our front-end in comparison with what they would have without it.

Its effect on throughput, however, can be eliminated by including larger buffers for requests and responses in the Delay Block. Table 6.6 shows the average end-to-end latency for each requestor both with and without the front-end. In this experiment, both the request and the response buffers in the Delay Block have depth of 16 and it can be seen that the throughput of requestors is not affected.

Table 6.6: Impact of the Resource Sharing Front-end on Performance

Requestor	**Without Front-end**		**With Front-end**	
	Throughput (MB/s)	*Average End-to-End Latency (Cycles)*	*Throughput (MB/s)*	*Average End-to-End Latency (Cycles)*
r_0	1	115	1	565
r_1	98.71	136	98.72	185
r_2	197.44	123	197.43	184
r_3	37.90	19	37.90	23

6.2 Test on FPGA

The system used for test on FPGA contains three *microblaze* [21] processors and on-chip memory interconnected with Æthereal Network-on-Chip. The system is shown in Figure 6.6. The first processor, *MB host* is responsible for configuring the network as well as our front-end.

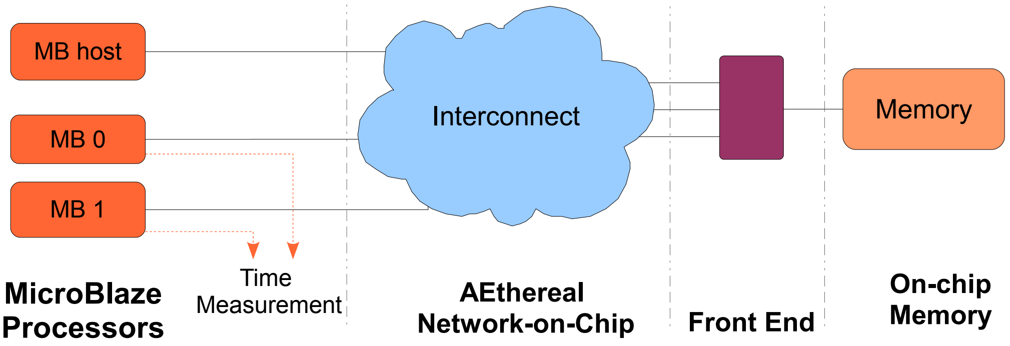


Figure 6.6: System on FPGA To Test The Resource Sharing Front-end

The on-chip memory, which is used as shared resource in our test, is a single cycle memory with 32 bit data width. For the test, the FPGA was run at 50MHz. We considered the usecase in Table 6.7 with three requestors.

Table 6.7: Service requirements of the three requestors

Requestor	Request Size (bytes)	Bandwidth (MB/s)	Latency
r_0	4	2	0
r_1	64	16	0
r_2	128	16	0

The service requirements, when expressed in resource accesses, are as shown in Table 6.8. The corresponding discrete parameters to be used for configuring each of the Delay Blocks and the CCSP Arbiter are given in Table 6.9

Table 6.8: Service requirements of the three requestors in resource accesses

Requestor	Request Size (bytes)	priority (p_i)	σ'	ρ'	Θ	$1/rho'$
r_0	4	0	1	0.03125	4	32
r_1	64	1	1	0.25	5	4
r_2	128	2	1	0.25	7	4

Result of Composability Test

The test carried out on FPGA was measuring the end-to-end latency of requestor r_2 in reading 128 byte data from the on-chip memory. The measurement was made twice first with the requestor r_1 turned off, i.e. generating no traffic and the second time with

Table 6.9: Configuration Values for the Front-end

Requestor	**Delay Block**		**CCSP Arbiter Parameters**			
	Θ_i	λ_i	priority (p_i)	numerator (n_i)	denominator (d_i)	Initial Credit (cr_i)
r_0	4	63	0	1	63	63
r_1	5	4	1	15	60	60
r_2	7	4	2	15	60	60

r_1 turned on. In both cases the end-to-end latency for requests from r_2 was found to be 544 cycles showing that requestor r_2 is shielded by the resource sharing front end from interference. The measurement was repeated eight times each time reading 128 Byte data and the same result was obtained. From the result, conclusion has been made that front-end has resulted in a composable resource sharing.

Synthesis Results 7

To estimate hardware cost and operating frequency, the resource sharing front-end has been synthesized with different settings. The settings are based on the number of requestors sharing the resource, the depth of request and response buffers in the Delay Block and the number of bits used to represent fractions in the CCSP arbiter as well as in the Delay Block. The design has been synthesized both for FPGA and ASIC. In Section 7.1, the results of FPGA synthesis are presented followed by the results of ASIC synthesis in Section 7.2.

7.1 Synthesis FPGA

The FPGA synthesis was done for *Virtex4, XC4VLX160* Device from Xilinix with speed grade -10. The result on device utilization and operating frequency for each block in the front-end is presented next.

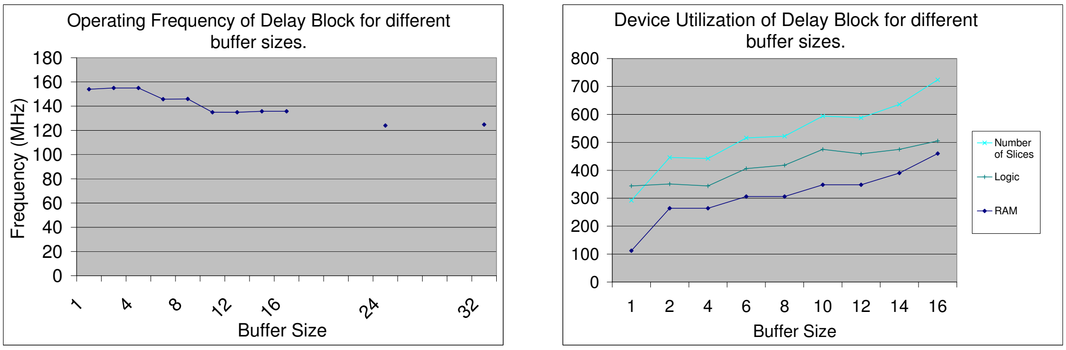
7.1.1 Atomizer

Since an atomizer is instantiated per requestor, the size and operating frequency of instances do not depend on the number of requestors. The basic Atomizer without any buffering for requests runs at *210 MHz* and consumes only *87 slices* on the FPGA.

7.1.2 Delay Block

Delay Block is also instantiated per requestor. Hence the number of requestors that share the resource does not affect synthesis results for the Delay Block. The size of the Delay Block is dominated by buffers for requests and responses. Synthesis results for the Delay Block with different buffer depths are plotted in Figure 7.1. A Delay Block with all buffers set to a depth of 1 consumes *292 slices* and can operate at *160 MHz.* As the depth of the buffers is increased, however, the device utilization increases. The frequency also drops slightly at distinct points, namely as the depth changes from 4 to 6 and from 8 to 10. Thus, the number of bits before the transitions, such as 4 and 8 are good choices for buffer sizes.

The speed of the Delay Block is determined by the critical path shown in Figure 7.2. It is due to the computation of the worst-case scheduling time stamp (t^k_{SW}) for requests. Since the credit approximation mechanism, described in Algorithm 5.3, is used in the computation of time stamps, the number of bits used to represent λ_n and λ_d determines the operating frequency of the Delay Block. The operating frequency and device utilization of the Delay Block for different precisions are plotted in Figures 7.3(a) and 7.3(b), respectively.



(a) Operating Frequency

(b) Device Utilization (Number of Slices)

Figure 7.1: Synthesis Result for The Delay Block - for different buffer sizes

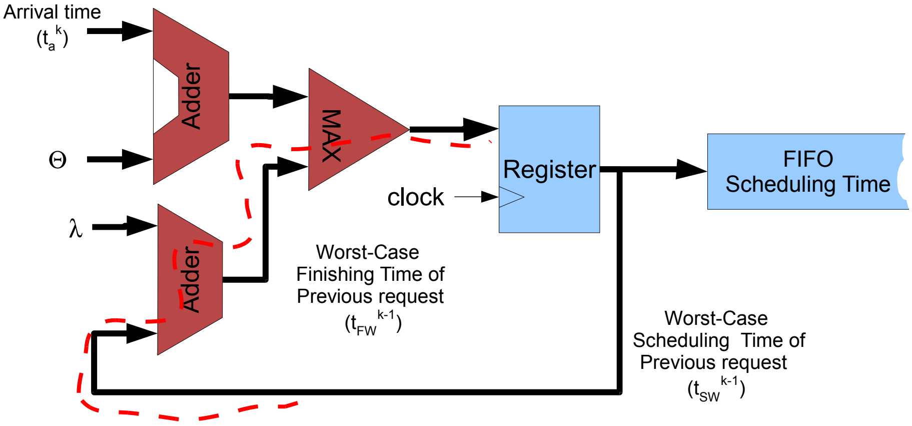
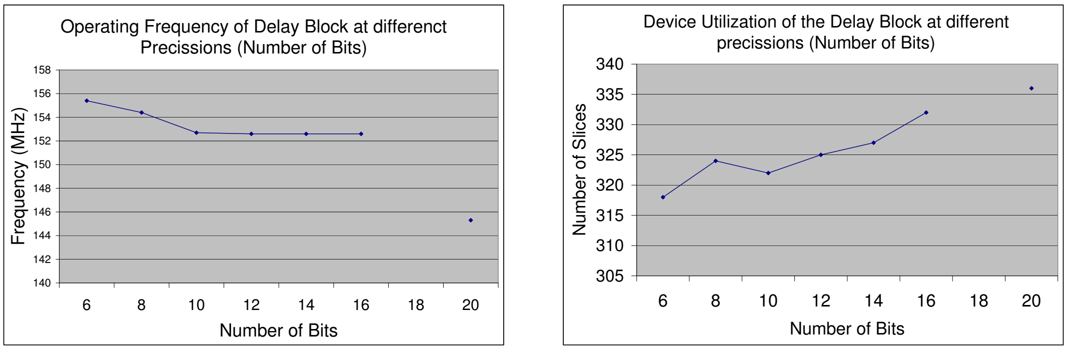


Figure 7.2: The Critical Path in the Delay Block



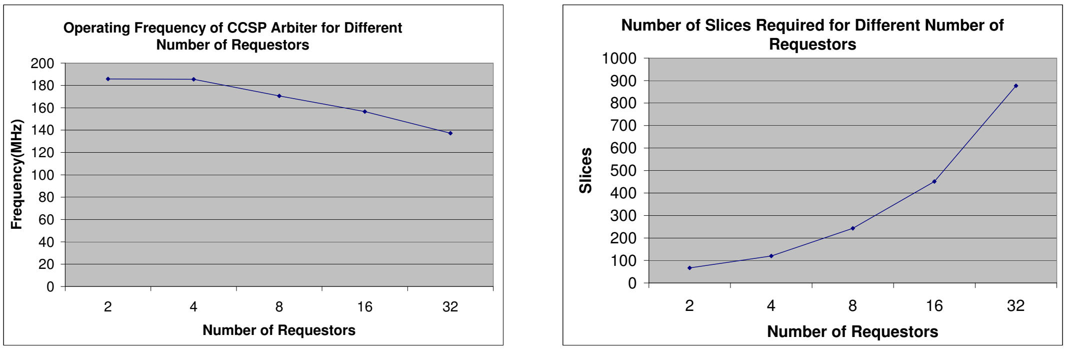
(a) Operating Frequency

(b) Device Utilization (Number of Slices)

Figure 7.3: Synthesis Result for The Delay Block - for Different Precisions

7.1.3 CCSP Arbiter

Since the CCSP arbiter is dimensioned according to the number of requestors that share the resource, its device utilization and operating frequency depend on their count. Figures 7.4(a) and 7.4(b) show how device utilization and operating frequency of the CCSP arbiter change with increasing number of requestors. The results indicate that as the number of requestors arbitrated by the CCSP arbiter increases, device utilization increases and operating frequency drops.

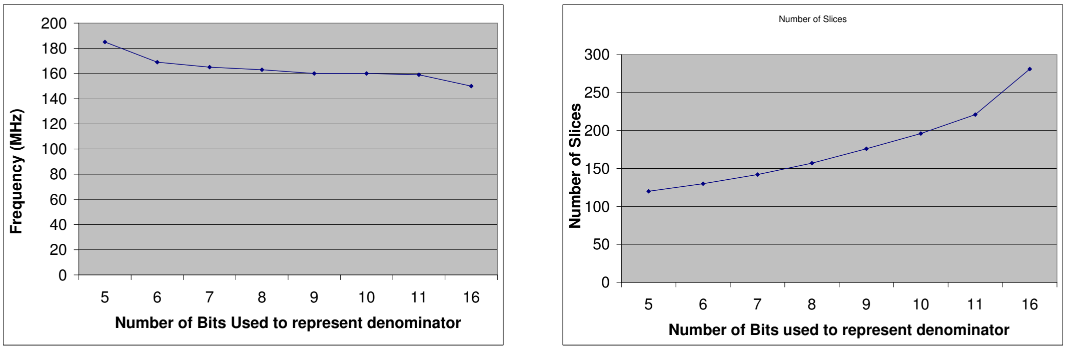


(a) Operating Frequency

(b) Device Utilization (Number of Slices)

Figure 7.4: Synthesis Result for the CCSP Arbiter - for Different Number of Requestors (a),(b)

While synthesizing the CCSP arbiter, it has been found that the critical path passes through the unit that performs credit management (See 5.2.3.4). The complexity of this operation depends on the precision used in credit representation and hence on the number of bits used to represent *numerator* and *denominator* values of the allocated service rate (ρ). To estimate the cost of increasing precision in the credit calculation used by the CCSP arbiter, synthesis has been carried out considering different number of bits to represent *numerator* and *denominator* values (See Section 5.2.3.1). Operating frequency and Device utilization of the CCSP Arbiter for different precisions are shown in Figures 7.5(a) and 7.5(b), respectively. As can be seen from the result in Figure 7.5(a), the operating frequency drops from 170 MHz to 160 MHz as the number of bits is increased from 6 to 12. The device utilization also increases proportionally with the number of bits used.



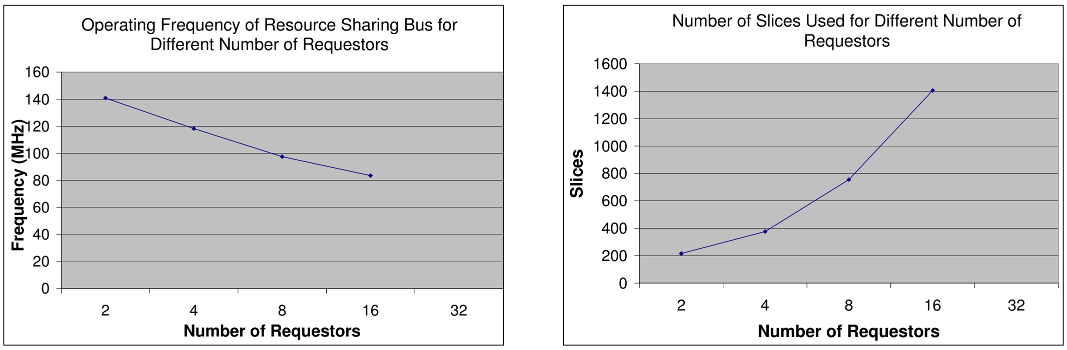
(a) Operating Frequency

(b) Device Utilization (Number of Slices)

Figure 7.5: Synthesis Result for CCSP Arbiter - for Different Precisions (a),(b)

7.1.4 Resource Sharing Bus

The resource sharing bus is also dimensioned according to the number of requestors that share the resource. The operating frequency and device utilization of the bus, with CCSP arbiter inside, are plotted for different number of requestors in Figures 7.6(a) and 7.6(b), respectively. The results, in Figure 7.6 show that with more number of requestors, the operating frequency of the bus drops and its device utilization increases. Once more, the operating frequency of the resource sharing bus, is determined by the credit calculation in the arbiter. This critical path, which starts from the CCSP arbiter and ends at the request multiplexer in the bus is shown in Figure 7.7.



(a) Operating Frequency

(b) Device Utilization (Number of Slices)

Figure 7.6: Synthesis Result for The Resource Sharing Bus

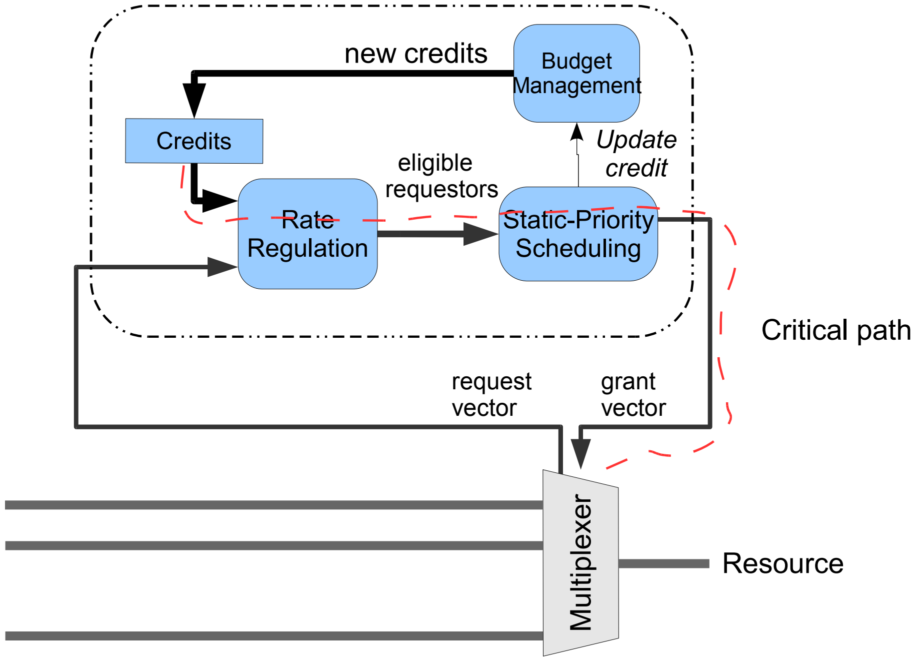


Figure 7.7: The Critical Path in the CCSP Arbiter and the Resource Sharing Bus

7.2 Synthesis ASIC

The front-end has been synthesized for ASIC targeting 90nm CMOS Technology with area optimization for different target frequency values. The synthesis was done using the *CMOS090LP Core Library corelib 3.0.1* with 1.2V core signaling. With four requestors and a buffer of depth 1 in the Delay Block, the entire front-end can operate at up to *425 MHz* and its total area is estimated to be *12210 cells*, equivalent to $0.120mm^2$. The size of the blocks in the front-end is shown with pie chart in Figure 7.8. The areas shown correspond to four Delay Blocks, four Atomizers, an arbiter dimensioned for four requestors and a resource sharing bus for four requestors. Hereafter, cell count is used to estimate the area of units. The area of a cell is, on average, $10^{-5}mm^2$.

The critical path in the front-end is the one through the Delay Block, shown in Figure 7.2, where time stamps are computed. As the number of requestors is increased, however, the operating frequency of the front-end drops and the critical path becomes the one in the CCSP arbiter (and the resource sharing bus). The critical path in the CCSP Arbiter is shown in Figure 7.7.

The individual blocks in the front-end have been synthesized with different settings. The CCSP arbiter has been synthesized for different number of requestors and different precision values. Similarly, the Delay Block has been synthesized with different buffer sizes and different precisions for the representation of *completion latency* (λ_n *and* λ_d). The synthesis result for each block is presented next. Unless explicit mention is made, the number of requestors is considered to be *4* and *6-bits* are used to represent *numerator* and *denominator* in the CCSP arbiter.

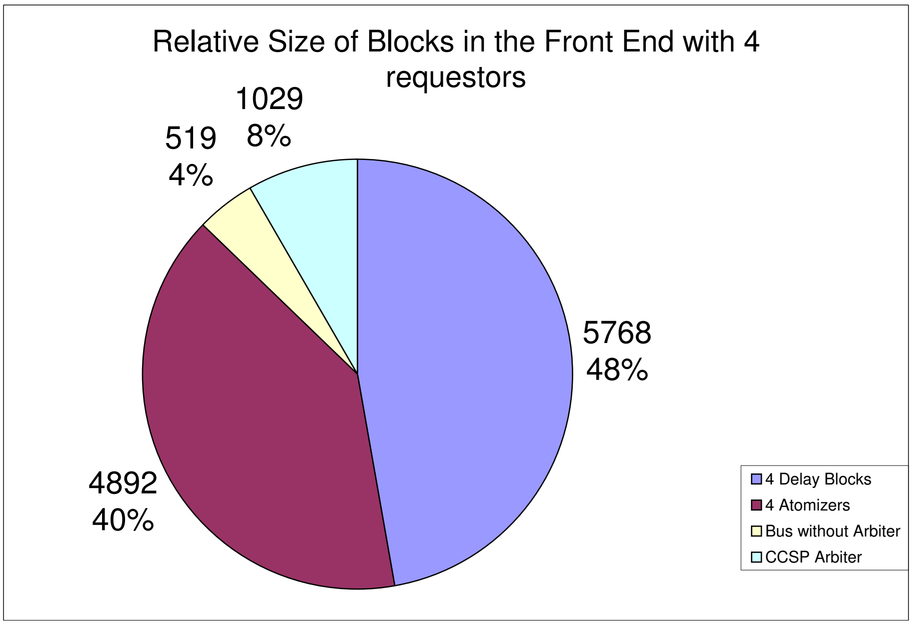


Figure 7.8: Relative Size of Units in the Front-end

7.2.1 Synthesis Results for Atomizer

The Atomizer is the fastest of all the blocks in the front-end. With optimization for speed, it can run up to *475 MHz* and consumes *1223 cells*. An Atomizer is instantiated per requestor and, hence, synthesis results do not change with number of requestors.

7.2.2 Synthesis Results for Delay Block

Like Atomizers, a Delay Block is instantiated per requestor. Thus, the size and performance of individual Delay Blocks does not depend on the number of requestors. The size of a Delay Block is determined by the depth of request and response buffers. With buffers of depth 1, a Delay Block consumes *1442 cells*. As the depth is increased, the area also increases proportionally. Figure 7.9 shows how its size grows with increasing buffer size.

Another setting of the Delay Block which affects its operating frequency is the number of bits used to represent *numerator* and *denominator* parts of the completion latency (λ). As can be seen from the plots in Figure 7.10, operating frequency of the Delay Block drops with increasing precision (*number of bits*). When 6-bits are used, the Delay Block can operate at *425 MHz*. When the number of bits is increased to 20, however, its operating frequency will be limited to *275 MHz*.

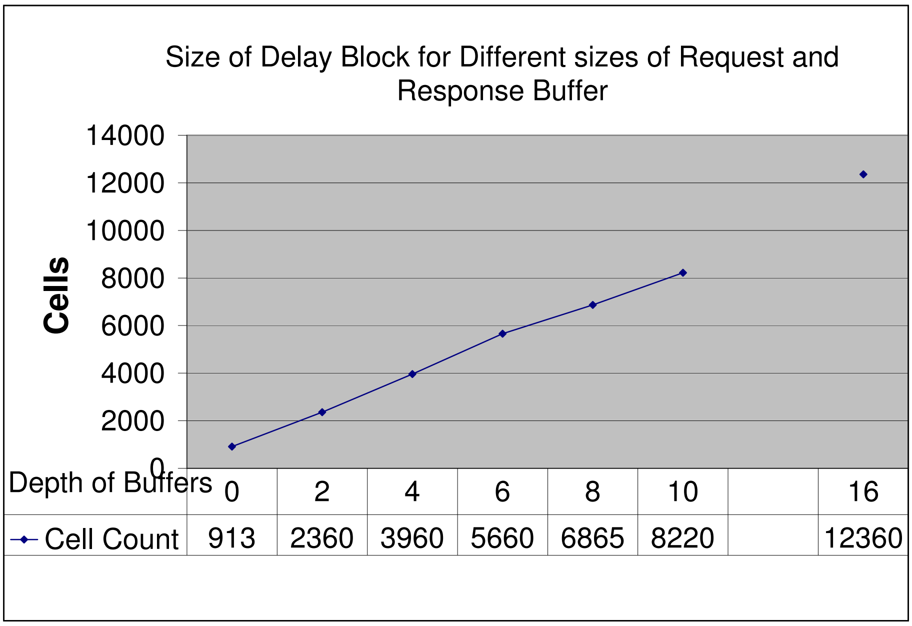


Figure 7.9: Size of the Delay Block as a Function of Buffer Size

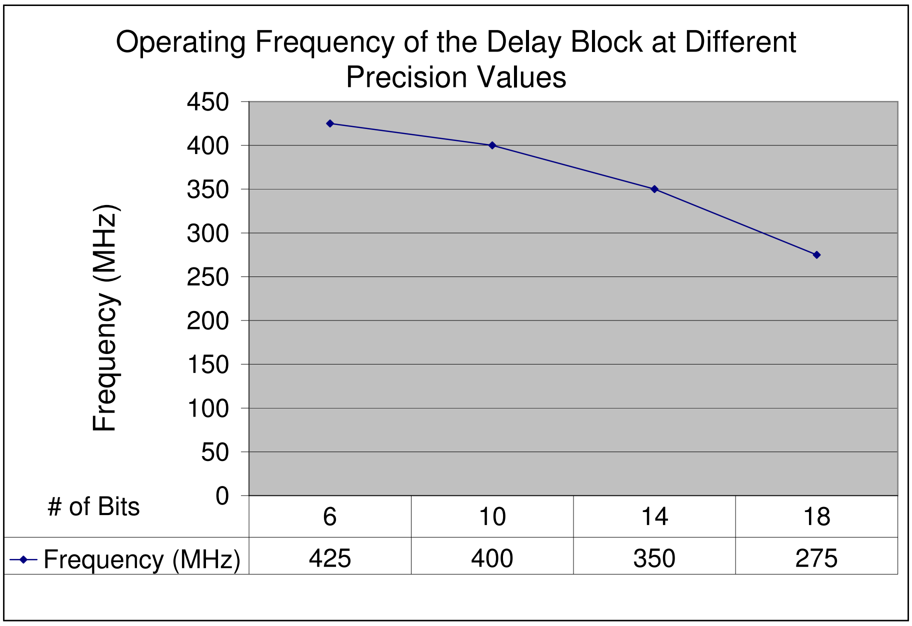
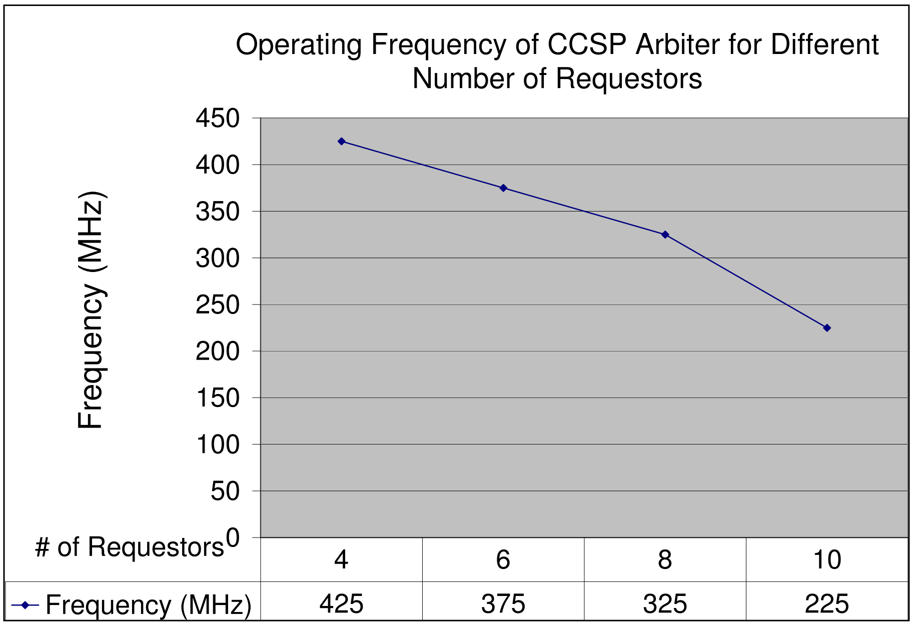


Figure 7.10: Operating Frequency of the Delay Block for Different Precision Values

7.2.3 Synthesis Results for CCSP Arbiter

The operating frequency and size of the CCSP Arbiter are determined by the number of requestors connected to the front-end and the precision used for credit management. Figure 7.11 shows how the operating frequency of the CCSP arbiter drops with increasing number of requestors. Considering 6-bits for the representation of *numerator* and *denominator*, the arbiter becomes the bottleneck of the entire front-end when the number of requestors exceeds 6. With 8 requestors, for instance, the operating frequency of the CCSP Arbiter stands at *325 MHz* whereas the Delay Block and the Atomizer are able to run above *400 MHz*.



# of Requestors	4	6	8	10
Frequency (MHz)	425	375	325	225

Figure 7.11: Operating Frequency of the CCSP Arbiter for Different Number of Requestors

The CCSP Arbiter has been synthesized for different number of requestors *(n = 4, 6, 8 and 10)* at different target frequencies *(f = 200, 250, 300 and 350)*. With 10 requestors, however, operating frequency of *300 MHz* and *350 MHz* could not be achieved due to negative slack in the CCSP Arbiter. The result shows area (number of cells) required to achieve a certain operating frequency for a given number of requestors. As can be seen from the plots in Figure 7.12, the size of the CCSP arbiter becomes larger with higher target frequency due to lower possibility of optimization.

Another factor that determines the operating frequency of the CCSP Arbiter is the number of bits used in the credit computation. Figure 7.13 shows the maximum operating frequency of the CCSP arbiter for different number of bits used to represent *numerator* and *denominator*.

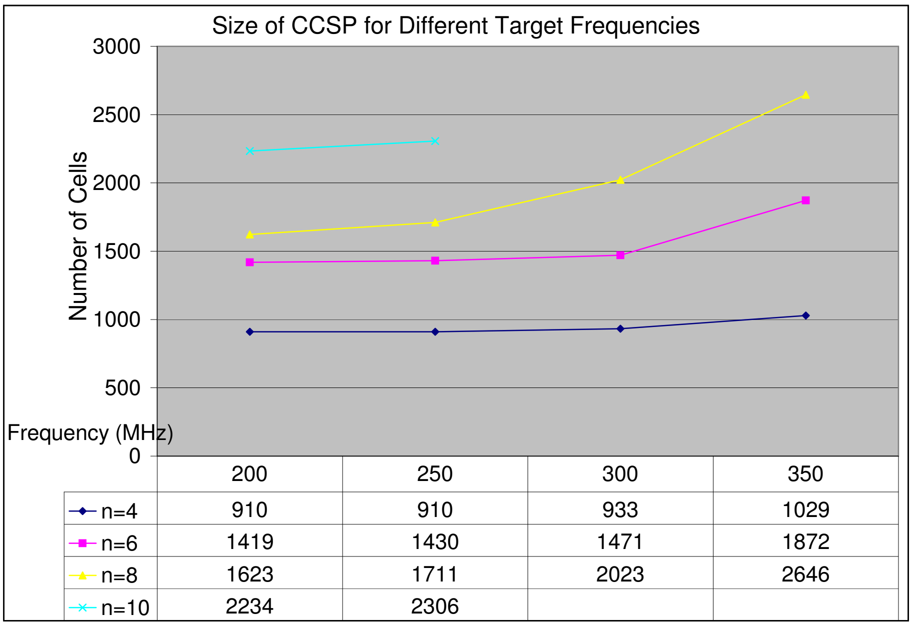


Figure 7.12: Size of the CCSP Arbiter for Different Number of Requestors

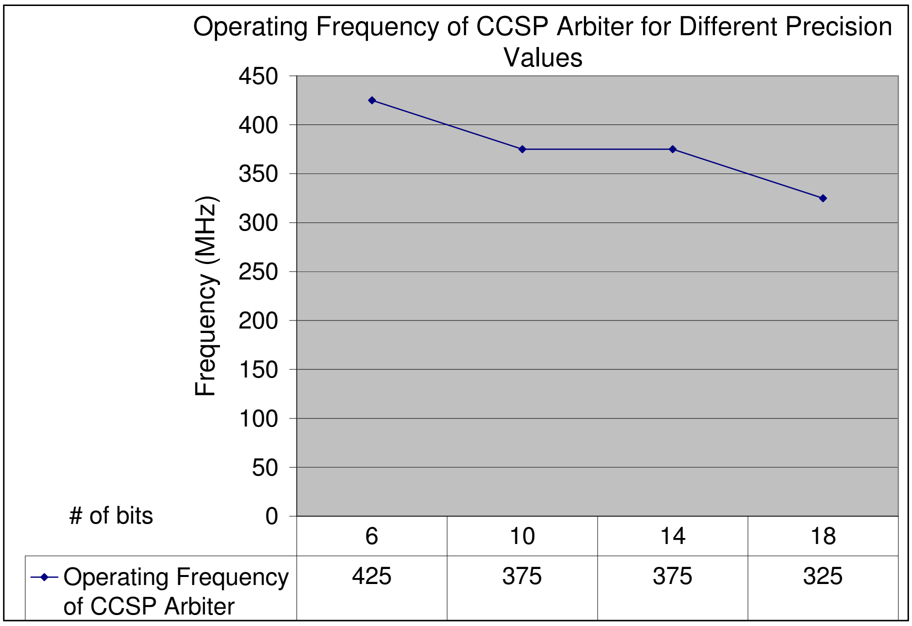
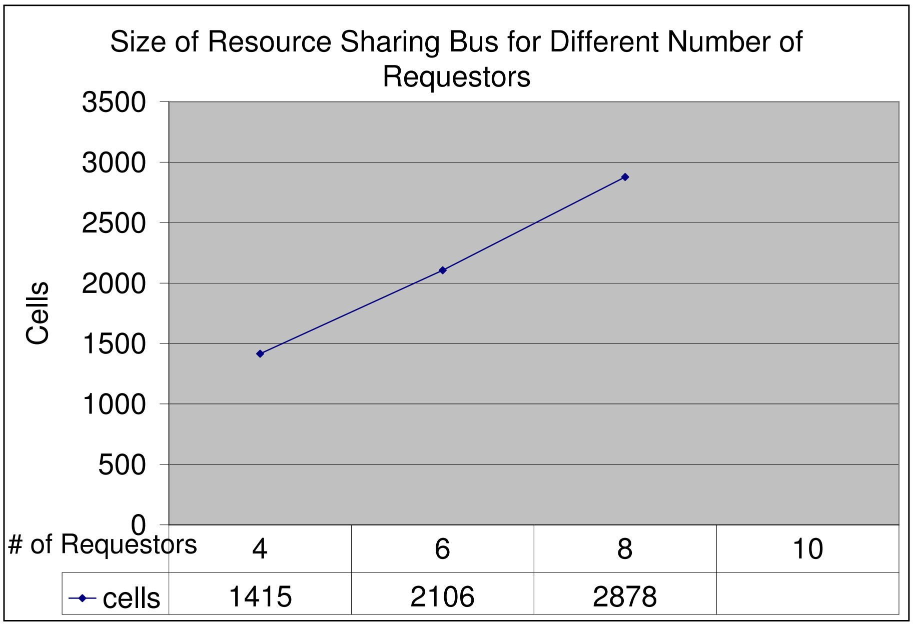


Figure 7.13: Operating Frequency of the CCSP Arbiter for Different Precision Values

7.2.4 Synthesis Results for Resource Sharing Bus

Since the Resource Sharing Bus contains the CCSP arbiter, its frequency is determined by the critical path of the arbiter. Hence its operating frequency is similar to that of the CCSP Arbiter. Since the bus has *Request Multiplexer* and *Response Demultiplexer*, in addition to the Arbiter, its size grows with the number of requestors attached to it. Figure 7.14 shows the size of the bus for different number of requestors, synthesized with



# of Requestors	4	6	8	10
◆ cells	1415	2106	2878	

Figure 7.14: Size of the Resource Sharing Bus for Different Number of Requestors

a target frequency of 300 MHz. For 10 requestors, 300 MHz could not be achieved due to the critical path in the CCSP Arbiter.

8 Conclusions

With the growing trend of integrating multiple applications on a single chip, the complexity of system verification has become tremendous. In such systems, applications are started and stopped at run time resulting in multiple use cases. As the various applications share the available resources, the performance of one application is affected by the behavior of other applications that share the same resources. Thus, in verifying such a system, the interference due to resource sharing should be put in to consideration.

Traditionally, the entire system is verified as a whole considering each use case. However, the number of use cases grows exponentially with the number of applications and, hence, the associated verification effort grows in the same manner. Composable Systems are proposed to mitigate the complexity of system verification and integration. In a composable system, interference between applications is eliminated and hence applications can be verified independently by simulation.

In this work, a front-end for composable resource sharing, using $\mathcal{LR}$ servers, has been designed and implemented. The front-end enables resource sharing among applications, in which the service provided to one application does not depend on how others behave. With this approach, maximum interference is emulated for each application thereby shielding interference from others. Applications are prevented from using any slack (unused resource capacity) that results from change in behavior of others.

The front-end has been tested by simulation using an SRAM as a shared resource. It has also been synthesized and tested on FPGA. It has been demonstrated that the service that each requestor gets is not affected by the activity of other requestors. Since the system is composable at the level of requestors, we can conclude that it will be composable on the level of applications too [2].

The front-end is required in the system to bring about composable resource sharing and, hence, reduce verification and integration effort. However, its inclusion in the system also has impact both on system performance and cost. Since the entire front-end has four pipeline stages, it increases the average end-to-end latency of requests. The front-end also lowers the throughput of the resource with small buffers in the Delay Blocks. By using larger buffers, however, it has been shown that it is possible to eliminate the effect on throughput.

From the results of synthesis (both on FPGA and ASIC), the following conclusions are made.

- The front-end incurs area overhead that grows with the number of requestors that share the same resource.
- With up to six requestors sharing a resource, the critical path of the design lies in the Delay Block due to the computation of time stamps. For larger number

of requestors, however, the CCSP arbiter the bottleneck due to the computations involved in credit management.

- The number of bits used to represent the numerator and denominator parts of the allocated rate have significant effect on performance. When more number of bits are used, the operating frequency drops. Increasing the number of bits also increases the size.

- As can be seen from the graphs in Figure 7.8 and 7.9, the size of the front-end is dominated by the Delay Blocks, specially with larger buffers for requests and responses. Hence, the buffers in each Delay Block should be sized to an optimal depth which is enough to satisfy throughput requirement of the requestor.

9 Future work

Although the design has been tested and has conformed to the requirements set, there are more things to be done in order to make it more robust and efficient. The following are recommended for consideration in the future :

- The role of the Atomizer is, currently, just chopping requests into a fixed size with the assumption that all requests are aligned. To lift this restriction on the type of requests, aligning functionality can be added to the atomizer.
- The critical path in the Delay Block is due to the computation of time stamps which involves the approximation mechanism based on credit counter (See Algorithm 5.4). Pipelining the computation can, thus, shorten the critical path and improve performance of the Delay Block.
- All the tests in this work have been carried out using an SRAM, which takes a cycle to serve requests, as a resource. The front-end has to be tested with other types of predictable resources.

Specification in the Æthereal Design Flow

A

A.1 Architecture Specification

```
< architecture id=”architecture_name”>
< parameter id=”clk” type=”int” value=”400” />
< parameter id=”slotsize” type=”int” value=”3” />
< parameter id=”wordsize” type=”int” value=”4” />
< parameter id=”pckhdr” type=”int” value=”1” />
< parameter id=”cmdsize” type=”int” value=”2” />
< parameter id=”maxfc” type=”int” value=”31” />
< parameter id=”maxpcklen” type=”int” value=”8” />
< parameter id=”riqueue” type=”int” value=”8” />
< parameter id=”niiqueue” type=”int” value=”16” />
< parameter id=”nioqueue” type=”int” value=”16” />
< parameter id=”link_pipeline_stages” type=”int” value=”0” />


<ip id=”mb0” type=”IP”>
  <port id=”i1” type=”Initiator” protocol=”MMIO_DTL”>
    < parameter id=”width” type=”int” value=”32” />
    < parameter id=”blocksize” type=”int” value=”32” />
  < /port>
< /ip>
< ip id=”mb1” type=”IP”>
  < port id=”i1” type=”Initiator” protocol=”MMIO_DTL”>
    < parameter id=”width” type=”int” value=”32” />
    < parameter id=”blocksize” type=”int” value=”32” />
  < /port>
< /ip>
< ip id=”mb2” type=”IP”>
  < port id=”i1” type=”Initiator” protocol=”MMIO_DTL”>
    < parameter id=”width” type=”int” value=”32” />
    < parameter id=”blocksize” type=”int” value=”32” />
  < /port>
< /ip>
< ip id=”mb3” type=”IP”>
  < port id=”i1” type=”Initiator” protocol=”MMIO_DTL”>
    < parameter id=”width” type=”int” value=”32” />
    < parameter id=”blocksize” type=”int” value=”32” />
  < /port>
< /ip>
< ip id=”memory” type=”IP”>
  < port id=”t1” type=”Target” protocol=”MMIO_DTL”>
    < parameter id=”width” type=”int” value=”32” />
    < parameter id=”delay” type=”string” value=”1” />
    < parameter id=”arbiter” type=”string” value=”CCSP” />
  < /port>
< /ip>
< /architecture>
```

A.2 Specification of Communication Requirements

```
<communication>
  <application id=”application0”>
    <connection id=”0” qos=”GT”>
      < initiator ip=”mb0” port=”i1” />
      < target ip=”memory” port=”t1” />
      < read bw=”1” burstsize=”32” latency=”0” />
      < parameter id=”delay” type=”string” value=”1” />
    < /connection>
    < connection id=”1” qos=”GT”>
      < initiator ip=”mb1” port=”i1” />
      < target ip=”memory” port=”t1” />
      < read bw=”100” burstsize=”4” latency=”0” />
      < parameter id=”delay” type=”string” value=”1” />
    < /connection>
    < connection id=”2” qos=”GT”>
      < initiator ip=”mb2” port=”i1” />
      < target ip=”memory” port=”t1” />
      < read bw=”200” burstsize=”8” latency=”0” />
      < parameter id=”delay” type=”string” value=”1” />
    < /connection>
    < connection id=”3” qos=”GT”>
      < initiator ip=”mb3” port=”i1” />
      < target ip=”memory” port=”t1” />
      < write bw=”40” burstsize=”4” latency=”0” />
      < parameter id=”delay” type=”string” value=”1” />
    < /connection>
  < /application>
< /communication>
```

Bibliography

[1] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, *Real-Time scheduling using Credit-Controlled Static-Priority arbitration*, Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on, 2008, pp. 3–14.

[2] Benny Akesson *et al.*, *Composable resource sharing based on latency-rate servers*, Proc. DSD, August 2009.

[3] Kees Goossens, John Dielissen, and Andrei Radulescu, *Æthereal network on chip: Concepts, architectures, and implementations*, IEEE Des. Test **22** (2005), no. 5, 414–421.

[4] RL Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics (1969), 416–429.

[5] Andreas Hansson, *A composable and predictable on-chip interconnect*, Technische Universiteit Eindhoven, 2009.

[6] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken, *CoMPSoC: a template for composable and predictable multi-processor system on chips*, ACM Trans. Des. Autom. Electron. Syst. **14** (2009), no. 1, 1–24.

[7] Applied Dynamics International, *What are soft and hard real-time applications?*, http://www.adi.com, 2007.

[8] *International technology roadmap for semiconductors(itrs) -design 2007*, http://www.itrs.net, 2007.

[9] H. Kopetz and G. Bauer, *The time-triggered architecture*, Proceedings of the IEEE **91** (2003), no. 1, 112–126.

[10] J.W. Lee and K. Asanovic, *METERG: Measurement-Based End-to-End performance estimation technique in QoS-Capable multiprocessors*, Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE, 2006, pp. 135–147.

[11] T. Lundqvist and P. Stenstrom, *Timing anomalies in dynamically scheduled microprocessors*, Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE, 1999, pp. 12–21.

[12] Roman Obermaisser, *Integrating automotive applications using overlay networks on top of a Time-Triggered protocol*, http://dx.doi.org/10.1007/978-3-540-77419-8_11, 2007.

[13] Philips, *Device Transaction Level Protocol*, SIAM Journal on Applied Mathematics (1969), 416–429.

[14] B. Rumpler, *Complexity management for composable real-time systems*, Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on, 2006, p. 9 pp.

[15] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P.P. Pande, C. Grecu, and A. Ivanov, *System-on-Chip: reuse and integration*, Proceedings of the IEEE **94** (2006), no. 6, 1050–1069.

[16] sharef2html. xsl 524 2006-06-29 20:38:52Z dret, *Cramming more components onto integrated circuits [gordon e. moore]*, http://dret.net/biblio/reference/moo65, June 2009.

[17] D. Stiliadis and A. Varma, *Latency-rate servers: a general model for analysis of traffic scheduling algorithms*, Networking, IEEE/ACM Transactions on **6** (1998), no. 5, 611–624.

[18] E. Strooisma, *A predictable and composable front-end for system on chip memory controllers*, Technische Universiteit Delft, 2008.

[19] VaST Systems Technology, *Virtual prototyping for embedded systems design*, http://www.vastsystems.com, 2006.

[20] Economist Intelligence Unit, *Consumer electronics gets ready for change how convergence will drive new business models*, http://www.chamber.org.hk/info/eiu/ThoughtLeadership, 2008.

[21] Xilinx, *Microblaze soft processor core*, http://www.xilinx.com, 2009.