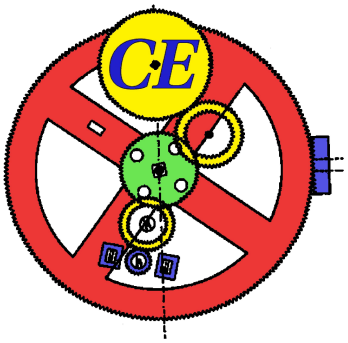


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Performance estimation technique for optimizing and integrating IPs in MPSoCs

Anand Subhash Khot

CE-MS-2011-12

Over the last decade, the complexity of system-on-chips (SoCs) has continuously increased owing to the increasing demand for high performance IPs and SoCs. However, the productivity of chip designers has not scaled up at the same rate. This has led to an enormous design productivity gap. At the same time, the increasing time-to-market pressure and the high risk of design failure have all fostered the development of IP re-use based designs. One of the major challenges in re-using IPs is that it is difficult to configure and verify the performance of IPs/ IP subsystems after they are integrated into an existing SoC with a given infrastructure (on-chip network, memory subsystem, etc.). To overcome these challenges, we propose two performance estimation techniques that are based on high-level performance modeling of IPs and SoC infrastructure. These models capture some of their key performance characteristics (e.g. latency tolerance of IPs) and help relate the performance dependence of IPs on the service provided by the SoC infrastructure. Along with the advantage of re-using the high-level IP models in multiple SoC designs, the models allow the SoC designer to iteratively estimate the performance of a SoC over a range of IP and SoC infrastructure configurations, thereby aiding the design space exploration process. The proposed performance estimation techniques are particularly useful in rapidly re-assessing the performance of all IP/ IP subsystems once they are integrated into a given SoC design. The performance estimates provided by these techniques in the early SoC design stages saves a significant portion of the precious design time. The performance estimation techniques therefore simplify the process of integrating new IPs/ IP subsystems into existing SoC designs.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Performance estimation technique for optimizing and integrating IPs in MPSoCs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Anand Subhash Khot
born in Belgaum, India

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

Performance estimation technique for optimizing and integrating IPs in MPSoCs

by Anand Subhash Khot

Abstract

Over the last decade, the complexity of system-on-chips (SoCs) has continuously increased owing to the increasing demand for high performance IPs and SoCs. However, the productivity of chip designers has not scaled up at the same rate. This has led to an enormous design productivity gap. At the same time, the increasing time-to-market pressure and the high risk of design failure have all fostered the development of IP re-use based designs. One of the major challenges in re-using IPs is that it is difficult to configure and verify the performance of IPs/ IP subsystems after they are integrated into an existing SoC with a given infrastructure (on-chip network, memory subsystem, etc.). To overcome these challenges, we propose two performance estimation techniques that are based on high-level performance modeling of IPs and SoC infrastructure. These models capture some of their key performance characteristics (e.g. latency tolerance of IPs) and help relate the performance dependence of IPs on the service provided by the SoC infrastructure. Along with the advantage of re-using the high-level IP models in multiple SoC designs, the models allow the SoC designer to iteratively estimate the performance of a SoC over a range of IP and SoC infrastructure configurations, thereby aiding the design space exploration process. The proposed performance estimation techniques are particularly useful in rapidly re-assessing the performance of all IP/ IP subsystems once they are integrated into a given SoC design. The performance estimates provided by these techniques in the early SoC design stages saves a significant portion of the precious design time. The performance estimation techniques therefore simplify the process of integrating new IPs/ IP subsystems into existing SoC designs.

Laboratory : Computer Engineering
Codenumber : CE-MS-2011-12

Committee Members :

Advisor:	Prof. Kees Goossens, ES, TU Eindhoven
Advisor:	Dr. ir. Pieter van der Wolf, Synopsys
Chairperson:	Dr. ir. Sorin Cotofana, CE, TU Delft
Member:	Dr. ir. Georgi Kuzmanov, CE, TU Delft

To my parents

Contents

List of Figures

List of Tables

Acknowledgements

Firstly, I would like to thank Prof. Kees Goossens and Dr. Pieter van der Wolf for giving me the opportunity to conduct my MSc thesis at Synopsys under their guidance and mentorship. I am grateful for Prof. Goossens's reviews, suggestions for improvements and expert guidance throughout the thesis. I am equally grateful to Pieter for his encouragement, valuable advice and supervision at Synopsys. I would like to express my deepest gratitude to Dr. Benny Akesson (Post. Doc at TU, Eindhoven) for his suggestions and thorough review of my work. I would specially like to thank Prof. Sorin Cotofana for accepting to supervise my thesis from Delft. I would also like to thank Prof. Georgi Kuzmanov for accepting to judge my thesis defense. I am grateful to the staff of Synopsys and the *Memory Team* at TU Eindhoven, especially Firew, Sven, Tim, Karthik, Manil and Davit for creating a pleasant work environment.

Finally, I would like to thank my parents and my sister for their love, care and support. Last but not the least, I would like to thank my friends in the Netherlands, especially my flatmates in Delft and Eindhoven, for making the last 2 years special in my life.

Anand Subhash Khot
Delft, The Netherlands
June 23, 2011

1 Introduction

1.1 System-on-Chip

System-on-Chip (SoC) refers to the technology of integrating all components of an electronic system into a single integrated circuit. Figure 1.1 depicts a system level representation of a typical SoC. It can be observed from Figure 1.1 that a contemporary system-on-chip contains various intellectual property (IP) elements such as processing cores (e.g. ARM, MIPS), DSPs, video engines, on-chip network, audio/video subsystem, memory subsystem etc., along with analog and RF components such as PLLs, ADCs, wireless transceivers, data converters etc. IPs such as processors, DSPs and other processing engines execute the associated software to provide the intended functionality. Thus, both the hardware (IPs) and the associated software are an integral part of a system-on-chip.

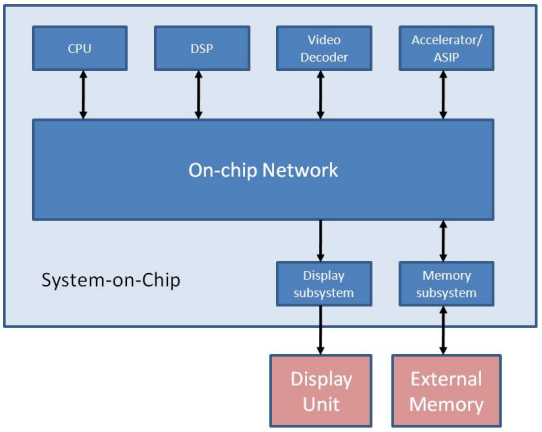


Figure 1.1: A contemporary system-on-chip (SoC)

1.2 SoC Design

Over the last several years, SoCs have become increasingly complex. This is primarily because of the increasing number of multimedia and real time applications supported by modern day SoCs. Due to these applications, the demand for high performance IPs and SoCs has increased. Consequently the number and types of IPs developed to execute these applications have increased over the years.

However, in spite of the ever-increasing growth in the demand for advanced SoCs and the semiconductor process technology, the productivity of chip designers has not scaled

up at the same rate. Over the last decade, the number of raw transistors increased at 58% per year (according to Moore's law), whereas the capability of chip designers to design them increased only at a rate of 20% per year, thereby creating an enormous design productivity gap [9]. The design productivity gap is predicted to further widen in the coming future, suggesting that the effort required to design complex chips will only increase. Also, the high cost associated with a design re-spin makes it inevitable for SoC designers to be 'first time right' with each and every SoC design. Thus, the increasing design productivity gap and the high risk of failure associated with SoC designs are the two main SoC design challenges. Along with this, SoC designers are also faced with the challenge of finding the right balance between the power, area, performance, cost (in dollars) and time-to-market constraints of modern SoCs. Due to all these factors, designing next generation SoCs has become extremely challenging and time consuming.

To address the issues of design productivity and design verification, several approaches have been developed. One approach is to develop new design methodologies and tools which allow designers to abstract the design at a higher level, thereby simplifying the design process [2, 6]. It mainly addresses the *design complexity* issues of SoC design. These methodologies and tools help raise the abstraction level at which SoCs are designed and thus, help designers build larger and more complex systems. Some of the design tools are equipped with fast simulators and formal verification techniques to improve design verification and thus, reduce the risk of design failures.

Another approach is to encourage and maximize IP re-use [3, 16, 17], which allows large SoC designs to be partitioned into smaller IP subsystems with specific, well-defined functionality. The reuse of qualified, pre-verified IPs in multiple SoC designs minimizes the risk of design failures and reduces the time-to-market and the high NRE cost associated with SoC designs. To summarize, IP re-use offers the following advantages:

1. Reduced time-to-market.
2. Reduced NRE cost.
3. Reduced risk of design failures.
4. Simplified design analysis and customization.

Over the last couple of decades, the market dynamics, the increasing SoC complexity and the time-to-market pressures have all led to the adoption and development of IP re-use based designs.

1.3 IP re-use

In general, IP re-use refers to the concept of directly using an IP or an IP subsystem from an existing SoC design. Such re-use of IPs is commonly found within the design groups of a semiconductor company [17]. The IP re-use based design methodology is particularly useful in rapidly evolving and highly competitive markets (such as mobile phones, consumer electronics, etc.), where the next generation SoCs are not designed from scratch, but are built by re-using a considerable portion of an existing SoC.

SoC design based on IP re-use typically involves starting the design process with an existing SoC and then, adding/replacing new IP subsystems as per the design requirements. In case the required IPs/ IP subsystems are not available with the design house, they are directly purchased from third party IP vendors [16]. The IP subsystems sold by the IP vendors are pre-verified for functional correctness and support industry standard interfaces (AHB, AXI, etc.), such that they can be integrated easily into custom SoC designs. Since the IP subsystems provide some commonly required functionality, they are generic enough to allow some of the design parameters (e.g. size of data buffers, etc.) to be tweaked and reconfigured as per the performance requirements of individual SoC designs. In this thesis, we consider that the on-chip network, the memory subsystem and the external DRAM memory together form the common, shared *SoC infrastructure* that is used by all IPs/ IP subsystems.

The task of the SoC designer is to verify if the newly added IP/ IP subsystem (with a given configuration) performs as expected when it is coupled to the shared SoC infrastructure. This may require exploring the different possible configurations of the IPs/ IP subsystems and the SoC infrastructure such that they fulfill all the performance requirements of real-time applications. Also, the addition/replacement of an IP/ IP subsystem may invalidate the performance guarantees of other IPs/ IP subsystems of the original SoC. Therefore, it is crucial to ensure that all IPs/ IP subsystems of the original SoC still perform as expected even after the integration of new IP/ IP subsystems.

Thus, as explained above, there are a number of challenges involved in adopting the IP re-use based design methodology . To summarize, it is extremely tedious and time consuming to customize (through configuration) both the IPs and the SoC infrastructure such that they fulfil their performance requirements. Also, the system level performance verification of IPs and SoC infrastructure every time a new IP/ IP subsystem is added to an existing SoC design consumes a significant portion of the precious design time. *Thus, rapid customization, integration and performance verification of IPs and the supporting SoC infrastructure are the key challenges in simplifying and enhancing the IP re-use based design methodology, which in turn, increases design productivity and minimizes time to market and the risk of design failure.*

1.4 Problem Statement

In this thesis, we focus on IP re-use based designs (explained in the Section 1.3) and investigate in detail, the following two critical design challenges:

1. IP customization and integration
2. Performance verification

1.4.1 IP customization and integration

In the context of this work, we use the term *IP customization and integration* to denote the challenges involved in customizing both the IP/ IP subsystems and the supporting infrastructure of a given SoC design. As explained previously, SoC designers typically re-use existing SoC designs and hence, do not need to select the IPs or the SoC infrastructure

per se. However, SoC designers often need to tweak the IP/ IP subsystems to suit a given application [19]. At the same time, they need to appropriately dimension the SoC infrastructure such that it successfully supports the performance requirements of all IP/ IP subsystems. The key questions which we specifically try to answer in this thesis are as follows:

1. Does an IP/ IP subsystem satisfy all the performance requirements of the application after it is integrated into a SoC with a given infrastructure? If not, can they be reconfigured to achieve the desired level of performance?

2. If a given IP/ SoC infrastructure does satisfy all the performance requirements, is there any headroom available to optimize it further?

3. In the case of pipelined multi-threaded IPs, how many outstanding memory requests should it ideally support in order to satisfy the performance requirements of all real time functions it executes?

4. In the case of IPs employing an advanced pre-fetching cache, how large should the pre-fetch data buffers be in order to allow sufficient pre-fetching of the necessary data, thereby fulfilling all the performance requirements of the IP and the application?

5. Given a SoC infrastructure, how do we configure it such that all IPs/ IP subsystems meet their performance requirements? The configuration of the SoC infrastructure typically involves configuring the network topology, the memory controller arbitration technique, etc.

In this thesis, we consider that one of the key concerns while answering the above questions, is to satisfy the performance requirements of all IPs/ IP subsystems sharing the SoC infrastructure. Although the performance requirements of the IP/ IP subsystems are determined by a variety of factors such as the net available bandwidth, the service latency of external memory requests, etc., in this thesis, we associate the performance of an IP/ IP subsystem with its *latency tolerance* and *latency criticality*. Latency tolerance and latency criticality are defined as follows:

Latency Tolerance

Latency tolerance of an IP/ IP subsystem is the ability of the IP to continue execution of the application code after issuing a memory request, until it finally stalls for the requested data. An IP/ IP subsystem which is not latency tolerant is termed as *latency sensitive* and vice versa.

Latency Criticality

Latency criticality of an IP/ IP subsystem is a measure of the severity of stalling the IP, with respect to the overall functionality of the application or the SoC. An IP/ IP subsystem which is not latency critical is termed as *latency non-critical* and vice versa.

Latency criticality and latency tolerance are orthogonal properties of an IP/ IP subsystem. It can be seen from Table 2.1 that is it possible to classify all IPs/ IP subsystems into the four possible combinations of latency tolerance and latency criticality. The concept of latency tolerance and latency criticality of IPs/ IP subsystems allows us to relate the performance of IPs with the service latency of the SoC infrastructure.

1.4.2 Performance verification

In the context of this work, we use the term *performance verification* to denote the process of performing system level performance verification of the entire SoC every time a new IP/ IP subsystem is added to the original SoC. Performance verification is a tedious and time consuming task, especially if the SoC designer undertakes design space exploration to find the right configuration of the IP/ IP subsystem or the SoC infrastructure. With respect to performance verification, we try to address the following key concerns:

1. Does a given IP/ IP subsystem (with specific configuration) satisfy all the performance requirements of the application when it is coupled to a SoC infrastructure (with specific configuration)?

2. Does addition of an IP/ IP subsystem invalidate the performance guarantees of other IPs/ IP subsystems already integrated into the SoC? Do all IPs/ IP subsystems still perform as expected?

These questions are crucial for the success of the SoC implementation. While re-using a considerable portion of an existing SoC design, the inclusion/replacement of a new IP/ IP subsystem should not degrade the performance of the other IP/ IP subsystems that share the common SoC infrastructure (i.e. NoC, memory subsystem, external DRAM memory). Thus, system level verification is necessary to verify whether the performance of all IPs/ IP subsystems are within their expected bounds after they are all integrated into the SoC.

Currently, performance verification is performed either by using cycle-accurate simulators/emulators [14, 10, 21] or by using analytical verification techniques[18, 12]. Simulation based verification is performed for the entire system, from processors/IPs to the on-chip network to the memory subsystem, which requires their HDL/ system-C models along with the entire application code. Furthermore, simulating the entire application using the IP's HDL/system-C model or an instruction set simulator, for a variety of different IP and SoC infrastructure configurations is extremely slow. On the other hand, analytical verification is fast, but relies on high level models that lack accuracy. Sometimes, analytical verification is performed using worst-case assumptions which works well with real time applications, but is not sufficiently realistic for soft/non real-time applications.

While integrating and optimizing IPs, designers often rely on past experiences and performance specifications of IPs/ SoC infrastructure to select and configure them. Often these IPs are over-dimensioned (by compromising on the bill of materials) to allow some headroom and thus, ensure performance guarantees. Even with over-dimensioning of the IPs/ SoC infrastructure, the performance guarantees need to be verified for every

change in the design. The fundamental issue behind all these challenges is the lack of performance estimation techniques that would allow SoC designers to rapidly select, configure and verify the performance of IPs and SoC infrastructure.

1.5 Aim

The aim of this thesis is to solve the challenges encountered in the integration, customization and system level performance verification of IPs and SoC infrastructure after they are integrated into a system-on-chip. One of the key challenges in successfully verifying the performance of an IP lies in our ability to characterize the IP by its latency tolerance and latency criticality. The thesis thus, aims to provide high level performance models which capture the latency tolerance and latency criticality of IPs, along with their interaction with the SoC infrastructure. The high level performance models allow rapid and accurate performance estimation of IPs and the SoC infrastructure. Any change in the configuration of the IPs and the SoC infrastructure is reflected in the model parameters, thereby allowing the SoC designer to perform rapid system-level verification of the entire SoC for a range of IP and SoC infrastructure configurations.

1.6 Proposed Solution

In this thesis, we propose two performance estimation techniques that allow rapid customization, integration and verification of IPs and the supporting SoC infrastructure. The performance estimation techniques are based on abstracting the IPs/ IP subsystems and the SoC infrastructure to their high level performance models. These performance models capture the key performance characteristics of IPs/ IP subsystems, such as their latency tolerance and latency criticality.

The latency tolerance of an IP is determined by analyzing the trace of external memory requests issued by that IP to the SoC infrastructure. Application developers generally provide a set of worst-case external memory request traces as a means for SoC designers to verify the performance of IPs for the worst-case application execution behaviour. In case the worst-case external memory request traces are not available, they can be easily extracted from the actual execution of the application code on the respective IPs/ IP subsystems. We choose to characterize the IPs/ IP subsystems using their external memory request traces since trace analysis is simple, quick and computationally cheap. Also, the characterization of IPs/ IP subsystems, once performed, can be re-used for customizing and integrating the respective IPs/ IP subsystems in all subsequent SoC designs.

The proposed performance estimation technique allows SoC designers to verify the performance of the SoC by either using an accurate trace simulator or a set of simplified equations utilizing average values. We call these performance estimation techniques as *trace-based performance estimation* and *simplified equation-based performance estimation*, respectively. Both these performance estimation techniques are applied to the SoC design after all IPs and the SoC infrastructure are abstracted to their simplified high-level performance models. The trace simulator performs cycle accurate performance estima-

tion by analyzing the memory request trace on a request-by-request basis, whereas the equation-based estimation technique estimates the performance of the entire SoC by using simple performance values. Both these performance estimation techniques allow rapid system-level performance estimation of the SoC in the early stages of SoC design.

Since the high-level performance models capture the key performance characteristics of the IP/ IP subsystems and the SoC infrastructure, any change in the configuration of the IP/ IP subsystem or the SoC infrastructure is reflected in the performance model also. Thus, by using the performance model, SoC designers can rapidly iterate over several IP and SoC infrastructure configurations to perform rapid design space exploration. Also, since the performance model allows SoC designers to quantify the tolerance of IPs, they can utilize any available headroom while configuring the IP or the SoC infrastructure.

Organization

The rest of the thesis is organized as follows. Chapter 2 introduces the high-level performance models for IPs and the SoC infrastructure. It briefly explains the basics of performance estimation and introduces the two performance estimation techniques. Chapter 3 presents the trace-based performance estimation technique and illustrates several experiments verifying the approach. Chapter 4 deals with the simplified equation-based performance estimation technique. The chapter provides thorough explanation of the approach, followed by experiments confirming its validity. Later the two performance estimation techniques are applied for verifying and integrating IP/ IP subsystems commonly found in real-life SoC designs. Chapter 5 presents the related work and highlights the differentiated solution offered by this thesis. Finally, chapter 6 summarizes the conclusions of the thesis, proposes several recommendations on using the performance estimation techniques and lastly, presents the possible future work.

General High level Performance Modeling

2

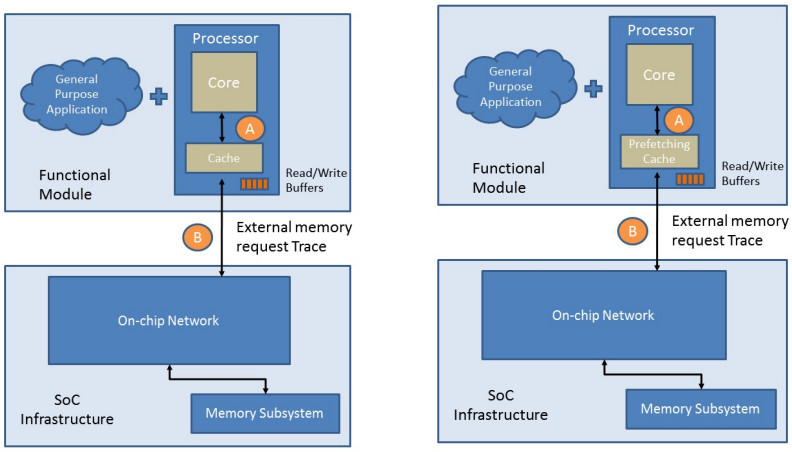
In Chapter 1, we discussed the various challenges associated with SoC design, especially from the IP re-use point of view. As a solution to the various challenges encountered in the customization, integration and verification of IPs in complex SoCs, we propose two novel performance estimation techniques. The goal of performance estimation is to instill confidence regarding the performance of an IP/ IP subsystem in co-ordination with a given SoC infrastructure early in the design phase. The proposed performance estimation techniques are based on the high-level performance models of IP/ IP subsystems and the SoC infrastructure. The performance models capture the execution behaviour of IP/ IP subsystems and the SoC infrastructure at a higher abstraction level. In this chapter, we study the performance models for IP/ IP subsystems and the SoC infrastructure.

2.1 Functional modules

The functionality of a typical SoC is achieved by executing a set of applications on a variety of IPs. In the context of this work, we consider that an application is made up of a number of functions, each of which is mapped to an IP. A function mapped to an IP constitutes a *functional module*. Each functional module performs a specific, well-defined function of the application and in most cases, interacts with other SoC elements such as the shared DRAM memory via the shared on-chip network. Henceforth, we use the term *functional module* to denote the combination of hardware (IP/ IP subsystem) and the associated software, whereas the term *IP/ IP subsystem* is used to specifically denote the hardware component of the functional module. We consider that the on-chip network, the shared memory subsystem and the external DRAM memory together constitute the *SoC infrastructure*.

2.1.1 Generic architecture of IPs:

Each IP (such as CPUs, GPUs, etc.) which constitutes a functional module, consists of a *core*, an *assistance unit* and *read/write buffers*. The core of an IP executes the given application code and is responsible for the actual data processing. The assistance unit of an IP, such as the on-chip cache, etc., assists the IP core(s) by providing them with the necessary data. If the IP is provided with an on-chip cache, all read/write requests issued by the core are catered by the cache itself. In case the requested data is not available in the cache, external memory requests are issued to the SoC infrastructure. Similarly, if the IP is provided with an advanced pre-fetching cache, the core collects the pre-fetched data directly from the on-chip *pre-fetch data buffers*. In this case, the pre-fetching cache of the IP issues all external memory requests (in advance) to the SoC infrastructure. The ability of the IPs to issue multiple outstanding read/write requests to the off-chip



(a) (b)

Figure 2.1: Examples of functional modules in SoCs

DRAM memory is directly dependent on the number of read/write buffers available in the IP.

2.1.2 Classification of functional modules

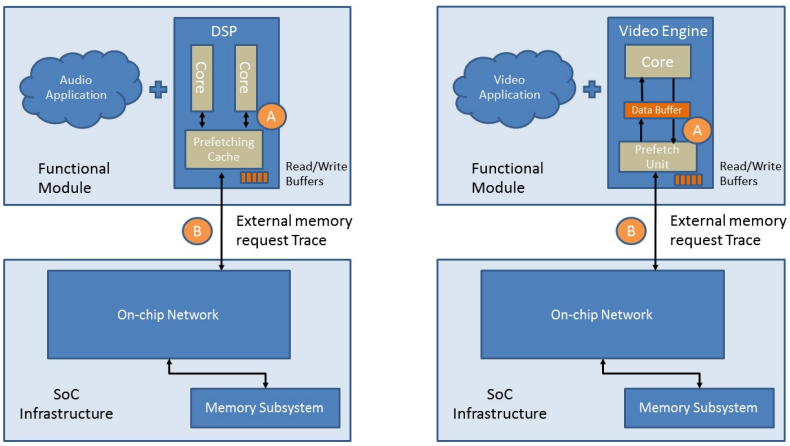
Functional modules can be characterized using a range of different parameters. However, as explained in Chapter 1, we characterize the functional modules by their latency tolerance and latency criticality. Based on their latency tolerance and latency criticality, functional modules can be classified into four categories as shown in Table 2.1. Given below are four functional modules that are commonly found in modern day SoCs. Each functional module belongs to one of the four functional module categories.

Basic CPU

A processor executing general purpose applications and assisted by a cache is shown in Figure 2.1(a). This functional module is latency sensitive since the processor stalls if its external memory requests are not served within a few (typically 3-4) clock cycles, thereby drastically reducing its performance. However, the functional module is latency non-critical since the consequences of stalling the processor while it executes general purpose applications are not severe with respect to the overall functionality of the SoC.

Pre-fetching CPU

An advanced processor executing general purpose applications and assisted by an advanced pre-fetching cache is shown in Figure 2.1(b). In this example, the functional module has higher latency tolerance than the basic CPU (i.e it can tolerate a higher SoC infrastructure service latency before it stalls). This is mostly because of the advanced pre-fetching capability of the cache, which starts pre-fetching the unavailable data as soon as it detects a pattern in the addresses of the



(a) (b)

Figure 2.2: Examples of functional modules in SoCs

issued memory requests. Again, the functional module is latency non-critical due to the nature of the general purpose applications that it executes.

DSP

A digital signal processor (DSP) executing hard real-time audio applications is shown in Figure 2.2(a). In this example, the functional module is latency sensitive and extremely latency critical, since missing audio sample deadlines leads to quick deterioration of the audio output which is unacceptable. In modern DSPs, caches and/or pre-fetch units are employed to take advantage of the streaming nature of audio applications. However, the code structure of audio applications is such that there are few opportunities to exploit memory level parallelism, thereby making these IPs latency sensitive.

Video Engine

A streaming video engine/ video decoder executing real-time video applications and assisted by a pre-fetch unit is shown in Figure 2.2(b). Both the streaming video engine and the video decoder exhibit high latency tolerance, but at the same time, are latency critical. Streaming video engines issue memory requests with a predictable address pattern, whereas the addresses of the requests issued by a typical video decoder are more random in nature. However, numerous algorithmic optimizations are performed on the decoding application, thereby allowing the decoder IP to successfully hide a large part of the external memory related latency. This makes the decoder IP latency tolerant. Similar to the audio applications, video applications also stream data from the external memory in a regular and predictable manner. Thus, a pre-fetch unit is employed to pre-fetch and store the required data in large data buffers. Missing a few external memory request deadlines for both, the streaming video IP and the video decoder IP is acceptable

as long as the collective deadline for a set of requests is met in time. Missing the collective deadline could result in dead pixels at the video output which is unacceptable from the performance point of view.

Table 2.1: Classification of functional modules

	Latency Tolerant	*Latency Sensitive*
Latency Critical	Video Engine	DSP
Latency Non-critical	Pre-fetching CPU	Basic CPU

2.2 Model for functional modules

The functional module, as the name suggests, performs a specific, well-defined function. It is essentially made up of IPs that execute specific function(s) on their core(s). At the lowest abstraction level, functions are made up of simple instructions such as add/-subtract/multiply (data processing), do-while/if-else/jump (flow control) and load/store (data movement).

Data processing and flow control instructions manipulate the application data and provide the intended functionality of the application. On the other hand, data movement instructions, such as the load/store instructions, help move the application data to and from the memory. The execution of the data movement instructions results into memory requests being issued by the core of the IP to the assistance unit. The core translates the load and store instructions to issue read and write memory requests respectively.

The data processing and flow control instructions are typically executed in fixed number of clock cycles. However, data movement instructions require variable number of clock cycles depending on where the required data is stored. If the requested data is available with the assistance unit of the IP, the request is served within a few clock cycles. However, when the requested data is not available with the assistance unit, it (assistance unit) issues multiple external DRAM memory requests to the SoC infrastructure. The number of clock cycles required to retrieve the data from the external off-chip memory then depends on the service latency of the SoC infrastructure.

The performance of a functional module is directly dependent on the timely availability of the necessary data. If the SoC infrastructure takes a long time to fetch the necessary data from the external DRAM memory, the core of the IP could stall while waiting for the necessary data. The execution behaviour of the functional modules is therefore directly dependent on how external memory requests are served by the SoC infrastructure. Thus, the performance model for the functional modules captures both the execution behaviour and the performance dependence of the functional modules on the service provided by the SoC infrastructure.

Since we are primarily interested in the interaction of the functional module and the SoC infrastructure, we can abstract the execution of application code to the execution of the *external memory request trace*. As a result, we can model the execution behaviour of the functional modules by modeling the execution of the external memory request trace, as explained in the next subsection.

2.2.1 External memory request trace

The external memory requests issued by a functional module constitute the *memory request trace*. The memory request trace is derived from the actual execution of the application code on an IP which is coupled to an ideal SoC infrastructure. Since the application code typically involves complex conditional and unconditional branching, each of which generates a unique external memory trace, we assume that a given trace is derived from the actual execution of the application code (with a specific user input data set) on the given IP. The ideal SoC infrastructure is made up of a zero latency memory subsystem (including the DRAM memory) and a zero latency on-chip network. This assumption is essential since the issuance of an external memory request directly depends on the service provided by the SoC infrastructure to the previous requests. The assumption guarantees generation of a unique external memory trace for a give application code executing on an IP and decouples the SoC infrastructure from the characterization of the functional module. Along with the application code, the hardware capabilities of the IP, especially its ability to pipeline multiple memory requests, directly influences the issuance of memory requests and hence the external memory request trace. Based on their ability to pipeline external memory requests, the IPs can be classified as:

Blocking IP

IPs which stall immediately after issuing a memory request are termed as blocking IPs. A blocking IP can issue a maximum of one outstanding memory request (N_{max}).

Split IP

An IP is classified as a split IP if, unlike a blocking IP, it can proceed with further execution even after issuing a memory request. The split IP does not stall for the response of the issued request until it (response) is required by an instruction. The maximum number of outstanding requests (N_{max}) supported by a split IP is equal to one.

Pipelined IP (N_{max})

An IP is classified as a pipelined IP if it can issue multiple outstanding requests without stalling for the response of previously issued requests, unless an explicit dependence exists in the application code. A pipelined IP is, by definition, also split. The maximum number of outstanding requests (N_{max}) supported by a pipelined IP is called as its *pipeline depth* and is always greater than one.

Thus, the actual execution of the application code on a given IP coupled to an ideal SoC infrastructure and the hardware capabilities of the IP together determine the external memory request trace.

2.2.1.1 Specification of the Trace

A trace is primarily defined by the *inter-request intervals* of the external memory requests. The inter-request interval indicates the temporal distribution of all external

memory requests issued by the functional module. Since the trace is essentially derived from the execution of the application code, the inter-request interval indicates the number of clock cycles consumed by the data processing and flow-control instructions.

The trace can additionally, but not necessarily, specify the dependencies associated with memory requests. In the context of this work, we consider two dependencies, namely the *request dependence* and the *code dependence*. Request dependence exists between two memory requests, whereas code dependence exists between a request and a non memory movement instruction of the application. The inter-request interval and the request/code dependencies, which characterize and uniquely define a given trace, can be represented using different semantics. The semantics chosen to express these phenomena should allow their easy representation and manipulation. At the same time, they should express each of the phenomenon precisely without any contradictions or ambiguity. In our model, a memory request trace is completely specified with the following parameters:

Inter-request interval (T)

The inter-request interval (T_i) of request R_i is defined as the number of clock cycles required by the functional module to issue request R_{i+1} after request R_i has been issued, with the assumption that all requests are completely independent and served instantaneously by an ideal (zero latency) SoC infrastructure.

Request dependence (RD)

A request R_i has a request dependence associated with it, if a future request, say R_j ($j > i$) cannot be issued until the response for request R_i has been successfully received. The request dependence of a request R_i, i.e RD_i, specifies the number of requests which can be issued after issuing request R_i, without the need to wait for the response of request R_i to have been successfully received. In other words, it specifies the number of requests which can be issued after issuing the request R_i, such that it is guaranteed the core of the IP will not stall until the issuance of RD_i further requests. The functional module could however, stall before issuing these RD_i number of requests due to dependencies of other requests or the hardware limitation of the IP, but it is guaranteed that the functional module will never stall due to the request R_i itself. In our model, the request dependence (RD_i) of a request R_i is upper bounded by the hardware capability of the IP to issue a maximum of N_{max} outstanding requests, i.e $RD_i < N_{max}$. If the request dependence associated with request R_i exceeds N_{max}, i.e $RD_i \geq N_{max}$, then the dependence associated with request R_i is ignored and the request is considered as an independent request.

Code dependence (CD)

A request R_i has a code dependence associated with it, if a particular non memory movement instruction (such as add/subtract/if-else) cannot be executed until the response of request R_i is successfully received. This is observed in scenarios where the critical instruction requires the data returned by request R_i to perform the necessary data manipulation or take the appropriate control decision.

The code dependence of request R_i, i.e CD_i, specifies the number of clock cycles that the core of the functional module can execute after having issued RD_i further

requests, before it stalls for the requested data. The code dependence of a request R_i is thus bounded by the inter-request interval of request $R_{(i+RD_i)}$, which is $T_{(i+RD_i)}$.

A trace of size N can be exactly described by specifying three vectors, as shown in Equation 2.1

$$Trace(N) = \begin{cases} \text{Inter-request interval } [T_i] & \forall i \in [1,N], T_i \in [1,\infty] \\ \text{Request dependence } [RD_i] & \forall i \in [1,N], RD_i \in [-1, N_{max}-1] \\ \text{Code dependence } [CD_i] & \forall i \in [1,N], CD_i \in [-1, T_{(i+RD_i)}-1] \end{cases} \tag{2.1}$$

Along with the above definitions, the specification of request dependence (RD) and code dependence (CD) depend on the type of the IP and follow the semantics given in Table 2.2.

Table 2.2: Semantics for specifying RD and CD

Request characteristic	RD_i	CD_i
Independent R_i	-1	-1
R_j depends on R_i	$(j-i-1)$	$T_{(j-1)}-1$
R_i on blocking IP	0	0
R_i on split IP	0	$[-1,\ T_i-1]$
R_i on pipelined IP	$[0,\ N_{max}-1]$	$[-1,\ T_{(i+RD_i)}-1]$

2.2.1.2 Sample Traces

In this subsection, we examine a few traces typically generated by the following functional modules:

1. A processor executing a general purpose application (Figure 2.1(a))
2. A streaming video engine executing a real-time video application (Figure 2.2(b))

These examples illustrate how the execution of application code generates the memory request traces and how the dependencies within the application code get transferred to the memory request traces. The examples show how different memory request traces exist at different points within the SoC and how they are correlated to each other. Using these examples, we also highlight the specific memory request trace that we analyze while characterizing the functional modules.

2.2.1.3 A processor executing a general purpose application

Figure 2.3 presents a sample general purpose application code written in assembly language. Figure 2.4 shows the trace derived from the execution of the above application code on a simple processor employing a simple, conventional cache. It presents the trace

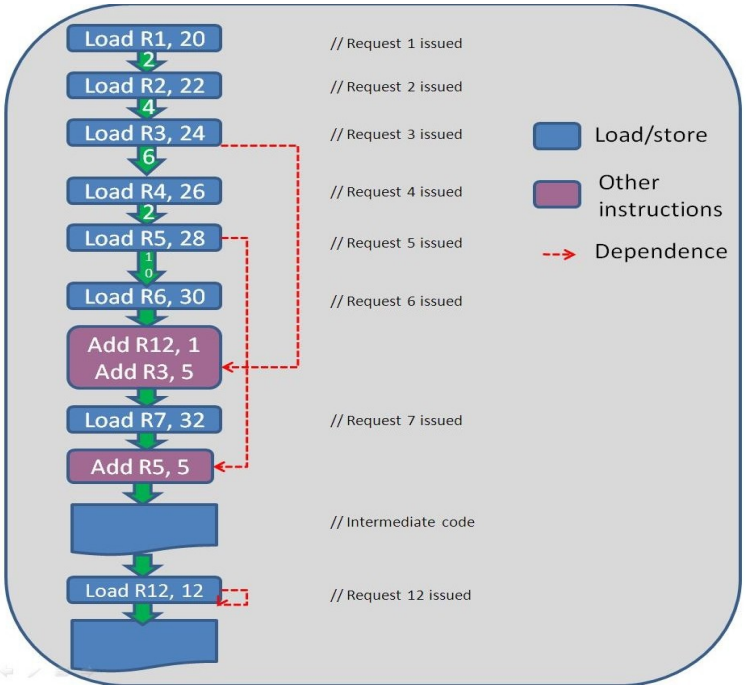


Figure 2.3: Assembly code of a task executing on the core IP.

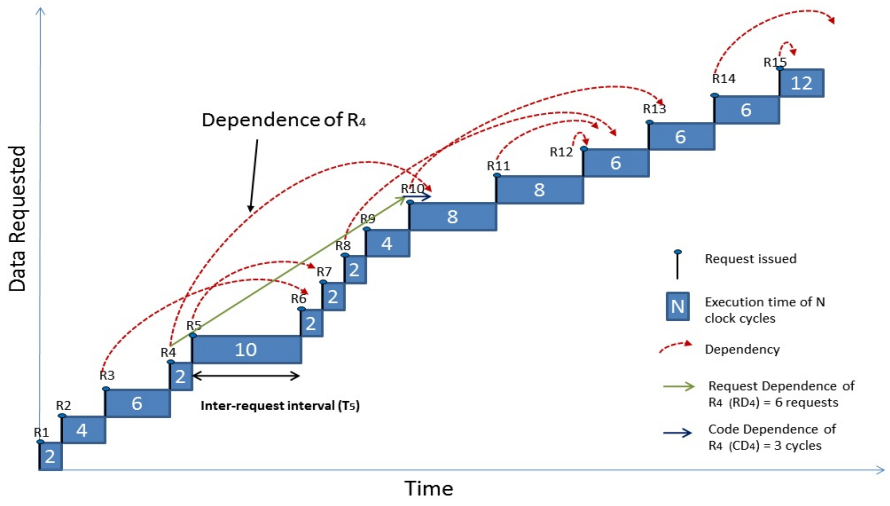


Figure 2.4: Memory trace resulting from executing audio application on a processor with an ideal memory subsystem.

of memory requests issued by the processor to the cache, observed at point A of the Figure 2.1(a). The mathematical representation of the trace is given by Equation (2.2).

$$Trace(15) = \begin{cases} \text{Inter-request interval} & [\ 2,\ 4,\ 6,\ 2,\ 10,\ 2,\ 2,\ 2,\ \ 4,\ 8,\ 8,\ 6,\ \ 6,\ 6,\ 12] \\ \text{Request dependence} & [\text{-}1,\ \text{-}1,\ 3,\ 6,\ 2,\ \text{-}1,\ \text{-}1,\ 4,\ \text{-}1,\ 3,\ 1,\ 0,\ \text{-}1,\ 3,\ 0] \\ \text{Code dependence} & [\text{-}1,\ \text{-}1,\ 1,\ 3,\ 0,\ \text{-}1,\ \text{-}1,\ 3,\ \text{-}1,\ 2,\ 1,\ 0,\ \text{-}1,\ 0,\ 0] \end{cases} \tag{2.2}$$

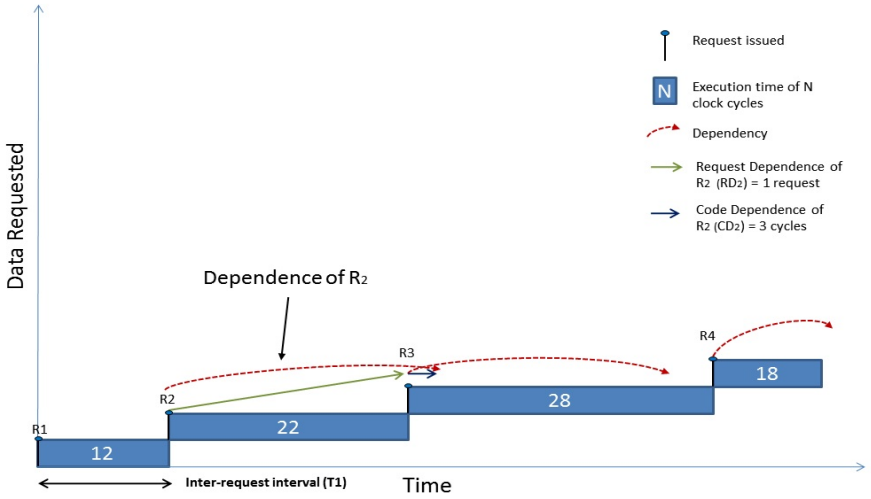


Figure 2.5: Memory trace generated by the cache (assistance unit) of the functional module.

Caches are designed such that most of the memory requests issued by the processor to the cache result in a cache hit. However, whenever there is a cache miss, the cache controller, on behalf of the processor, issues external memory requests to the SoC infrastructure. These memory requests constitute the trace observed at point B of Figure 2.1(a). It is clear that the trace generated by the cache controller consists only of external memory requests. This trace is essentially derived from the original trace issued by the processor to the cache, but has fewer requests. The trace generated by the cache controller is as shown in Figure 2.5. In this thesis, we are mainly concerned with the performance dependence of the functional modules on the service provided by the SoC infrastructure. To assess this performance dependence, we inspect and analyze only the external memory request trace of each functional module.

In the given example, requests R_1, R_4, R_{10} and R_{14} of the processor trace result into cache misses and subsequently get manifested as requests R_1, R_2, R_3 and R_4 respectively, in the trace generated by the cache controller. The cache controller trace is derived from the processor trace and hence, the two traces are inter-related. The issuing instants of the corresponding requests from both traces are assumed to be exactly the same, along with their dependencies. Although the cache trace is derived from the processor trace, the specifications of the two traces in our model are distinct. The processor trace (Figure 2.4) and the cache trace (Figure 2.5) are specified using Equation (2.2) and Equation (2.3) respectively.

$$Trace(4) = \begin{cases} \text{Inter-request interval} & [12,\ 22,\ 28,\ 18] \\ \text{Request dependence} & [\text{-}1,\ 1,\ \ 0,\ 1] \\ \text{Code dependence} & [\text{-}1,\ 3,\ 24,\ 8] \end{cases} \tag{2.3}$$

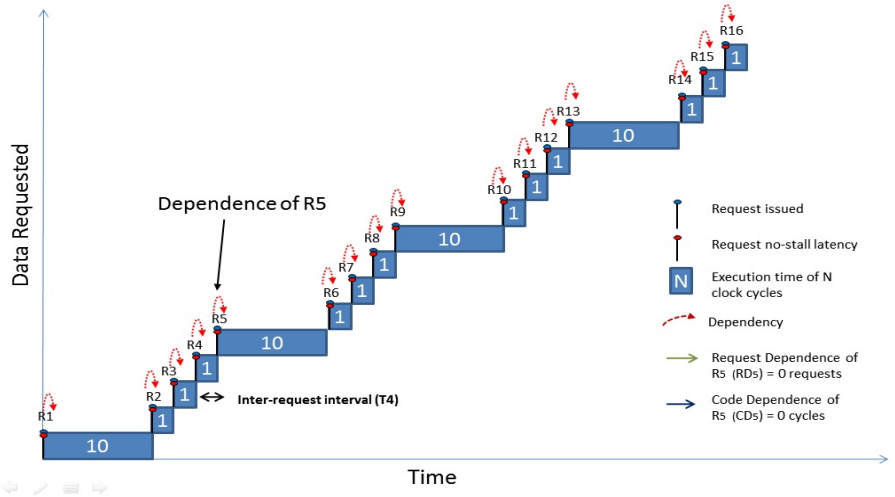


Figure 2.6: Memory trace resulting from the execution of real-time video application on a streaming graphics engine with an ideal memory subsystem.

2.2.1.4 A streaming video engine executing a real-time video application

For IPs like video engines or custom hardware accelerators, all memory requests are issued to the pre-fetch unit, which is responsible to have already fetched and stored the necessary data in its data buffer. These requests constitute a trace which is presented in Figure 2.6. It is representative of the trace typically observed at point A of Figure 2.2(b).

The pre-fetch unit, on the other hand, issues external memory requests to the SoC infrastructure so that it can pre-fetch the necessary data in advance. The pre-fetched data is temporarily held in the data buffers until it is eventually consumed by the core of the functional module. The external memory requests that are issued by the pre-fetch unit constitute the trace shown in Figure 2.7. It represents the trace observed at point B of Figure 2.2(b)

Similar to the previous example, the trace generated by the pre-fetch unit is related to the streaming video engine trace. The request R_2 of the pre-fetch unit trace requests for the data required by requests R_6, R_7, R_8 and R_9 of the streaming video engine trace. For simplicity, we assume that the instant the data is received by the pre-fetch unit, it is immediately available to the streaming video engine via the data buffers. Again, in this thesis, we are only interested in the external memory trace, which, in this example, is the pre-fetch unit trace.

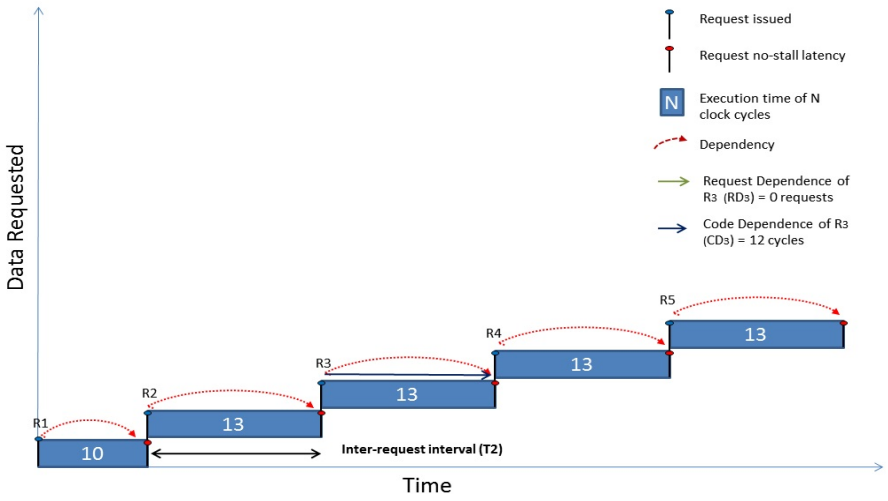


Figure 2.7: Memory trace generated by the pre-fetcher (assistance unit) of the functional module.

2.2.2 Execution behaviour of functional modules

To derive the high-level performance models of functional modules, it is necessary to capturing their execution behaviour. The execution behaviour of functional modules is influenced by two factors:

1. Performance of the SoC infrastructure
2. Performance of the IPs

2.2.3 Performance impact of the SoC infrastructure

The SoC infrastructure is composed of the on-chip network, the memory subsystem and the external DRAM memory. Thus, the performance of the SoC infrastructure is a combination of the performance of the on-chip network, the memory subsystem and the external DRAM memory. It is influenced by a variety of factors such as the quality of service (QoS) facilities it provides, the protocol it uses, arbitration policy for the NoC and memory controller, etc. The performance of the SoC infrastructure can be measured by different parameters such as the bandwidth, service latency etc. However, in this thesis, we are interested in the service latency aspect of the SoC infrastructure and thus relate the performance of the SoC infrastructure to its service latency. The service latency for any memory request issued to the SoC infrastructure involves the latency of the memory subsystem to retrieve the data from the external memory along with the latency of the on-chip network to transport the data from the memory subsystem to the functional module. This service latency is the *actual latency* required to serve the requests from the instant they are issued by the functional module. We formally define actual latency as follows:

Actual latency (*AL*)

1. Read Request: Actual latency (AL_i) of a read request R_i specifies the number of clock cycles required by the functional module to receive the entire requested data in its buffer, from the instant it issues request R_i.
2. Write Request: The actual latency (AL_i) of a posted write request R_i is considered to be zero since we do not wait for the acknowledgement associated with such write requests. However, for non-posted write requests, the actual latency (AL_i) specifies the number of clock cycles required by the functional module to receive an acknowledgement from the SoC infrastructure after successfully writing the data to the external memory.

In our model, the actual latency of the processor trace (Figure 2.4) is represented by a vector of trace size (N) as indicated by Equation (2.4).

$$\text{Actual Latency (N)} = [12, 10, 14, 14, 12, 10, 12, 12, 14, 14, 14, 12, 10, 8, 14] \quad (2.4)$$

2.2.4 Performance impact of the IPs

The IP executes the application code, thereby generating the external memory request trace. The performance of an IP is significantly affected by the amount of stalling it experiences. In fact, in this work, we determine the performance of a given IP by accounting all the stall cycles it experiences. Thus, to determine the performance impact of the IPs, we need to understand the phenomenon of stalling of the IP.

The stalling of the IP (or functional module) can be defined as a temporary suspension of code execution on the core of the IP. We assume that it is the core of the IP that executes the application code, thereby providing the expected functionality of the functional module. Due to the hardware capabilities of the IPs and the application code structure, all IPs experience some amount of stalling, which is acceptable. However, too much stalling of the IP core can prove detrimental to its performance. Thus, SoC designers must ensure that the stalling experienced by the IP core(s) is within the given bounds.

Reasons for the stalling of IP core

Typically, the core of the IP stalls whenever the assistance unit, in co-operation with the SoC infrastructure, is incapable of providing it with the necessary data before a fixed deadline. Thus, the stalling of the processing core is directly dependent on the performance of the assistance unit and the SoC infrastructure. The stalling of the IP cores is attributed to two fundamental restrictions imposed on the IP.

Hardware restriction

The hardware restriction is imposed on an IP by its pipeline depth. The pipeline depth of the IP is defined by its ability to issue a certain number of maximum outstanding requests (N_{max}). The ability of the IP to issue a maximum of N_{max} requests depends on the hardware support built into it to perform the necessary

bookkeeping of all in-flight requests. If the number of outstanding requests issued at any given time is equal to N_{max}, a new memory request cannot be issued until the response of a previous request is successfully received. This causes the processing core to stall even though there is no explicit dependence prohibiting the issuance of the next request.

Software restriction

The software restrictions are imposed on an IP by the request and code dependencies associated with the memory requests as explained in Chapter 1. During the execution of a given application on an IP core, if a request dependence or a code dependence (associated with a request) is encountered, the core of the IP stalls, irrespective of its hardware capability to issue further requests.

Along with the hardware restrictions mentioned above, the assistance unit can also stall due to the finite size of the pre-fetch data buffers. However, the stalling of the assistance unit does not necessarily lead to the stalling of the functional module.

Due to the stalling of the functional module, the service latency of the SoC infrastructure for memory requests, as experienced by the core of the functional module varies. To mathematically express the phenomenon of stalling, we introduce the following concepts:

1. **Total execution time (E)**
2. **No-stall interval (NI)**
3. **Perceived latency (PL)**
4. **Derived benefit (DB)**

2.2.5 Total execution time (E):

The total execution time (E) is defined as the number of clock cycles required by a given IP to execute the entire application code. The total execution time is a direct measure of the performance of an IP to execute a given application code. It is dependent on a variety of factors such as:

1. Application characteristics (e.g. length of application code, the number of memory movement instructions, etc.)
2. IP characteristics (e.g. latency tolerance of the processor/ IP, cache size, cache hit-rate, etc.)
3. SoC infrastructure characteristics (e.g. service latency of the SoC infrastructure, etc.)

The total execution time (E) of the base trace executed on a trace simulator coupled to an ideal (zero service latency) SoC infrastructure is defined as *base time* (E_0). It is mathematically defined by Equation (2.5).

$$E_0 = \sum_{i=1}^{N} T_i \tag{2.5}$$

For a SoC infrastructure with a non-zero service latency, the total execution time is generally greater than the base time.

2.2.6 No-stall interval

The no-stall interval (NI_i) of a request R_i is the time interval during which the core can continue with the application execution without stalling for the data requested by request R_i. It can also be seen as an allowance period given to the SoC infrastructure to produce the requested data, after the functional module issues the corresponding memory request. Every request R_i has a fixed no-stall interval (NI_i) which can be derived by analyzing the application code or the external memory request trace. Equation (2.6) can be used to determine the no-stall interval (NI_i) of a given request R_i.

$$NI_i = (\sum_{j=i}^{(i+RD_i)} T_j) + CD_i \tag{2.6}$$

2.2.7 Perceived latency

The perceived latency (PL_i) of a request R_i is the number of clock cycles for which the core of the functional module actually stalls while waiting for the response of request R_i. Here, we assume that the sole reason for the IP to stall is the inability of the SoC infrastructure to successfully produce the requested data within the no-stall interval of request R_i.

For a given application code, the total execution time of the functional module is at least equal to the base time (E_0). This generally happens when the functional module is coupled to an ideal (zero service latency) SoC infrastructure. For a SoC infrastructure with a non-zero service latency, the total execution time is greater than the base time if the non-zero service latency of the SoC infrastructure leads to the stalling of the functional module. In fact, the total execution time of a functional module can be determined by adding the base time (E_0) to the sum of perceived latencies of all external memory requests issued by the functional module. The total execution time (E) of a trace (size N, base time (E_0)), executing on an IP coupled to a SoC infrastructure with a non-zero service latency is given by using Equation (2.7).

$$E = E_0 + \sum_{i=1}^{N} PL_i \tag{2.7}$$

2.2.8 Derived Benefit

The perceived latency for a given memory request R_i does not account for the stalls caused by any of the previous or next ($N_{max} - 1$) requests. The stalling of the IP due to a given request provides additional no-stall interval to all the current outstanding

requests. This additional no-stall interval available to all outstanding requests is termed as *derived benefit*. The derived benefit is propagated to the neighbouring outstanding requests, both in the forward and the backward direction. In our model, we use the term derived benefit of request R_i, represented by DB_i, to denote the cumulative sum of derived benefits provided by all neighbouring requests of the given request R_i. In essence, the derived benefit of a given request is the sum of the perceived latencies of all neighbouring requests which overlap with the given request. The parameter of derived benefit is computed dynamically (at run-time) based on the knowledge of which requests overlap with the given request. The method of computing the derived benefit is presented as an algorithm explained in Appendix A.

Mathematically, perceived latency can be expressed in terms of the actual latency, the no-stall latency and the derived benefit using Equation (2.8).

$$PL_i = max(0, AL_i - NI_i - DB_i) \tag{2.8}$$

The no-stall interval, perceived latency, actual latency and the derived benefit together allow us to capture the execution of the external memory request trace. At the same time, these parameters capture the performance dependence of the functional modules on the service provided by the SoC infrastructure. Thus, by analyzing the external memory request trace with respect to these parameters, the execution behaviour of the functional modules can be captured in their performance models.

2.3 Performance modeling of functional modules

The functional module can be abstracted to a high-level model by capturing its execution behaviour and its performance dependence on the SoC infrastructure, especially its latency tolerance towards the service latency of the SoC infrastructure. The parameter of no-stall interval mainly quantifies and expresses the latency tolerance of functional modules. The model thus uses the parameter of no-stall interval for characterizing the functional module.

The advantage of using the no-stall interval to characterize the functional module is that the no-stall interval is solely dependent on the functional module (i.e the application code and the hardware capabilities of the IP). It is completely independent of the supporting SoC infrastructure and is also independent of other functional modules sharing the SoC infrastructure. Thus, the functional modules can be independently characterized at design time by using the parameter of no-stall interval.

2.4 Performance modeling of the SoC infrastructure

The performance model for the SoC infrastructure mainly captures the performance of the SoC infrastructure in terms of its service latency. The service latency of the SoC infrastructure for a given memory request is a combination of the latency of the memory subsystem to retrieve the data from the external memory along with the latency of the on-chip network to transport the data from the memory subsystem to the functional module. This service latency is the *actual latency* required to serve the requests from

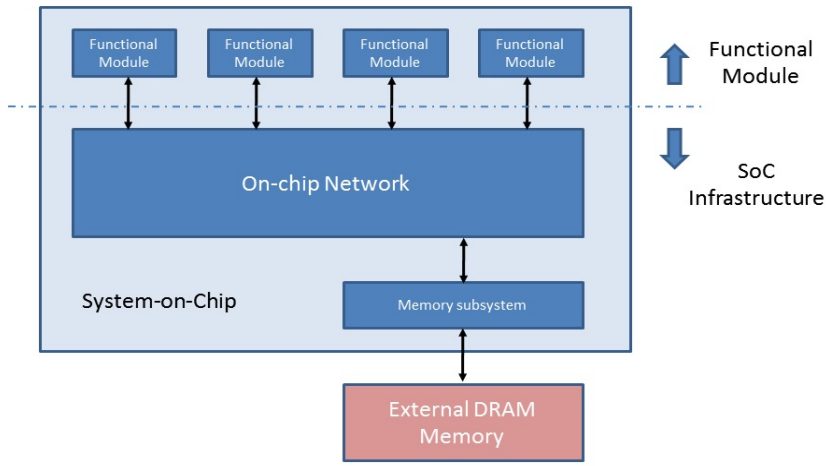


Figure 2.8: System level representation of the SoC

the instant they are issued to the SoC infrastructure by the functional modules. Thus, we model the performance of the SoC infrastructure by the parameter of actual latency, which is a direct measure of its service latency.

The actual latency of a SoC infrastructure is directly dependent on the characteristics of the on-chip network (topology, configuration, arbitration, QoS capabilities, etc.), the memory subsystem (configuration, arbitration, QoS capabilities) and the external DRAM memory (gross available bandwidth, etc.). It is also dependent on the request issuing behaviour and the performance of functional modules. The actual latency value is therefore determined either by measuring the actual latency for each external memory request issued by the functional module or by simply considering the average/ worst-case service latency of the SoC infrastructure. At times, the SoC designer can utilize statistical data and approximate the actual latency of individual requests by providing an average actual latency value along with the deviation (or spread) of the values over the length of a given trace.

2.5 The complete SoC model

The system on chip can be viewed as a collection of functional modules connected to each other and the SoC resources (like external memory, display system etc.) via the SoC infrastructure (Figure 2.8). Thus, the entire SoC can be modeled by connecting together the performance models of individual functional modules to the performance model of the SoC infrastructure. The performance model of the complete SoC is thus, built using the performance models of the functional modules and the SoC infrastructure as shown in Figure 2.9.

The interaction of the functional modules and the SoC infrastructure is captured using the perceived latency. The perceived latency indicates the number of stall cycles

experienced by the functional module while receiving service from the SoC infrastructure. It can also be viewed as a figure of merit for the performance of the functional module in association with a given SoC infrastructure.

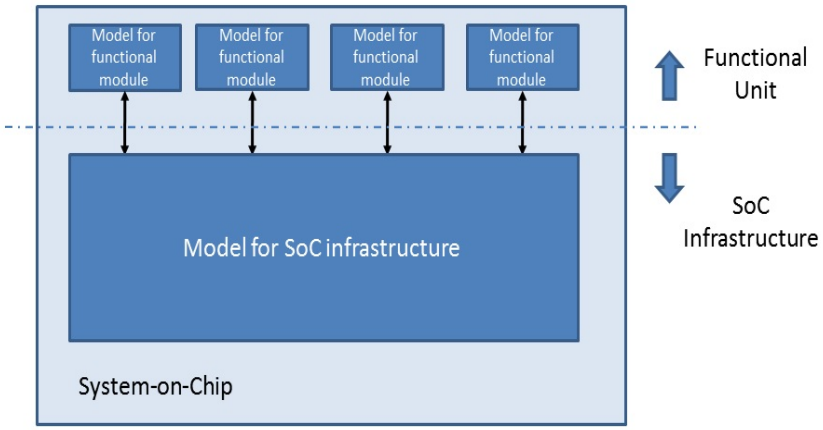


Figure 2.9: Model of the complete SoC

2.6 Basics of Performance Estimation

The high-level performance models discussed earlier are used for the performance estimation of the functional modules and the SoC infrastructure. Rapid performance estimation ultimately enables the SoC designer to efficiently integrate and verify the performance of IPs. Performance estimation can be achieved by using either a cycle-accurate trace simulator or a set of simplified equations based on average performance metrics. Both the approaches use the same performance models and hence, are conceptually identical. However, they differ in their complexity, accuracy, verification time and implementation effort.

Performance estimation requires the designer to execute a given task on a functional module coupled to a given SoC infrastructure and by some means, estimate its performance. As discussed previously, the execution of application code on an IP can be abstracted with the execution of its external memory request trace. The execution of the external memory request trace can be performed using either a trace simulator or can be approximated using a set of average performance values. Performance estimation is therefore performed in two steps:

1. Characterization of functional modules and the SoC infrastructure.

2. Execution of the external memory request trace.

2.6.1 Characterization of functional modules and the SoC infrastructure

Characterization of functional modules

The characterization of a functional module is performed differently depending on which performance estimation technique is used. In the trace-based performance estimation technique, the functional module is characterized by going through the (worst-case) application trace on a per-request basis and evaluating the no-stall interval of each request. To evaluate the no-stall interval of each request, it is also necessary to specify the hardware capabilities of the IP, mainly its ability to issue multiple outstanding requests N_{max}. The trace-based technique thus uses the application trace to characterize the functional module.

On the other hand, the simplified equation-based performance estimation technique utilizes average performance values for characterizing the functional module. These average performance values can be determined by analyzing the trace and extracting the average no-stall interval per request NI_{avg} and the spread of the no-stall interval NI_{spread} about the average value. They can also be determined by simply considering the worst-case values or making a calculated guess based on past experiences. The functional module can then be abstracted to a black box characterized by the parameters of average no-stall interval (NI_{avg}) and its spread (NI_{spread}).

The parameter of no-stall interval characterizes the latency tolerance of the functional module. A high value of average no-stall interval suggests that the functional module can tolerate a large service latency per request until it finally stalls for the requested data. At the same time, the spread in no-stall interval suggests how far the no-stall intervals of individual requests deviate from the average value. A large value of spread in no-stall interval indicates that the no-stall interval of individual requests varies significantly about the average value, possibly due to the bursty request issuing nature of the functional module. Likewise, a small value of spread in no-stall interval is indicative of a more regular request issuing behaviour of the functional module.

Characterization of the SoC infrastructure

The SoC infrastructure is characterized by the parameter of actual latency. For characterizing the SoC infrastructure, we assume that each request issued to the SoC infrastructure is served with a given average actual latency (AL_{avg}). Along with the average actual latency, we also utilize the spread in actual latency (AL_{spread}). The average actual latency value is determined either by using statistical data or by simply considering the worst-case service latency, whereas the spread is mainly determined by using statistical data.

The average actual latency (AL_{avg}) indicates the average service latency of the SoC infrastructure to serve any given external memory request. The spread of actual latency (AL_{spread}) indicates the deviation in the service latency provided to individual memory requests as compared to the average actual latency value. A high value of spread indicates that the SoC infrastructure is not consistent in providing service to individual memory requests. It could also indicate that the functional modules coupled to the given SoC infrastructure issue requests for localized data in a bursty manner, due to which some

requests are served with a large service latency, whereas others are served with a relatively small service latency.

The functional module and the SoC infrastructure are characterized and abstracted to their respective performance models independently. For a given functional module, the characterization step is performed only once. The performance model of the functional module can then be used over multiple SoC designs. This is particularly useful from the design re-use point of view. The performance model of the SoC infrastructure captures the performance dependence of the SoC infrastructure (on parameters like burstiness, bandwidth, request behaviour, priority, etc.) into a single parameter of actual latency, thereby simplifying the performance estimation process. The characterization of functional modules and the SoC infrastructure is essential in both the performance estimation techniques.

2.6.2 Execution of the external memory request trace

After the functional module and the SoC infrastructure are characterized, the external memory request trace is executed to estimate the performance of the functional module when it is coupled to the given SoC infrastructure. The execution of the memory request trace determines the derived benefit and the perceived latency of each request. The individual derived benefit values allow us to determine the average derived benefit (DB_{avg}) and its spread (DB_{spread}). Likewise, the individual perceived latency values allow us to determine the average perceived latency (PL_{avg}).

The average perceived latency (PL_{avg}) is a critical parameter determining the performance of functional modules and the SoC infrastructure. It specifies the average number of stall cycles encountered by the functional module per request, when it is coupled to the given SoC infrastructure. A high value of average perceived latency indicates that the functional module stalls for a considerable amount of clock cycles per request, which might not be tolerable to a real-time application. If the given SoC infrastructure fails to provide the expected quality of service to the functional module, it is reflected by the average perceived latency value of the worst-case application trace exceeding some pre-defined threshold value. The average perceived latency value thus, instills confidence in the SoC designer regarding the performance of the complete SoC in the early design stages.

2.7 Conclusion

Using the performance models, we aim to characterize the execution behaviour of functional modules and the SoC infrastructure. Such a characterization of execution behaviour eventually allows us to correlate the performance of functional modules with the service latency of the SoC infrastructure. Once the high-level models of the functional modules are derived, they can be re-used over multiple SoC designs, thereby promoting IP re-use. These high-level performance models of functional modules and SoC infrastructure play a central role in the proposed performance estimation techniques (explained in Chapter 3 and Chapter 4). In the next chapter (Chapter 3), we study the trace-based performance estimation technique.

Trace-based Performance Estimation

3

In Chapter 2, we presented the high-level performance models of functional modules and the SoC infrastructure. These high-level performance models play a central role in the trace-based and the equation-based performance estimation techniques. In Chapter2, we also explained in detail the basics of the proposed performance estimation techniques. In this chapter, we study the trace-based performance estimation technique in detail.

The trace-based performance estimation technique is one of the two proposed performance estimation techniques. It utilizes a trace simulator to simulate the execution of the external memory request trace. In this estimation technique, we characterize the functional module by analyzing the entire external memory request trace on a request-by-request basis. The trace analysis allows us to extract the no-stall interval of individual memory requests. Finally, we estimate the performance of the functional modules by simulating the execution of their external memory request traces on the trace simulator.

3.1 Trace simulator

To simulate the execution of the external memory request trace, a cycle accurate trace simulator was implemented in Matlab. The simulator uses the high-level performance models of the functional modules and the SoC infrastructure introduced in Chapter 2. The input to the trace simulator is an external memory request trace that is generated from the execution of the application code on an IP that is coupled to an ideal SoC infrastructure. The trace is specified according to the semantics explained in Chapter 2. The external memory request trace represents the application code at a higher abstraction level. The trace simulator can be configured to execute a given memory request trace as either a blocking, split or pipelined IP. The simulator outputs the estimated performance (in terms of the perceived latency (PL)) of the functional module in association with the given SoC infrastructure.

In Section 3.2, we present the experiments performed using the trace simulator. The experiments offer significant insights into the execution behaviour of the functional modules and their performance dependence on the service provided by the SoC infrastructure.

3.2 Experiments based on the trace simulator

To illustrate the performance estimation technique using the trace simulator discussed in the Section 3.1, we perform a set of experiments. The experiments start with illustrating the performance estimation technique for simple blocking processors and later deal with the more complicated pipelined processors.

3.2.1 Simple Blocking Processor

The experiment involves using a standard instruction set simulator (Simit-ARM) to execute a typical audio decoder application so as to compare its performance results with that of the trace simulator executing the external memory request trace of the same application. The goal of this experiment is to illustrate that the execution of application code on a given IP can be abstracted to the execution of its external memory request trace on a simple trace simulator. To highlight the similarity in the temporal behaviour of the two simulators, we measure and compare the total execution time of the audio decoder application (executing on the Simit-ARM simulator) with the total execution time of its external memory request trace (executing on the trace simulator).

Simit-ARM simulator:

The Simit-ARM simulator [1] takes the compiled binary of an audio decoding application and executes it instruction by instruction. Whenever a load/store instruction is executed, the Simit-ARM simulator issues read/write requests to its bus interface unit (BIU) which is also a part of the simulation environment. The bus interface unit mimics the behaviour of the memory subsystem that serves all read/write requests with a fixed read/write service latency respectively. The bus interface unit parameters, i.e the read/write service latency, can be specified for every simulation run.

Trace simulator:

The trace simulator uses the performance models of a blocking processor and the SoC infrastructure. Every external memory request issued by the blocking processor is considered to have a no-stall interval (NI_i) of zero clock cycles. The no-stall interval of zero clock cycles is in accordance with the behavior of blocking IPs that stall immediately after issuing a memory request. The external memory requests issued by the blocking processor are served by the SoC infrastructure that is abstracted to a black box characterized by its service latency. For the sake of simplicity, it is considered that the SoC infrastructure serves all external memory requests with a fixed service latency of AL_{avg}. The SoC infrastructure service latency, i.e. the average actual latency (AL_{avg}) can be specified for every simulation run.

The experiment is performed in two stages.

- **Stage 1**

  In the first stage, the audio decoder application is executed on the Simit-ARM simulator coupled to an ideal memory subsystem. The Simit-ARM simulator simulates the StrongARM 1100 processor core. It mimics the behaviour of a blocking processor that stalls immediately after issuing an external memory request. The read/write latency of the bus interface unit coupled to the Simit-ARM simulator is set to zero so as to extract the external memory trace without the influence of the memory subsystem latency. The external memory request trace generated from the execution of audio decoder application code on the Simit-ARM simulator coupled to a perfect memory subsystem is denoted as the *base trace*.

- **Stage 2**

  In the second stage, the base trace (generated in stage 1) is executed on the trace simulator. It is only by executing the external memory request trace that we can estimate the performance of the functional module coupled to a given SoC infrastructure. Thus, stage 2 is responsible for estimating the performance of the functional modules for the different SoC infrastructure configurations. Since we are interested in estimating the performance of functional modules when they are coupled to SoC infrastructures with varying configurations, we repeat the simulation for varying values of AL_{avg}. At the same time, the Simit-ARM simulator simulates the application execution for the same set of bus interface unit (BIU) configurations. While the simulations on the Simit-ARM simulator are performed by changing the BIU's read/write latencies, the simulations on the trace simulator are performed by changing the value of AL_{avg} associated with the SoC infrastructure.

Results:

The results of the experiment are presented in Table 3.1 and are also plotted in a graph of total execution time v/s actual latency (Figure 3.1). Table 3.1 presents the total execution time of the Simit-ARM simulator and the trace simulator, for varying values of their memory subsystem service latencies (AL_{avg}). The table also presents the difference in the execution times of the two simulators and highlights the percentage error introduced thereof. It can be observed from the table that the percentage error is less than 1%.

Table 3.1: Simit-ARM v/s Trace simulator results

Memory subsystem service latency (clock cycles)	*Simit-ARM Execution Time (clock cycles)*	*Trace simulator Execution Time (clock cycles)*	*Difference (clock cycles)*	*Percent Error (%)*
0	39107	38844	263	0.67
5	150216	150576	360	0.21
10	267310	267726	416	0.15
15	384440	384876	436	0.11
20	501570	502026	456	0.09
30	735830	736326	496	0.06
40	970090	970626	536	0.05
50	1204350	1204926	576	0.04
70	1672870	1673526	656	0.03

The graph, shown in Figure 3.1 plots:

1. The total execution time (in clock cycles) of the application code executed on the Simit-ARM simulator against the service latency of the BIU

2. The total execution time of the base trace executed on the trace simulator against the actual latency of the SoC infrastructure coupled to it.

It can be seen from the plot that the results for the Simit-ARM simulator and the trace simulator are perfectly matched, due to which, the two curves overlap each other. Also, the total application execution time increases linearly with the increasing service latency of the SoC infrastructure. This is intuitively correct since the higher the service latency of the SoC infrastructure, the longer it takes for the IP to execute the given application code. The linear increase in the total execution time of the blocking processor suggests that the memory latency is a highly dominant factor in the performance of blocking processors.

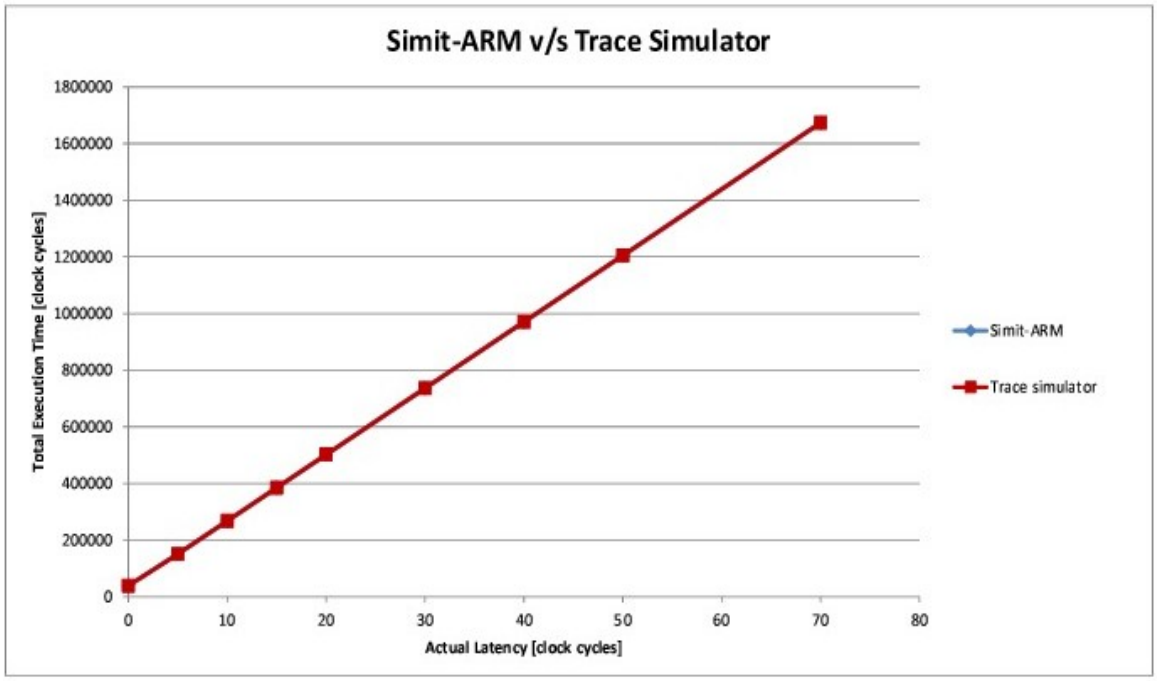


Figure 3.1: Simit-ARM experiment

Conclusions

The results produced by the trace simulator are very close (1% error) to those produced by the Simit-ARM simulator. The high level of similarity observed in the results of the two simulators illustrates that *the execution of application code on the Simit-ARM simulator (ISS) is identical to the execution of the external memory request trace on the trace simulator when both the simulators are coupled to identical memory subsystems.* In other words, the temporal behaviour of an IP executing a given application code is equivalent to the temporal behaviour of the trace simulator executing its base trace.

The base trace can therefore be repeatedly used to emulate the application execution for various SoC infrastructure configurations. This offers the advantage of extracting the trace only once and later, re-using the same trace to perform multiple trace simulations (with varying SoC infrastructure configurations) for aiding the design space exploration process. Also, the simulation of traces is much faster and computationally cheaper than simulating the entire application on instruction set simulators/ HDL models.

The results also prove that *the performance model of the blocking processors and the SoC infrastructure captures their execution behaviour as accurately as their true simulators*. It is important to note that Simit-ARM uses a constant read/write latency for all external memory request (independent of the address, request behaviour of the processor, etc.) which is similar to the constant actual latency value used by the trace simulator. This is one of the important reasons for the similarity in results of the two simulators. In reality, the service latency of the SoC infrastructure varies due to a range of factors (such as addresses of memory requests, issuing behaviour of IPs, etc.) and SoC designers using this performance estimation technique should appropriately adjust the average actual latency (AL_{avg}) values to account for these influences.

3.2.2 Experiments for split and pipelined IPs

The following experiments illustrate the performance modeling of split and pipelined processors and provide insights into their execution behaviour. The pipelined processors are more complex to model since they can issue N_{max} outstanding requests, thereby making their execution behaviour difficult to capture in a model. The experiments are conducted using synthetically generated traces. In real SoC design environments, the traces would be generated by the actual execution of the application code on either the actual split/pipelined processor or its instruction set simulator. Once the base trace is extracted from the application code, it can be used for characterizing the split/pipelined processors to generate their high-level models.

High-level performance model of pipelined processors

We investigate three experiments involving split and pipelined processors. The experiments focus on analyzing the execution behaviour of the split/pipelined processors and estimating the effect of various parameters on their performance. We study the following three experiments:

1. Experiment to compare the performance of split and pipelined processors to that of the blocking processors.

2. Experiment to analyze the effect of the spread in no-stall interval on the performance of split/pipelined IPs.

3. Experiment to analyze the effect of the spread in actual latency on the performance of split/pipelined IPs.

3.2.2.1 Experiment to compare the performance of split and pipelined processors to that of the blocking processors

The experiment involves simulating the execution of a synthetically generated trace on the trace simulator configured for blocking, split and pipelined processors. The simulations are performed for varying SoC infrastructure configurations. From this experiment, we aim to understand and compare the execution behaviour of the blocking, split, pipelined IPs while executing the same application code for varying SoC infrastructure service latency values. Furthermore, to study how the ability to issue multiple outstanding requests affects the performance of pipelined IPs, we simulate the given trace on three different pipelined processors, each differing in its ability to issue the maximum number of outstanding requests (N_{max}).

The results of the experiment are as follows:

1. Blocking processor

   The total execution time and the spread in perceived latency of a blocking processor for varying values of average actual latency are presented in Table 3.2. Similar to the results observed in the Simit-ARM experiment, the total execution time of the blocking processor increases linearly with the increasing service latency of the SoC infrastructure. At the same time, the average perceived latency of the blocking processor also increases linearly with the SoC infrastructure's service latency. In fact, the average perceived latency is equal to the average actual latency, as can be derived from Equation (2.8). The trend of the processor's average perceived latency (PL_{avg}) along with the changing SoC infrastructure service latency is plotted in Figure 3.2.

Table 3.2: Blocking processor results

Actual Latency (clock cycles)	*Total Execution Time (clock cycles)*	*Average perceived latency (clock cycles)*
5	750048	5
10	1000048	10
15	1250048	15
20	1500048	20
25	1750048	25
30	2000048	30
35	2250048	35
40	2500048	40
45	2750048	45
50	3000048	50

2. Split processor

   The total execution time and the perceived latency of a split processor for varying values of average actual latency are presented in Table 3.3. Figure 3.3 presents the plot of the total execution time of split processors against the average actual

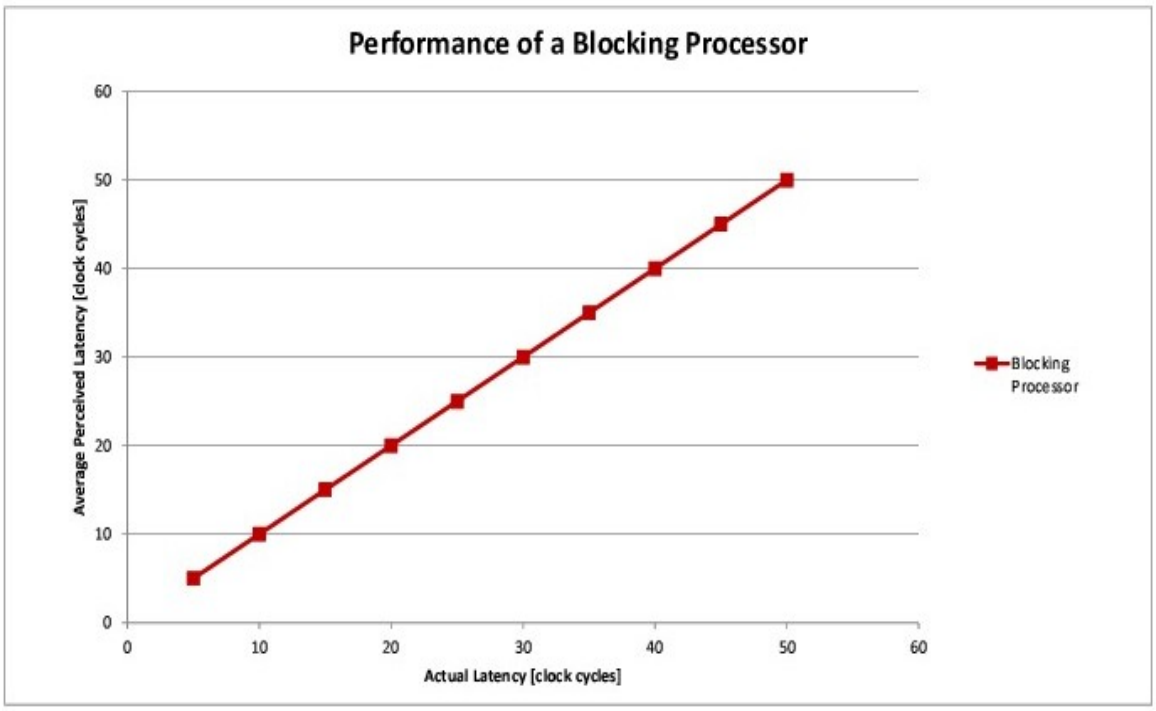


Figure 3.2: Performance of blocking processor

latency of the SoC infrastructure. The plot illustrates that the total execution time of the split processor is almost equal to its base time as long as the average actual latency of the SoC infrastructure is less than the processor's average no-stall interval. It implies that the average perceived latency of the split processor is equal to zero clock cycles (i.e the split processor experiences no stalling) while the average actual latency is less than the processor's average no-stall interval.

$$PL_{avg} \approx 0 (when AL_{avg} \leq NI_{avg}) \tag{3.1}$$

After the average actual latency of the SoC infrastructure exceeds the average no-stall interval of the processor, the total execution time as well as the average perceived latency experienced by the processor increases linearly with the average actual latency. The experiment therefore illustrates that *as long as the average actual latency of the SoC infrastructure is less than the average no-stall interval of a functional module (having negligible no-stall interval spread), the functional module experiences negligible stalling, thereby having no negative influence on its performance.* Also, *after the average actual latency of the SoC infrastructure exceeds the average no-stall interval of the functional module, the average perceived*

latency experienced by the functional module increases linearly with the average actual latency.

Table 3.3: Split processor results

Actual Latency (clock cycles)	*Total Execution Time (clock cycles)*	*Average perceived latency (clock cycles)*
5	500048	0
10	553619	1.07
15	800000	5.99
20	1050000	10.99
25	1300000	15.99
30	1550000	20.99
35	1800000	25.99
40	2050000	30.99
45	2300000	35.99
50	2550000	40.99

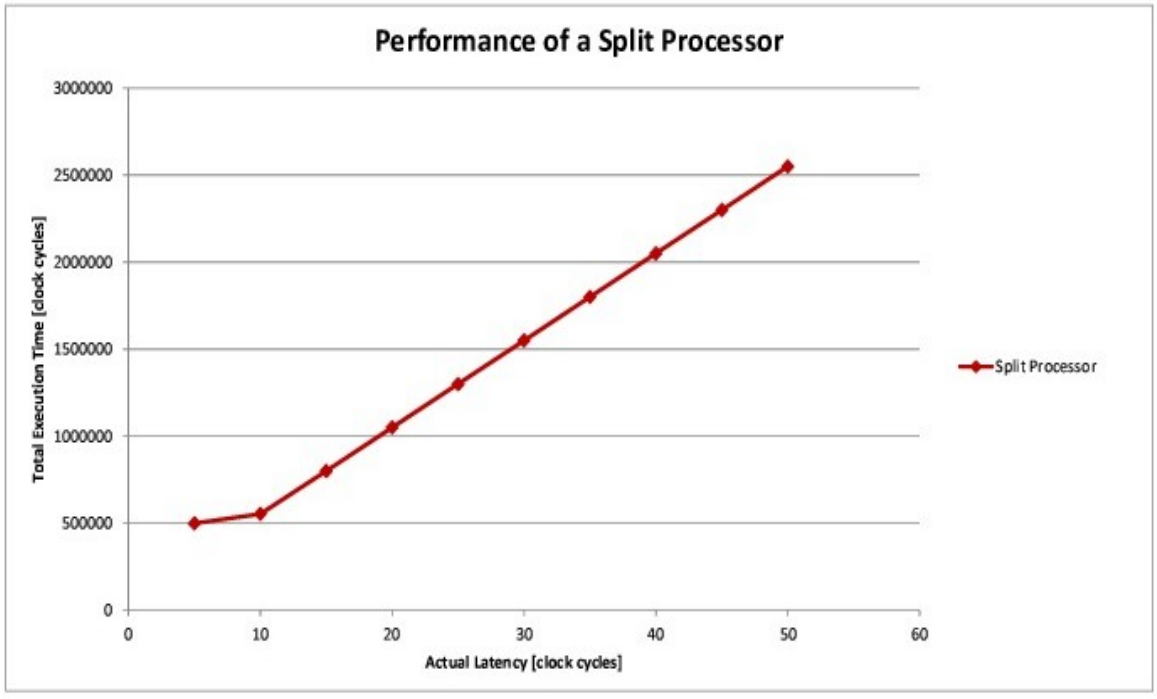


Figure 3.3: Performance of split processor

3. Pipelined processor

In this experiment, we investigate three different pipelined processors having the hardware capabilities of issuing a maximum of 2, 4 and 8 outstanding requests, respectively. The no-stall interval of a functional module depends on the application code and the hardware capabilities of the IP, specifically its ability to issue N_{max} outstanding requests (Equation (2.6)). Thus, although the same external memory request trace is used for the simulation of all three pipelined processors, their hardware capabilities differ and hence, they are characterized by different no-stall interval values. The average no-stall interval of the external memory request trace executing on the pipelined processors with N_{max} of 2, 4 and 8 requests is 11, 24 and 32 clock cycles respectively.

Table 3.4: Pipelined processor results

	N_{max}=2		N_{max}=4		N_{max}=8	
AL_{avg}	*Ex Time*	PL_{avg}	*Ex. Time*	PL_{avg}	*Ex. Time*	PL_{avg}
5	511351	0.22	511304	0.22	511224	0.22
10	572127	1.44	550883	1.01	550803	1.01
15	724928	4.49	610923	2.21	610834	2.21
20	890501	7.82	688707	3.77	688603	3.77
25	1064839	11.41	776505	5.53	776386	5.53
30	1239505	15.01	870724	7.41	870590	7.41
35	1414185	18.62	968197	9.39	968044	9.39
40	1588865	22.22	1067614	11.47	1067016	11.44
45	1763545	25.83	1168715	13.64	1166654	13.52
50	1938225	29.43	1270218	15.83	1266430	15.61

The difference in the total execution times of the trace on the three pipelined processors, with the changing SoC infrastructure configuration is shown in Figure 3.4. It is evident from the plot that the performance of pipelined processors is similar to that of the split processors. It is similar to the split processors because as long as the average actual latency of the SoC infrastructure is less than the average no-stall interval of the pipelined processor and there is negligible spread in the no-stall interval values, the average perceived latency experienced by the processor is negligible. When the average actual latency of the SoC infrastructure exceeds the processor's average no-stall interval, the perceived latency increases linearly with the actual latency.

Table 3.4 presents the performance results of the three pipelined processors. It is evident from the presented results that *the total execution time of the external memory request trace decreases with the increasing ability of the pipelined processors to issue multiple outstanding requests*, it being the lowest for N_{max}=8 and highest for N_{max}=2. This is intuitively correct since a pipelined processor which has the ability to issue more outstanding requests stalls for fewer clock cycles than the one which issues fewer outstanding requests.

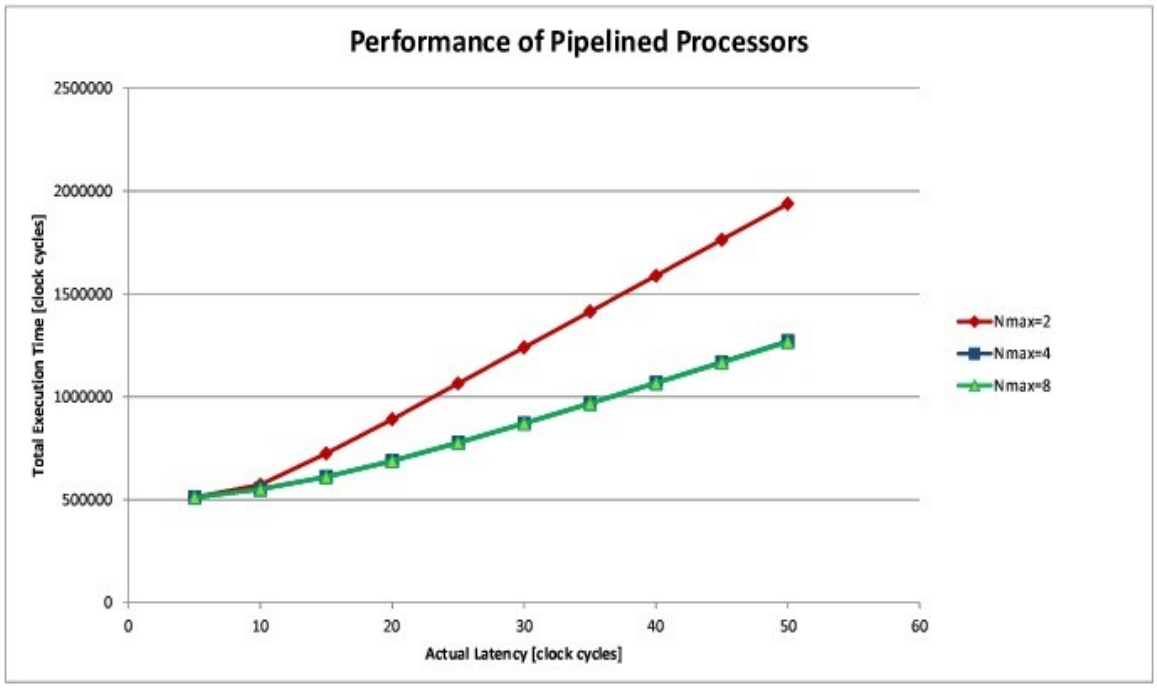


Figure 3.4: Comparison of the performance of pipelined processors

It is important to note that the external memory request trace under consideration has inherent request and code dependencies up to 4 requests. Hence, the improvement in the total execution time of the external memory request trace on pipelined processors with a pipeline depth larger than 4 requests is limited by the dependencies inherent within the application code. Thus, while the pipelined processors with N_{max} of 2, 4 and 8 outstanding requests exhibit a trend of improving performance, the performance gains reduce significantly once the inherent code dependencies start affecting the processor's requesting behaviour.

Comparison of the performance of all processors

As it can be clearly seen from Figure 3.5, blocking processors require the largest execution time for any given value of average actual latency. Since the no-stall interval of all requests issued by the blocking processor is zero, the total execution time and the perceived latency of blocking processors increases linearly with the increasing memory subsystem latency. As compared to blocking processors, split processors exhibit lower execution times for all values of the SoC infrastructure's service latency. This is attributed to the ability of split processors to hide a part of the actual latency with the

inter-request interval of their requests. However, split processors also exhibit a linear increase in the total execution time after the SoC infrastructure's service latency exceeds their average no-stall interval. The pipelined processors exhibit the best performance among all processors mainly due to their ability to hide most of the actual latency associated with a request with that of multiple overlapping requests. As proposed in the model, a pipelined processor with the ability to issue a higher number of outstanding requests performs better than the one with an ability to issue lesser number of outstanding requests.

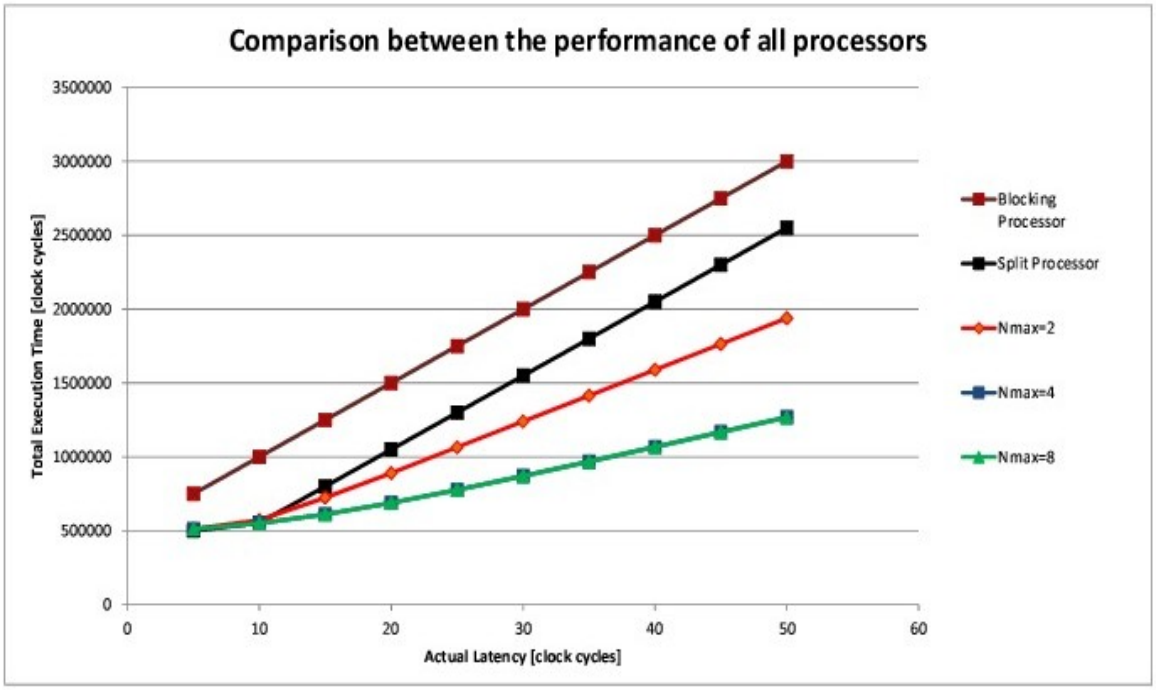


Figure 3.5: Comparison between performance of all processors

Conclusions

The simulation results illustrate that the execution behaviour and the performance of different IPs follow the assumptions made in the model. Also, it shows that the performance of a given application is limited earlier by the hardware restrictions (e.g. blocking processor) and later by the software restrictions (e.g. code dependencies affecting pipelined IPs with $N_{max} > 4$). The experiments illustrate that the proposed performance estimation technique can not only be used to estimate the performance of functional modules across a range of SoC infrastructure configurations, but can also

be used to configure the pipeline depth of IPs that best matches the application under consideration.

3.2.2.2 Experiment to analyze the effect of the spread in no-stall interval on the performance of split/pipelined IPs

This experiment aims to illustrate the effect of spread or deviation in the no-stall interval of individual requests, on the performance of functional modules employing split/ pipelined IPs. The spread in no-stall interval values indicates the deviation of no-stall interval of individual requests in comparison to the average no-stall interval value of the entire memory request trace. If the SoC infrastructure is designed such that it provides a consistent service, indicated by a negligible spread in the actual latency of memory requests, the spread in no-stall interval of individual memory requests starts affecting the performance of the functional module. This is because, if the average actual latency of the SoC infrastructure is guaranteed to be lower than the average no-stall interval of the given functional module by a fixed number of clock cycles, the SoC designer would expect the average perceived latency, i.e the average number of stall cycles encountered per request, to be zero. However, a high spread in the no-stall interval values would imply that out of all the memory requests, some requests have a significantly higher no-stall interval while others have a significantly lower no-stall interval value. While requests having a significantly high no-stall interval do not affect the performance of the functional module, the requests having a significantly low no-stall interval stall the functional module, thereby degrading its performance.

To understand the phenomenon, we simulate three synthetically generated external memory request traces on split/pipelined IPs. The average no-stall interval of the three traces is 20 clock cycles. The spread in no-stall interval for the three traces is 0, 5 and 10 clock cycles respectively. The average actual latency of the SoC infrastructure is varied from 0 to 50 clock cycles

Results

Figure 3.6 presents a plot of the total execution time of the three traces executing on a split IP against the average actual latency of the SoC infrastructure. From the plot, we observe that the total execution time of the trace increases with the increasing spread in no-stall interval values, but only when the average actual latency is close to the average no-stall interval value. At other values of average actual latency, the total execution time of the trace is the same, irrespective of the spread in no-stall interval values. The increase in the total execution time indicates a degradation in the performance of the split IP due to the increasing spread in the no-stall interval value.

Conclusions

This experiment analyzes the effect of the spread in no-stall interval on the performance of functional modules employing split/ pipelined IPs. It is clear that *the spread in the no-stall interval values degrades the performance of split/pipelined IPs only when the average actual latency of the SoC infrastructure is reasonably close to the average no-stall interval of the functional module.* At the same time, *the spread in no-stall interval*

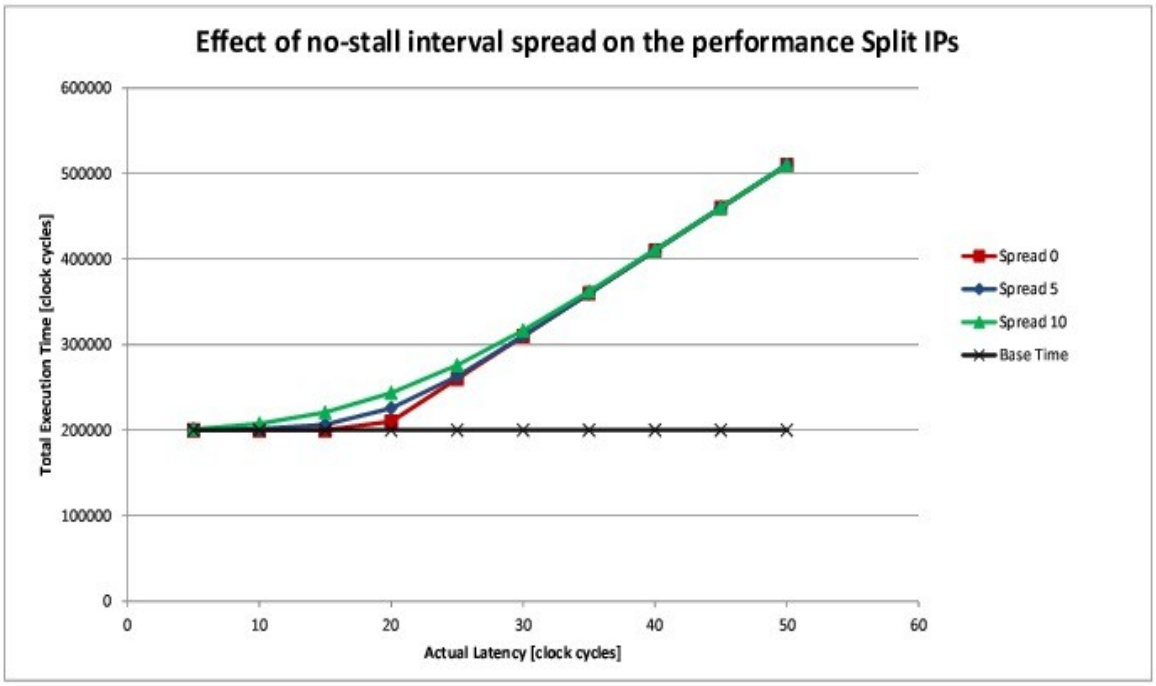


Figure 3.6: Effect of the spread in no-stall interval on the performance of split/pipelined IPs

values does not have any bearing on the performance of the functional module when the average service latency of the SoC infrastructure is either much smaller or much larger than the average no-stall interval of the functional module.

3.2.2.3 Experiment to analyze the effect of the spread in actual latency on the performance of split/pipelined IPs.

This experiment aims to illustrate the effect of the spread in actual latency of the SoC infrastructure on the performance of functional modules employing split/pipelined IPs. The spread in the actual latency of the SoC infrastructure indicates the deviation observed in the actual latency of individual requests, as compared to the average actual latency of the SoC infrastructure. It is indicative of the consistency of service provided by the SoC infrastructure. The spread in actual latency affects the performance of a functional module, since requests which are served with an actual latency lower than their no-stall interval do not lead to performance gains, but requests which are served with an actual latency greater than their no-stall interval lead to the stalling of the functional module.

To understand this phenomenon, we simulate a synthetically generated external memory request trace with an average no-stall interval of 20 clock cycles on blocking, split and pipelined IPs. The simulations are carried out while keeping the average actual latency of the SoC infrastructure constant at 20 clock cycles, while varying the spread in actual latency from 0 to 20 clock cycles. The average actual latency of the SoC infrastructure is intentionally set to the average no-stall interval of the functional module. This is because, similar to the spread in no-stall interval, the spread in actual latency affects the performance of the functional module only when it is reasonably close to the average no-stall interval value.

Results

Figure 3.7 shows a plot of the total execution time of the trace executing on blocking, split and pipelined IPs against the spread in actual latency of the SoC infrastructure. From the plot, we observe that the total execution time of the trace increases for all three IPs- blocking, split and pipelined. The total execution time of the trace executing on the pipelined IP increases by approximately 20% when the spread in actual latency is increased from zero to its maximum value. At the same time, the total execution time for the trace executing on the split IP increases by approximately 37% when the spread is similarly increased to from zero to its maximum value. The increase in the total execution time indicates a degradation in performance of the blocking, split and pipelined IPs due to the increasing deviation in the service provided by the SoC infrastructure.

Conclusions

As expected, the performance of the blocking IP is least affected by the increasing spread in actual latency of the SoC infrastructure. However, it is interesting to note that the pipelined IPs are more resilient to the increasing actual latency spread as compared to the split IPs. While there is a linear degradation in the performance of the split IPs, the percentage loss in the performance of pipelined IPs is much lesser. This is because pipelined IPs overlap multiple external memory requests and are thus able to compensate better for the deviation in the SoC infrastructure's service. However, split IPs only hide a part of the actual latency using the inter-request interval. This does not allow the split IPs to sufficiently compensate for the fluctuations in the service latency of the SoC infrastructure. Thus, the split IPs are affected to a greater extent by the spread in actual latency. From this experiment, we thus conclude that *the pipelined IPs are more tolerant to the spread in the actual latency of the SoC infrastructure than the split IPs.*

3.3 Conclusion

In this chapter, we studied the trace-based performance estimation technique. The chapter presented several experiments conducted using the trace simulator which offer a deeper insight into the execution behaviour of different types of IPs and how their performance is affected by the high-level model parameters. In Chapter 4, we study the simplified equation-based performance estimation technique.

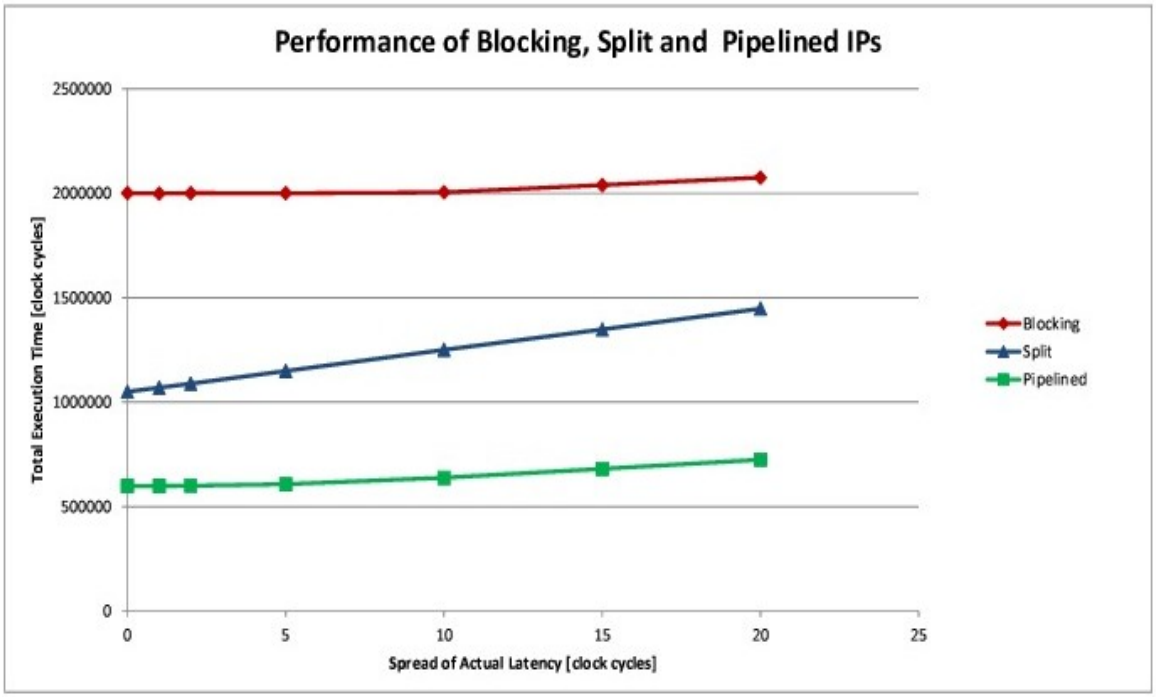


Figure 3.7: Effect of the spread in actual latency on the performance of blocking, split and pipelined IPs

Simplified equation-based Performance estimation

4

In Chapter 2, we introduced two techniques for performance estimation, namely the trace-based performance estimation and the simplified equation-based performance estimation. Both these performance estimation techniques are based on the same high-level performance models explained in Chapter 2.

While the trace-based estimation technique involves executing the entire external memory request trace, the simplified equation-based estimation technique involves approximating the execution of the trace by using simple equations and simple performance metrics. Since the simplified equations approximate the execution of the external memory request trace, the performance results are not always precise. However, the simplified equations allow the SoC designer to make rough performance estimates of the functional modules and the SoC infrastructure, as against using the traditional ball-park figure based approach to pessimistically dimension the functional modules/ SoC infrastructure. The simplified equation-based performance estimation is mainly used for rapid average/ worst-case performance estimation of functional modules and the SoC infrastructure. The simplified equation-based performance estimation is explained in the Section 4.1.

4.1 Performance estimation using simplified equations

In Chapter 3, we studied the trace-based performance estimation technique. In this technique, the functional module is characterized by the entire external memory request trace and the performance is estimated by executing the entire trace on a request by request basis using a trace simulator. The performance estimation technique can be simplified by characterizing the functional modules using simple values and by approximating the execution of the external memory request trace.

In the simplified equation-based performance estimation, we characterize the functional modules by their no-stall interval (NI_{avg}, NI_{spread}). Similarly, the SoC infrastructure is characterized by its actual latency (AL_{avg}, AL_{spread}). The average performance values (NI_{avg}, AL_{avg}) can be derived either by taking the mean of individual values (NI_i, AL_i) or by considering the worst-case performance values (say, NI_{min}, AL_{max}) or by simply making a calculated guess based on past experience. Similarly, the spread in the performance values (NI_{spread}, AL_{spread}) is determined by considering the request issuing behaviour of the functional modules, the request serving behaviour of the SoC infrastructure, hardware capabilities of the IP (e.g. N_{max}), etc.

The proposed performance estimation technique is based on deriving the average perceived latency (PL_{avg}) using a set of simple equations that approximate the execution of the trace. The average perceived latency ultimately enables the SoC designer to determine the estimated performance of the given functional module in association with the SoC infrastructure. The basic equation used for performance estimation is given by

Equation (4.1).

$$PL_{avg} = max(0, AL_{avg} - NI_{avg}) \tag{4.1}$$

Equation (4.1) is the simplest and the most fundamental equation used for performance estimation in this technique. It does not consider the effect of the spread in no-stall interval and actual latency values (NI_{spread}, AL_{spread}).

4.2 Experiments

To verify the equation-based performance estimation technique and gain some deeper insights, we perform experiments similar to those illustrated in Chapter 3. We first study the simplified equation-based technique for the blocking IPs and later deal with the split and pipelined IPs.

4.2.1 Blocking IP

This experiment involves estimating the performance of a blocking IP using the equation-based technique. We use the same audio decoder application trace that was used in the Simit-ARM experiment (refer Chapter 3). Since the experiment involves blocking IPs, the average no-stall interval (NI_{avg}) of the functional module is zero. Hence, Equation (4.1) is reduced to Equation (4.2)

$$PL_{avg} = AL_{avg} \tag{4.2}$$

Table 4.1 presents the performance metrics (in terms of average perceived latency) of the blocking IP derived using the trace-based technique and the equation-based technique, for varying values of average actual latency of the SoC infrastructure.

Table 4.1: Comparison between the results of the trace-based and the equation-based performance estimation techniques for blocking IPs

	Trace-based technique	*Equation-based technique*	
AL_{avg} *(clock cycles)*	PL_{avg} *(clock cycles)*	PL_{avg} *(clock cycles)*	*Error (clock cycles)*
5	4.76	5	0.23
10	9.76	10	0.23
15	14.76	15	0.23
20	19.76	20	0.23
30	30.76	30	0.23
40	39.76	40	0.23
50	49.76	50	0.23

It is clear that the results of the equation-based estimation technique are similar to those produced by the trace-based estimation technique. The main reason for the high accuracy of the results is the simple execution behaviour of blocking IPs.

4.2.2 Split IPs

The performance of a split IP is mainly determined by its average no-stall interval and the average actual latency of the SoC infrastructure (NI_{avg}, AL_{avg}). However, the performance of the split IPs is also influenced by the spread in no-stall interval and actual latency values (NI_{spread}, AL_{spread}). The experiments for the split IP are thus performed with the goal of analyzing the effect of the spread (NI_{spread}, AL_{spread}) on the accuracy of the simplified equation-based technique as compared to the trace-based technique. The experiment involves estimating the performance of a split IP for the following three configurations:

- with $AL_{spread} = 0$ and $NI_{spread} = 0$
- with $AL_{spread} \neq 0$ and $NI_{spread} = 0$
- with $AL_{spread} = 0$ and $NI_{spread} \neq 0$

$AL_{spread} = 0$ and $NI_{spread} = 0$

For this experiment, we utilize a trace with an average no-stall interval (NI_{avg}) of 9.1 clock cycles. The experiment involves estimating the performance of the functional module using the trace-based technique and the simplified equation-based technique (Equation (4.1)) for varying values of average actual latency, from 0 to 50 clock cycles. The experiment results are presented in Table 4.2.

Table 4.2: Comparison between the performance results when there is no spread in the actual latency and the no-stall interval values

	Trace-based performance estimation	*Equation-based performance estimation*	
AL (clock cycles)	PL_{avg} *(clock cycles)*	PL_{avg} *(clock cycles)*	*Error (clock cycles)*
5	0	0	0
10	1.07	0.91	0.16
15	5.99	5.99	0
20	10.99	10.99	0
25	15.99	15.99	0
30	20.99	20.99	0
35	25.99	25.99	0
40	30.99	30.99	0
45	35.99	35.99	0
50	40.99	40.99	0

From Table 4.2, we observe that when the spread in actual latency and the no-stall interval are zero, the results obtained using the simplified equation (Equation (4.1)) are similar to those obtained using the trace-based technique.

$AL_{spread} \neq 0$ and $NI_{spread} = 0$

For this experiment, we utilize a trace with an average no-stall interval (NI_{avg}) of 40 clock cycles. The spread in actual latency is varied from 0 to 30 clock cycles at suitable intervals and a set of simulations is performed for each value of the actual latency spread (AL_{spread}). Each set of simulation involves estimating the performance of the functional module for varying values of average actual latency, from 0 to 70 clock cycles. The experiment results are best presented through an error plot (Figure 4.1) which illustrates the error observed in the results of the simplified equation-based technique compared to that of the trace-based technique for all sets of simulations. The error (ε) is calculated using Equation (4.3). In Equation (4.3), the term $PL_{avg}{}'$ denotes the average perceived latency derived using the trace-based technique while the term PL_{avg} denotes the average perceived latency derived using the equation-based technique (Equation (4.1)).

$$\varepsilon = PL'_{avg} - PL_{avg} \tag{4.3}$$

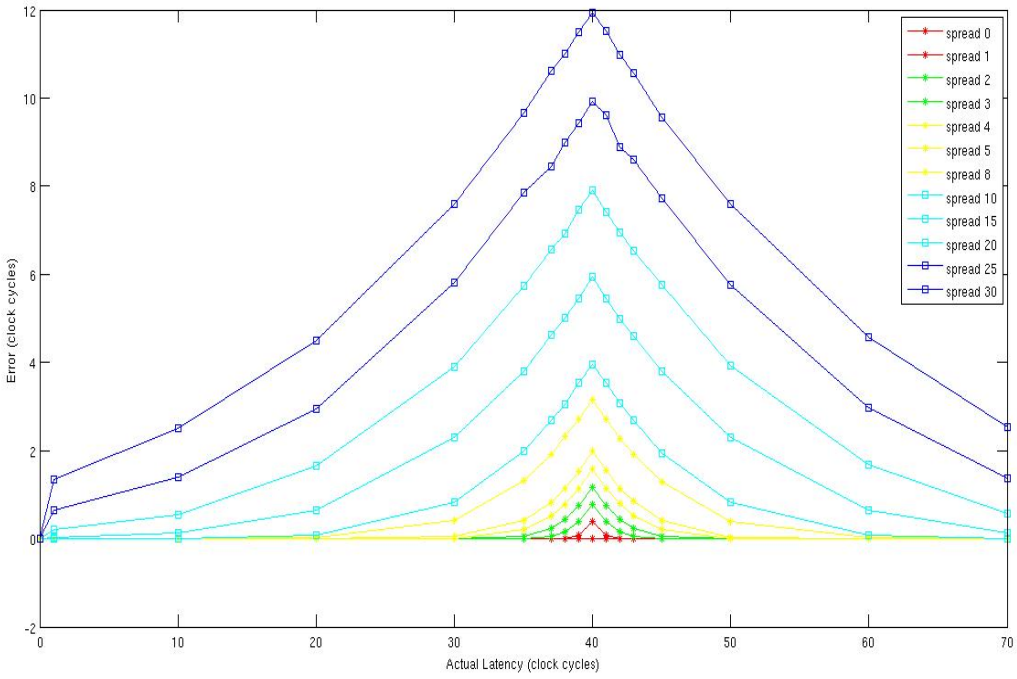


Figure 4.1: Error plot for the simulation and simplified equation-based technique

From Figure 4.1, it is observed that with the introduction of spread in the actual latency values, the results obtained using Equation (4.1) deviate from those obtained via the trace-based estimation technique. In this case, the average perceived latency ($PL_{avg}{}'$) estimated by the trace-based technique is greater than that estimated by the simplified equation-based technique (PL_{avg}). As seen in Chapter 3, the spread in actual latency affects the performance results only when the average actual latency (AL_{avg}) of

the SoC infrastructure is reasonably close to the average no-stall interval (NI_{avg}) of the functional module.

Error analysis

As seen from Figure 4.1, the results of the trace-based performance estimation technique deviate from those of the simplified equation-based technique. This is because Equation (4.1) assumes that all requests are served with an actual latency of AL_{avg}. However, due to the non-zero spread in the actual latency values, some requests are served with an actual latency larger than AL_{avg}, whereas others are served with an actual latency smaller than AL_{avg}. The requests that are served with an actual latency smaller than their no-stall interval do not cause any stalling of the functional module, but the requests that are served with an actual latency greater than their no-stall interval do cause the functional module to stall. Therefore, Equation (4.1) underestimates the total amount of stalling experienced by the functional module.

To mitigate the error introduced in the results of the simplified equation-based technique due to the spread in the actual latency, we propose Equation (4.4) over Equation (4.1). In Equation (4.4), the term AL_{eqv} denotes the *equivalent actual latency* of the SoC infrastructure.

The sole reason for introducing Equation (4.4) *is to ensure that the average perceived latency ($PL_{avg}{}''$) computed using the equivalent actual latency is the same as the average perceived latency ($PL_{avg}{}'$) derived using the trace-based technique (considered as the accurate value of perceived latency).*

$$PL''_{avg} = max(0, AL_{eqv} - NI_{avg}) = PL'_{avg} \tag{4.4}$$

Equivalent actual latency ($\boldsymbol{AL_{eqv}}$)

The concept of equivalent actual latency is introduced to minimize the error in the performance estimation of functional modules caused by the spread in actual latency. In the absence of any spread in the actual latency values, the equivalent actual latency is equal to the average actual latency ($AL_{eqv} = AL_{avg}$). To minimize the effect of the spread in the actual latency values, the equivalent actual latency must be approximated to a value greater than the average actual latency ($AL_{eqv} > AL_{avg}$).

The equivalent actual latency can be derived by multiplying the average actual latency by a suitable correction factor. The correction factor can be mathematically derived from Equation (4.4). The equation for correction factor (C.F.) is given by Equation (4.5).

$$PL''_{avg} = PL'_{avg} = max(0, AL_{eqv} - NI_{avg}) \tag{4.5a}$$

$$\implies PL'_{avg} = max(0, AL_{eqv} - NI_{avg}) \tag{4.5b}$$

$$\implies PL'_{avg} = AL_{eqv} - NI_{avg} \text{ [if } AL_{eqv} > NI_{avg}] \tag{4.5c}$$

$$\implies AL_{eqv} = PL'_{avg} + NI_{avg} \tag{4.5d}$$

$$\implies AL_{avg} \cdot C.F. = PL'_{avg} + NI_{avg} \tag{4.5e}$$

$$\implies C.F. = \frac{PL'_{avg} + NI_{avg}}{AL_{avg}} \tag{4.5f}$$

Figure 4.2 shows a graph of the correction factor (determined using Equation (4.5)) plotted against the average actual latency for various values of actual latency spread. The derivation of the correction factor (and the equivalent actual latency) for a few sample points in the correction factor plot (Figure 4.2) is presented in the following section.

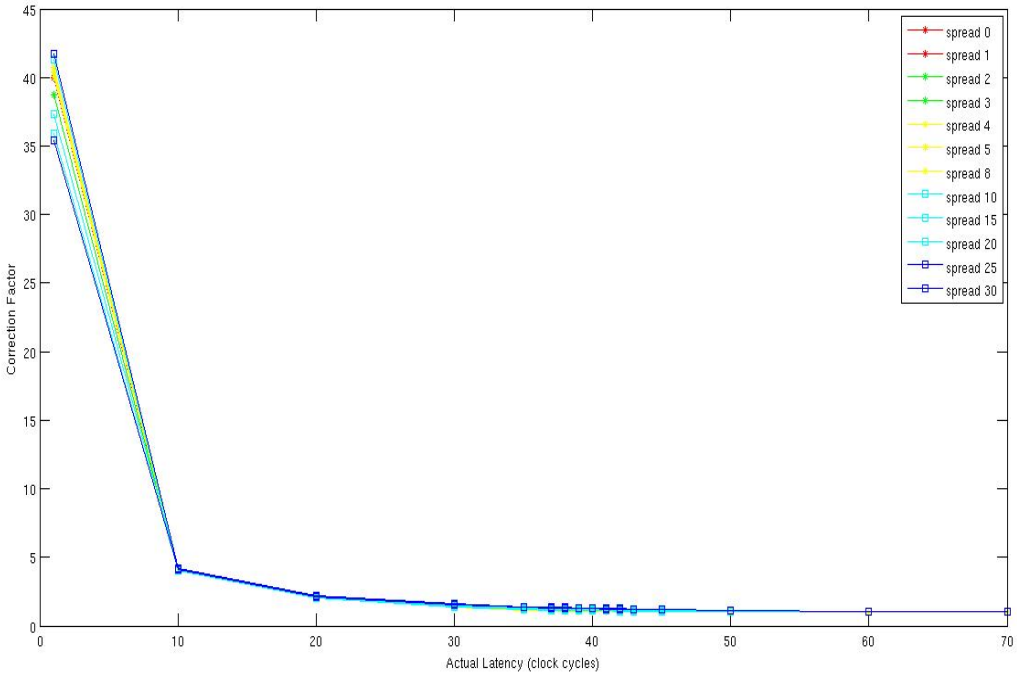


Figure 4.2: Correction factor plot for computing the equivalent actual latency

Deriving the equivalent actual latency for sample data points

From experimental data, the average perceived latency of the functional module (when AL_{spread}= 30 clock cycles) at AL_{avg}=10 clock cycles is 2.5 clock cycles using the trace-based estimation technique and 0 clock cycles using the simplified equation-based

technique. With reference to Equation (4.3), $PL_{avg}{}'$ is 2.5 clock cycles, whereas PL_{avg} is 0 clock cycles. Therefore, the error in the performance estimates of the two techniques is 2.5 clock cycles (observed in Figure 4.2). Now, we need to ensure that the average perceived latency of the trace derived using equation (4.4), i.e $PL_{avg}{}''$, is equal to $PL_{avg}{}'$ (2.5 clock cycles) which is the accurate average perceived latency derived using the trace-based technique. For $PL_{avg}{}''$ to be equal to 2.5 clock cycles, the equivalent actual latency (AL_{eqv}) must be equal to 42.5 clock cycles since the average no-stall interval (NI_{avg}) of the given trace is 40 clock cycles (Equation (4.4)). We compute the equivalent actual latency (AL_{eqv}) by multiplying the average actual latency (AL_{avg}) with an appropriate correction factor (C.F.). Since the average actual latency (AL_{avg}) is equal to 10 clock cycles, the correction factor (C.F.) for the given data point must be 4.25 so that the equivalent actual latency is calculated as 42.5 clock cycles and thus, the average perceived latency is correctly computed as 2.5 clock cycles.

Similarly, the average perceived latency of the functional module (when AL_{spread}= 30 clock cycles) at AL_{avg}=60 clock cycles is 24.42 clock cycles using the trace-based estimation technique and 20 clock cycles using the simplified equation-based technique. In short, $PL_{avg}{}'$ is 24.42 clock cycles and PL_{avg} is 20 clock cycles. This gives an error of 4.42 clock cycles (observed in Figure 4.2) in the performance estimates of the two techniques. We need to ensure that the average perceived latency of the trace derived using equation (4.4), i.e $PL_{avg}{}''$, is equal to $PL_{avg}{}'$ (24.42 clock cycles) which is the accurate average perceived latency derived using the trace-based technique (Equation (4.4)). For $PL_{avg}{}''$ to be equal to 24.42 clock cycles, the equivalent actual latency (AL_{eqv}) must be equal to 64.42 clock cycles since the average no-stall interval (NI_{avg}) of the given trace is 40 clock cycles. Since the average actual latency (AL_{avg}) is equal to 60 clock cycles, the correction factor (C.F.) for the given data point must be $\frac{64.42}{60} = 1.07$.

Approximation of the correction factor

The correction factor plot (Figure 4.2) is essentially derived from the knowledge of the accurate average perceived latency. The next step is to approximate the correction factor (C.F.). From Figure 4.2, we observe that the correction factor (C.F.) can be approximated to a decaying exponential curve. We approximate the correction factor (C.F.) with Equation (4.6).

$$C.F. = max(1, NI_{avg} \cdot e^{-\lambda \cdot AL_{avg}}) \tag{4.6}$$

Here, the term λ denotes the *decay constant* which is indicative of the rate at which the correction factor decays. The λ varies with the functional modules's NI_{avg} and hence, difficult to precisely formulate. It is determined empirically. Using Equation (4.6), the equivalent actual latency (AL_{eqv}) can be mathematically expressed as Equation (4.7).

$$AL_{eqv} = \begin{cases} AL_{avg} & AL_{spread} = 0 \\ AL_{avg} \cdot max(1, NI_{avg} \cdot e^{-\lambda \cdot AL_{avg}}) & AL_{spread} \neq 0 \end{cases} \tag{4.7}$$

$AL_{spread} = 0$ and $NI_{spread} \neq 0$

Similar to the previous experiment, the results obtained via the simplified equation-based technique deviate from those obtained via the trace-based technique if there exists

a non-zero spread in the no-stall interval. In this experiment, we utilize ten different traces each having an average no-stall interval (NI_{avg}) of 40 clock cycles, but with varying amounts of no-stall interval spread (from 0 clock cycles to 20 clock cycles). For each different value of the no-stall interval spread, the performance of the SoC is estimated using the trace-based technique and the equation-based technique, for varying values of average actual latency (from 0 to 40 clock cycles).

The experiment results are best presented using an error plot as shown in Figure 4.3. The plot presents the error observed in the results of the simplified equation-based technique as compared to that of the simulation based technique for varying values of spread in the no-stall interval. Similar to the previous experiment, the error (ε) is calculated using Equation (4.3). Also, the spread in no-stall interval affects the performance of the functional module only when the average actual latency (AL_{avg}) and the average no-stall interval (NI_{avg}) are reasonably close to each other.

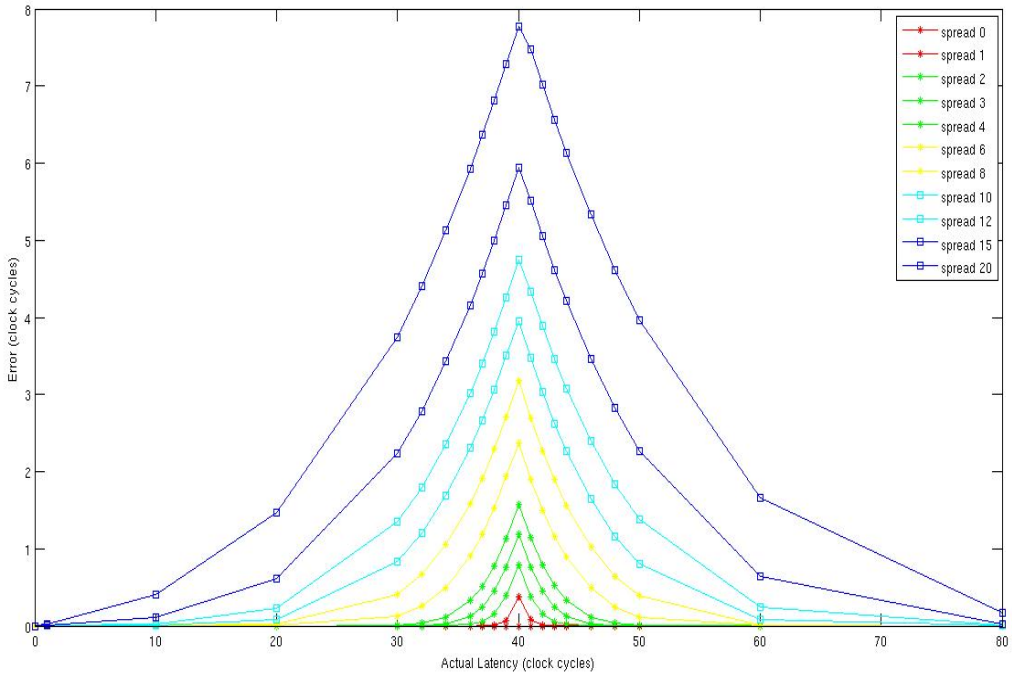


Figure 4.3: Error plot for the simulation and simplified equation-based technique

Error Analysis:

As seen from Figure 4.3, the results of the trace-based performance estimation technique deviate from those of the simplified equation-based technique. Equation (4.1) assumes that all requests have a no-stall interval of NI_{avg}. However, due to the non-zero spread in the no-stall interval values, some requests have a no-stall interval larger than NI_{avg}, whereas others have a no-stall interval smaller than NI_{avg}. For a SoC infrastructure which serves all requests with an average actual latency of AL_{avg}, the requests that

have a no-stall interval larger than the AL_{avg} do not provide any performance gain, but requests with a no-stall interval smaller than AL_{avg} lead to the stalling of the functional module. Since Equation (4.1) does not take this into account, it underestimates the amount of stalling experienced by the functional modules.

We know that an external memory request trace with an average no-stall interval of NI_{avg} and a non-zero spread ($NI_{spread} \neq 0$) has the same performance as that of a trace with average no-stall interval $NI_{avg}{}'$ and a zero no-stall interval spread ($NI_{spread} = 0$), such that $NI_{avg}{}' < NI_{avg}$. To mitigate the error introduced in the results of the simplified equation-based technique due to the spread in the no-stall interval, we propose Equation (4.8) over Equation (4.1). In Equation (4.8), the term NI_{eqv} denotes the *equivalent no-stall interval* of the functional module.

Similar to the previous experiment, the main reason for introducing Equation (4.8) *is to ensure that the average perceived latency ($PL_{avg}{}''$) computed using the equivalent no-stall interval is the same as the average perceived latency ($PL_{avg}{}'$) derived from the trace-based technique (considered as the accurate value of perceived latency).*

$$PL_{avg} = max(0, AL_{avg} - NI_{eqv}) \tag{4.8}$$

Equivalent no-stall interval (NI_{eqv})

The concept of equivalent no-stall interval is introduced to minimize the error introduced in the performance estimation of functional modules due to the spread in no-stall interval. In the absence of any spread in the no-stall interval values, the equivalent no-stall interval is equal to the average no-stall interval ($NI_{eqv} = NI_{avg}$). To minimize the effect of the spread in the no-stall interval values, the equivalent no-stall interval must be approximated to a value smaller than the average no-stall interval ($NI_{eqv} < NI_{avg}$). *The equivalent no-stall interval can be derived by multiplying the average no-stall interval by a suitable correction factor.*

The correction factor can be mathematically derived from Equation (4.8). The equation for correction factor is given by Equation (4.9).

$$PL''_{avg} = PL'_{avg} = max(0, AL_{avg} - NI_{eqv}) \tag{4.9a}$$

$$\implies PL'_{avg} = max(0, AL_{avg} - NI_{eqv}) \tag{4.9b}$$

$$\implies PL'_{avg} = AL_{avg} - NI_{eqv} \text{ [if } AL_{avg} > NI_{eqv}] \tag{4.9c}$$

$$\implies NI_{eqv} = AL_{avg} - PL'_{avg} \tag{4.9d}$$

$$\implies NI_{avg} \cdot C.F. = AL_{avg} - PL'_{avg} \tag{4.9e}$$

$$\implies C.F. = \frac{AL_{avg} - PL'_{avg}}{NI_{avg}} \tag{4.9f}$$

Figure 4.4 shows a graph of the correction factor (determined using Equation (4.9)) plotted against the average actual latency for various values of the no-stall interval spread. The derivation of the correction factor (and the equivalent no-stall interval) for a few sample points in the correction factor plot (Figure 4.2) is presented in the following section.

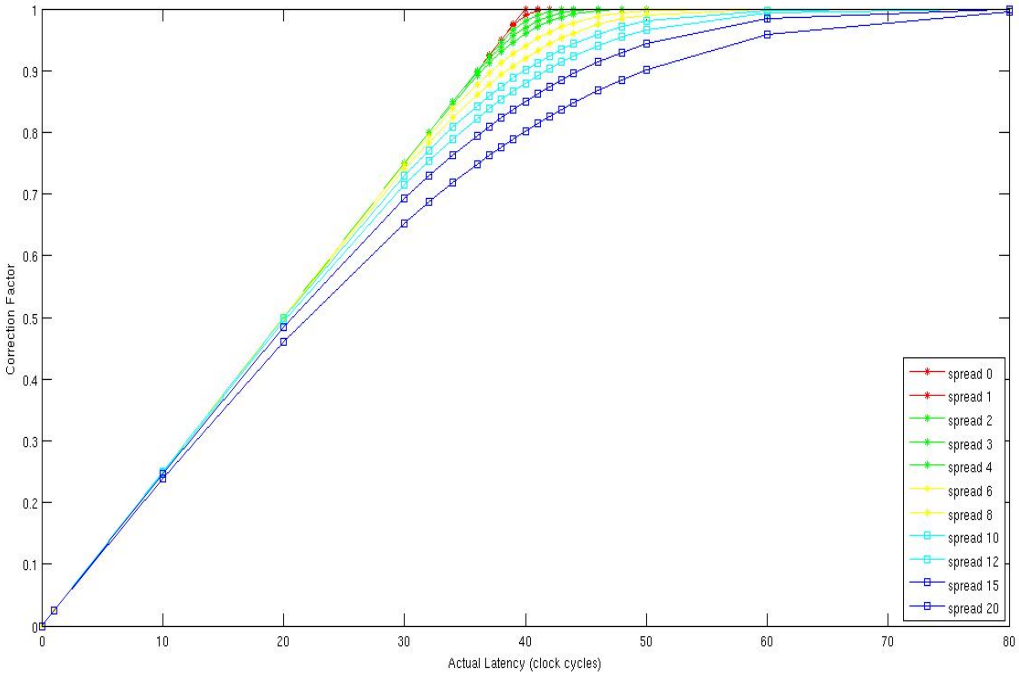


Figure 4.4: Correction factor plot for computing the equivalent no-stall interval

Deriving the equivalent no-stall interval for sample data points

From experimental data, the average perceived latency of the functional module (when NI_{spread}= 20 clock cycles) at AL_{avg}=10 clock cycles is 0.41 clock cycles using the trace-based estimation technique and 0 clock cycles using the simplified equation-based technique. With reference to Equation (4.3), $PL_{avg}{}'$ is 0.4 clock cycles and PL_{avg} is 0 clock cycles. Thus, the error in the performance results of the two technique is 0.4 clock cycles (observed in Figure 4.4). We need to ensure that the average perceived latency of the trace derived using equation (4.8), i.e $PL_{avg}{}''$, is equal to $PL_{avg}{}'$ (0.4 clock cycles) which is the accurate average perceived latency derived using the trace-based technique.

For PL_{avg}'' to be equal to 0.4 clock cycles, the equivalent no-stall interval (NI_{eqv}) must be equal to 9.6 clock cycles since the average actual latency (AL_{avg}) at the given data point is 10 clock cycles. We compute the equivalent no-stall interval (NI_{eqv}) by multiplying the average no-stall interval (NI_{avg}) with an appropriate correction factor (C.F.). Since the average no-stall interval (NI_{avg}) is equal to 40 clock cycles, the correction factor (C.F.) for the given data point must be $\frac{9.6}{40} = 0.24$ so that the equivalent no-stall interval is calculated as 9.6 clock cycles and thus, the average perceived latency is correctly computed as 0.4 clock cycles.

Similarly, the average perceived latency of the functional module (when NI_{spread}= 20 clock cycles) at AL_{avg}=60 clock cycles is 21.47 clock cycles using the trace-based estimation technique and 20 clock cycles using the simplified equation-based technique. In short, PL_{avg}' is 21.47 clock cycles and PL_{avg} is 20 clock cycles. This gives an error of 1.47 clock cycles (observed in Figure 4.4). We need to ensure that the average perceived latency of the trace derived using equation (4.8), i.e PL_{avg}'' , is equal to PL_{avg}' (21.47 clock cycles) which is the accurate average perceived latency derived using the trace-based technique. For PL_{avg}'' to be equal to 21.47 clock cycles, the equivalent no-stall interval (AL_{eqv}) must be equal to 38.53 clock cycles since the average actual latency (AL_{avg}) at the given data point is 60 clock cycles. Since the average no-stall interval (NI_{avg}) is equal to 40 clock cycles, the correction factor (C.F) for the given data point must be $\frac{38.53}{40} = 0.96$.

Approximation of the correction factor

The correction factor plot (Figure 4.4) is derived by using the accurate average perceived latency values. The next step is therefore, to approximate the correction factor (C.F.). From Figure 4.4, we observe that the correction factor (C.F.) can be approximated to a linear curve given by Equation (4.10).

$$C.F. = min(1, \frac{AL_{avg}}{(NI_{avg} + NI_{spread})}) \tag{4.10}$$

Using Equation (4.10), the equivalent no-stall interval (NI_{eqv}) can be mathematically expressed as Equation (4.11).

$$NI_{eqv} = \begin{cases} NI_{avg} & NI_{spread} = 0 \\ NI_{avg} \cdot min(1, \frac{AL_{avg}}{(NI_{avg}+NI_{spread})}) & NI_{spread} \neq 0 \end{cases} \tag{4.11}$$

Conclusion

From the above experiments, we conclude that the simplified equation-based performance estimation technique can be effectively used for split IPs, either when there is no spread in the no-stall interval or the actual latency values or when the average no-stall interval (NI_{avg}) is either much larger or much smaller than the average actual latency (AL_{avg}). In scenarios where the average no-stall interval and the average actual latency are comparable to each other and there exists a non-zero spread in either of the two parameters, the SoC designer should be aware of the possible errors resulting from the spread in actual latency or no-stall interval. The error in the results of the simplified equation-based technique can be minimized by using the following equations.

$$PL_{avg} = max(0, AL_{avg} - NI_{avg}) \quad [AL_{spread} = 0 \text{ and } NI_{spread} = 0]$$
$$PL_{avg} = max(0, AL_{eqv} - NI_{avg}) \quad [AL_{spread} \neq 0 \text{ and } NI_{spread} = 0]$$
$$PL_{avg} = max(0, AL_{avg} - NI_{eqv}) \quad [AL_{spread} = 0 \text{ and } NI_{spread} \neq 0]$$
$$PL_{avg} = max(0, AL_{eqv} - NI_{eqv}) \quad [AL_{spread} \neq 0 \text{ and } NI_{spread} \neq 0]$$

4.2.3 Pipelined IPs

As compared to the blocking and split IPs, the execution behaviour of the pipelined IPs is much more complex and hence, more challenging to evaluate using the simplified equation-based technique. The key issue in employing the simplified equation-based approach for estimating the performance of pipelined IPs is that Equation (4.1) does not account for the derived benefit experienced by the individual requests of a pipelined IP. This is because derived benefit, as the name suggests, is a derived property. To determine the average derived benefit (DB_{avg}) per request, the SoC designer needs to simulate the execution of the entire trace on a per request basis.

This experiment ignores the average derived benefit (DB_{avg}) and evaluates the accuracy of the equation-based technique against that of the simulation-based technique. The experiment involves simulating a synthetic trace on a pipelined IP with $N_{max} = 4$ requests. The average no-stall interval (NI_{avg}) of the external memory request trace is 24 clock cycles with a no-stall interval spread (NI_{spread}) of 12 clock cycles. The performance of the functional module and the SoC infrastructure is estimated using both the trace-based technique and the simplified equation-based technique (Equation (4.8)). The experiment involves simulating the above mentioned trace for varying values of average actual latency (AL_{avg}), from 0 to 50 clock cycles, with zero actual latency spread (AL_{spread}). The results of the experiment are presented in Table 4.3.

Table 4.3: Comparison between the results of the trace-based technique and the equation-based estimation technique for pipelined IPs

	Trace-based technique	*Equation-based technique*	
AL (clock cycles)	PL_{avg} *(clock cycles)*	PL_{avg} *(clock cycles)*	*Error (clock cycles)*
5	0.22	1.66	-1.44
10	1.01	3.33	-2.32
15	2.21	5.00	-2.79
20	3.77	6.66	-2.89
25	5.53	6.94	-1.61
30	7.41	10.00	-2.59
35	9.39	11.66	-2.27
40	11.47	16.39	-4.92
45	13.64	21.39	-7.75
50	15.83	26.39	-10.56

The experimental results illustrate that the equation-based performance estimation technique over-estimates the average perceived latency of the pipelined IP. The over-estimation of the perceived latency is attributed to the fact that the equation-based technique cannot estimate the average derived benefit received by individual requests, thereby arriving at a much higher perceived latency than that observed in reality. The error introduced in the estimation of the perceived latency is as high as 100%, due to which the equation-based technique is not well suited for pipelined IPs.

Conclusion

To summarize, the simplified equation-based performance estimation technique over-estimates the stalling experienced by the pipelined IPs. This is mainly attributed to the inability of the estimation technique to capture the derived benefit of memory requests. We therefore recommend the SoC designers to use the trace-based estimation technique for pipelined IPs.

4.3 Application of the performance estimation techniques

In this section, we study the application of the proposed performance estimation techniques to aid SoC designers in the customization, verification and integration of the four example functional modules presented in Chapter 1. Out of the four examples presented in Chapter 1, two examples (basic CPU and advanced pre-fetching CPU) belong to the class of latency non-critical functional modules, whereas the other two examples (DSP and video engine) belong to the latency critical class of functional modules. Although both the trace-based and the simplified equation-based techniques can be used to estimate the performance of all four functional modules, we study the application of trace-based technique for integrating the DSP and the video engine. Similarly, we apply the simplified equation-based technique for integrating the basic CPU and the advanced pre-fetching CPU.

4.3.1 Performance estimation using trace simulator

The trace-based technique is best utilized for the performance estimation of functional modules employing pipelined IPs such as the DSP and the video engine.

DSP example (Chapter 2)

In this example, the functional module (DSP executing an audio application) is both latency sensitive and latency critical. Typically the DSP issues multiple external memory requests (via the pre-fetch unit) for fetching a given audio frame. The requirement is that the given audio sample should be completely available before a fixed deadline so that the DSP can process the data fast enough to prevent any loss in audio quality. The execution of application code on the DSP generates a memory request trace as shown in Figure 2.6, where memory requests typically have an average no-stall interval (NI_{avg}) of 15-20 clock cycles. However, the external memory request trace generated by the pre-fetch unit of the DSP resembles the trace shown in Figure 2.7, where all requests typically get an average no-stall interval (NI_{avg}) of 60-80 clock cycles with a little spread

(15-20 clock cycles) in the no-stall interval values. This is because the pre-fetch unit of the DSP can typically issue multiple outstanding requests (N_{max} =4 requests). While it is not critical that each external memory request issued by the IP is served within its individual deadline, it is essential that a collective set of requests (say, for one audio sample) are all served before a fixed deadline. The SoC designer expects the perceived latency for each request issued by the pre-fetch unit to be less than a threshold value, which is determined by taking into account the base time of the trace and the deadlines set for the collective set of requests. The average actual latency (AL_{avg}) of the SoC infrastructure is typically around 100 clock cycles. The performance estimation results of the trace-based technique for a DSP executing a synthetic audio decoder application trace with the above mentioned specifications (NI_{avg} = 60 clock cycles, NI_{spread}= 25 clock cycles) is presented in Table 4.4.

Table 4.4: Performance estimation results of a DSP ($N_{max} = 4$) executing a typical audio decoder application trace.

Actual Latency *(clock cycles)*	*Total Execution Time* *(clock cycles)*	*Average perceived latency* *(clock cycles)*
20	1330596	1.61
40	1557508	6.15
60	1885195	12.70
80	2258434	20.19
100	2649149	28.27
120	3051390	36.96
140	3454292	45.70

Suppose the threshold for the average perceived latency (PL_{avg}) is set to 30 clock cycles, we can conclude (from Table 4.4) that a SoC infrastructure which serves memory requests with an average actual latency (AL_{avg}) less than 100 clock cycles is suitable for the given functional module.

Video engine example (Chapter 2)

In this example, the video IP is latency tolerant, but is latency critical. The requests issued by the pre-fetch unit of the video IP typically have an average no-stall interval (NI_{avg}) of a 200-400 clock cycles. The average actual latency (AL_{avg}) of the SoC infrastructure is typically around 100-150 clock cycles with a considerable amount of spread (AL_{spread} = 50 clock cycles) in the actual latency values. By performing simulation of the worst case external memory request trace for a range of SoC infrastructure configurations, we can perform rapid estimation of the average number of stall cycles experienced by the video IP for every external memory request. Similar to a DSP executing an audio application, the video IP requires a set of requests to be collectively served before a fixed deadline. In this example, we assume that the video IP is designed such that average perceived latency of all requests should be at most a couple of clock cycles. Thus, the SoC designer is mainly interested in ensuring that the average perceived latency of the external memory request is below two-three clock cycles.

The average actual latency (AL_{avg}) is varied from 20 clock cycles to 150 clock cycles, with a spread (AL_{spread}) of 50 clock cycles. The results of the simulator executing a synthetic trace (NI_{avg} = 200 clock cycles, NI_{spread} = 25 clock cycles) with the above mentioned specifications on a pipelined video IP with varying SoC infrastructure configurations is given in Table 4.5.

Table 4.5: Performance estimation results of a video IP ($N_{max} = 4$) executing a typical video application trace.

Actual Latency *(clock cycles)*	*Total Execution Time* *(clock cycles)*	*Average perceived latency (clock cycles)*
20	3256809	0
40	3256809	0
60	3256809	0
80	3256809	0
100	3272655	0.46
120	3305108	1.10
150	3427107	3.55

From Table 4.5, we conclude that the video IP does not experience any stalling and hence, does not miss any deadline until the average actual latency (AL_{avg}) is less than 100 clock cycles. The SoC infrastructure satisfies the performance requirements of the video IP until its average actual latency is less than 120 clock cycles, in spite of the large spread in actual latency values.

4.3.2 Performance estimation using simplified equation

The simplified equation-based performance estimation technique can be utilized for performing rapid worst case performance estimation of the functional modules and the SoC infrastructure, especially for the blocking and split types of IPs. The technique is useful in real-life SoC design, especially in dealing with split type processors executing latency non-critical applications.

Simple CPU example (Chapter 2)

In the simple CPU example of Chapter 2, the functional module (simple processor) is latency sensitive, but latency non-critical. The processors are typically split in nature, having an average no-stall interval (NI_{avg}) of a couple of clock cycles with very little or no spread in the no-stall interval values. The average actual latency (AL_{avg}) of the SoC infrastructure is typically around 100-120 clock cycles. Thus, using the simplified equations, we can make a rough estimate that every external memory request would stall the processor for an average of 100-120 clock cycles. Depending on the criticality of the application being executed, we can decide whether the performance offered by the processor and/or the SoC infrastructure is acceptable or not.

Pre-fetching CPU example (Chapter 2)

In the advanced pre-fetching CPU example of Chapter 2, the functional module (processor with pre-fetching cache) is latency tolerant and latency non-critical. The processors typically having an average no-stall interval (NI_{avg}) of 20-30 clock cycles with considerable amount of spread in the no-stall interval values. The spread in no-stall interval is mainly because the pre-fetching cache does not equally benefit all memory accesses. Thus, while most requests benefit from the pre-fetching capability of the CPU, some requests do not benefit from it and therefore, have a small no-stall interval. These requests introduce a considerable amount of spread in the no-stall interval values. The average actual latency (AL_{avg}) of the SoC infrastructure is typically around 100-120 clock cycles. Using Equation (4.1), we can estimate that a every external memory request would stall the processor for an average of 90-100 clock cycles. Depending on the criticality of application being executed, the SoC designer can easily estimate whether the performance offered by the processor and the SoC infrastructure is acceptable or not.

4.4 Conclusion

In this chapter, we studied the simplified equation-based performance estimation technique in detail. We analyzed the effect of the spread in no-stall interval and actual latency on the accuracy of the equation-based technique and proposed solutions to minimize the error induced by the spreads. We also studied the application of the performance estimation techniques for commonly found functional modules in today's SoC designs.

5 Related work

In this thesis, we proposed two performance estimation techniques that address the challenges encountered in the integration of IPs into complex SoC designs. The performance estimation techniques allow rapid performance estimation of IPs and the SoC infrastructure, thereby allowing the SoC designers to appropriately configure and integrate them. These estimation techniques are based on high-level performance models that capture the latency tolerance of IPs and their dependence on the service provided by the SoC infrastructure. Often in real-life SoC designs, the main concern of the SoC designers is to ensure that the performance of the processors and other IPs match their expectations after they are integrated into custom SoCs. In such scenarios, it is crucial to study and understand the latency tolerance aspect of the processors.

Latency tolerance and latency criticality of IPs with respect to the service latency of the SoC infrastructure (memory subsystem, on-chip network, etc.) are widely known concepts and have received attention in [4, 11, 13, 15]. [15] introduces a metric termed as the *likelihood of criticality*, which denotes the criticality of the load/store instructions and communication latencies with respect to the overall performance of a clustered system. It further uses this metric to implement a criticality-based scheduler which prioritizes instructions based on their criticality, thereby improving the overall performance of the system. Similar to [15], [4] deals with the *stall-time criticality* of applications and proposes several prioritization policies to improve the throughput (and thus, performance) of on-chip networks while still preserving fairness in the network. Although the paper introduces several techniques to determine the stall-time criticality of applications, it fails to provide a strong mathematical framework for quantifying the latency criticality of applications. These papers utilize the latency tolerance and latency criticality of IPs to either improve the performance of the IPs [8, 15] or to perform better arbitration of the shared resource (e.g. the on-chip network) [4]. The concepts of latency tolerance and latency criticality have not been explored for estimating the performance of IPs, which is central to our work.

Researchers have proposed numerous performance estimation techniques for IPs and SoC infrastructure such as those found in [5, 7, 12, 18, 20, 22]. [5] propose a performance estimation technique for bus based communication architectures using Stochastic Automata Network (SAN). The proposed performance estimation technique aims at providing early estimates of performance metrics (such bandwidth, queue length and the waiting time for processing elements (IPs)) that help SoC designers to select the appropriate communication architecture for their SoCs. The paper specifically deals with utilizing the SAN model for single shared bus and hierarchical bus bridge architectures. The model assumes that all processing elements connected to the bus are served according to a priority based arbitration policy. The proposed analytical technique is derived from this assumption, which restricts the use of this estimation technique in SoCs using

on-chip networks and buses that employ other arbitration schemes.

Performance estimation of SoCs using SANs is also proposed in [12]. This paper presents an analytical technique for system-level power and performance analysis that helps SoC designers to select the right IPs for a given set of target applications. The paper uses the Stochastic Automata Networks (SANs) as a formalism for the average-case analysis of a set of application-architecture combinations so as to aid the SoC designer in identifying the best combination early in the design cycle. The paper fails to relate the performance of the memory subsystem to the performance of the application-architecture combination. Thus, this technique is suited only for determining the optimal application-architecture combination for a given SoC infrastructure, irrespective of whether the SoC infrastructure can support its performance requirements.

Similarly, Platune [7] is another performance estimation framework that aids the SoC designer in choosing the appropriate architectural parameters for a given application mapped onto a parameterized SoC platform with the goal of satisfying its performance and power requirements. It is mainly focused on simplifying the design space exploration problem encountered in selecting the right set of parameters in a parameterized SoC platform to deliver optimal power and performance. The processors, caches, memory subsystem and other on-chip peripherals are modeled and analyzed independently without capturing the performance dependence among them. Furthermore the performance estimation for each of the configurations is not accurate in the absolute sense, but is aimed to be relative to that of the other configurations, thereby making it unsuitable for SoCs providing (hard/soft) real time functionality.

[22] introduces a novel SoC performance evaluation methodology that is based on modeling memory accesses. It first finds the performance bottleneck in a given SoC design and then proposes an appropriate SoC architecture in the early SoC design stages, thereby saving a lot of time and effort involved in the design iteration cycles. It exploits the AHB bus characteristics to estimate the performance of the SoC. However, since the modeling of the memory access behaviour of the AHB bus is central to this performance estimation technique, it cannot be used for SoCs which employ other types of interconnects. Although the paper relates the performance of the SoC to that of the memory subsystem and the interconnect, the possibility of IPs issuing multiple memory requests is ignored. It, therefore, does not discuss the concept of memory level parallelism or the latency tolerance/ criticality of IPs which have become important issues in SoC design today.

A high-level model characterizing the latency tolerance of processors, called the *processor queuing model* is proposed in [20]. This paper is the closest to our work. The paper aims at modeling multiprocessor systems running memory intensive, commercial online transaction processing (OLTP) workloads in order to improve the accuracy of their performance estimates, thereby aiding the design of such systems. Similar to our approach, the paper proposes modeling of the multiprocessor system by analyzing the external memory request trace. It also deals with re-iterating the simulations for evaluating the performance of the processors for various memory subsystem configurations. Although the paper deals with the performance dependence of processors on the memory subsystem, it fails to consider the application code dependencies that affect the performance of the processors. The load-load dependencies (similar to the request dependencies) is

claimed to have negligible impact on the processor's performance for the OLTP workloads they execute. This restricts the use of their model for a specific type of workload (i.e OLTP workloads). Furthermore, the paper does not deal with the code dependencies (i.e. a non-memory movement instruction dependent on a memory request) since they are assumed to be non-existent in the applications they consider. Although the paper deals with the stalling of processors and the concept of overlapping memory requests, they do not provide a mathematical framework for quantifying the latency tolerance of processors. We believe that our model is better than the one proposed in [20] mainly because we present an elaborate mathematical framework that allows us to quantify the latency tolerance of IPs and co-relate it with the performance of the SoC infrastructure, thereby enabling us to accurately estimate their performance.

6 Conclusion and Future work

This chapter is presented as follows. Firstly, we summarize the thesis and present the broad conclusions derived from this work. Later, we propose recommendations for using the performance estimation techniques in the context of real-life SoC designs. Finally, we propose possible future work that can be undertaken as an extension of this thesis.

6.1 Summary

In this thesis, we have proposed two novel performance estimation techniques, namely the trace-based performance estimation technique and the simplified equation-based performance estimation technique. The performance estimation techniques are based on high-level performance models of IPs and the SoC infrastructure that capture some of their key performance characteristics (e.g. latency tolerance of IPs). The high-level performance models accurately capture the execution behaviour of three types of IPs - blocking, split and pipelined. The high-level performance models are useful in characterizing a functional module once and re-using it in subsequent SoC designs. This enhances the re-usability of functional modules.

The trace-based performance estimation utilizes the entire (worst-case) application traces for characterizing the functional modules. The external memory request traces are analyzed to determine the no-stall interval of individual memory requests. The effort and time invested in extracting and analyzing the application trace, as opposed to executing application code on an instruction set simulators is justified since trace extraction and analysis is fairly simple and quick.

The simplified equation-based performance estimation technique characterizes the functional modules and the SoC infrastructure using average performance values, which can either be derived from a worst-case application trace or simply determined by past experience. The simplified equation-based technique is useful for providing rapid performance estimates of functional modules and the SoC infrastructure in the early design stages, although with a possible reduction in the accuracy for the split and pipelined functional modules.

Both these performance estimation techniques are particularly useful in rapidly re-assessing the performance of all functional modules, every time a new functional module is added to a given SoC design. The high-level performance models allow the SoC designer to iteratively estimate the performance of a given SoC over a range of IP and SoC infrastructure configuration. This allows the SoC designer to perform rapid design space exploration. The performance estimates provided by these techniques in the early SoC design stages saves a significant portion of the design time. The performance estimation techniques therefore simplify the process of integrating new IPs/ IP subsystems into existing SoC designs.

6.2 Conclusion

From the experimental results, we present the following conclusions with respect to the performance of functional modules and their dependence on the service provided by the SoC infrastructure. To begin with, the total execution time of an application/ trace reduces with the increasing hardware capability of the IPs, from blocking to pipelined IPs. For the pipelined IPs, the total execution time of the external memory request trace (and hence, the application) decreases with the increasing ability of the pipelined IPs to issue multiple outstanding requests.

As far as the stalling of functional modules is concerned, the average perceived latency experienced by the blocking IPs increases linearly with the average actual latency of the SoC infrastructure. The split and pipelined IPs experience negligible stalling as long as the average actual latency of the SoC infrastructure is less than their average no-stall interval. When the average actual latency of the SoC infrastructure exceeds the average no-stall interval of the split and pipelined IPs, the average perceived latency experienced by these IPs increases linearly with the average actual latency. These conclusions are appropriate as long as the spread in no-stall interval and actual latency is negligible.

With the introduction of spread in no-stall interval and actual latency, the performance of split/pipelined IPs degrades whenever the average actual latency of the SoC infrastructure is reasonably close to the average no-stall interval of the functional module. This is because, an external memory request trace with an average no-stall interval of NI_{avg} and a non-zero spread ($NI_{spread} \neq 0$) produces the same performance as that of a trace with an average no-stall interval $NI_{avg}{}'$ and a zero no-stall interval spread ($NI_{spread} = 0$), such that $NI_{avg}{}' < NI_{avg}$. The spread in no-stall interval and actual latency values does not have any bearing on the performance of the functional module when the average service latency of the SoC infrastructure is either significantly smaller or significantly larger than the average no-stall interval of the functional module. Also, pipelined IPs are more tolerant to the spread in the actual latency of the SoC infrastructure than the split IPs.

6.3 Recommendations

From the various experiments and our understanding of the performance estimation techniques, we provide the following recommendations on using them in real-life SoC designs. Both the performance estimation techniques can be used for estimating the performance of blocking, split and pipelined IPs. However, we recommend the following:

1. While estimating the performance of pipelined IPs, we recommend the use of trace-based performance estimation technique over the simplified equation-based technique since it offers higher accuracy and thus, a more realistic performance estimation.
2. For the same reason, the trace-based performance estimation should also be preferred whenever there is a large spread in the values of no-stall interval or the actual latency. An exception to this recommendation would be the case when the average no-stall interval of the functional module is either very large or very small

as compared to the average actual latency of the SoC infrastructure. In such cases, both the estimation techniques are equally effective.

6.4 Future work

The future work can be classified in the following directions:

1. The simplified model can be further extended to improve the accuracy of performance estimation for pipelined IPs, thereby enabling SoC designers to utilize the simplified equation-based technique for all types of IPs.
2. The proposed performance estimation technique, especially the simplified equation-based estimation technique, can be implemented in hardware to perform real-time analysis of the performance of functional modules and SoC infrastructure. This real-time analysis can be performed at the entrance of the on-chip network (say a port connecting multiple latency critical functional modules to the on-chip network). The average no-stall intervals of the functional modules can be programmed into the port. The average actual latency provided by the SoC infrastructure to every individual functional module can be tracked and updated in real time. With the knowledge of the average no-stall interval of the functional modules and the average actual latency with which each of the functional modules is being served, the port could assign priority stamps to the requests issued by the functional modules based on their urgency. These priority stamps can be used within the on-chip network (to prioritize between the different packets) or at the memory controller (to prioritize memory requests).
3. In this thesis, we characterize the functional modules and the SoC infrastructure independently, with no inter-dependence between them. However, in reality, the SoC infrastructure characterization should be closely linked to the execution behaviour of all functional modules it serves. This is because, parameters like the actual latency (AL) significantly depend on the request issuing behaviour of the functional modules. Future work may therefore involve analyzing the dependence between the functional modules and SoC infrastructure and ultimately, capturing this dependence in their performance models.

Bibliography

[1] *The simit-arm simulator*, `http://simit-arm.sourceforge.net/`, 2007.

[2] D. Araki, A. Nakamura, and M. Miyama, *Model-based soc design using esl environment*, SoC Design Conference (ISOCC), 2010 International, nov. 2010, pp. 83 –86.

[3] P.J. Bricaud, *Ip reuse creation for system-on-a-chip design*, Custom Integrated Circuits, 1999. Proceedings of the IEEE 1999, 1999, pp. 395 –401.

[4] R. Das, O. Mutlu, T. Moscibroda, and C.R. Das, *Application-aware prioritization mechanisms for on-chip networks*, Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, dec. 2009, pp. 280 –291.

[5] U. Deshmukh and V. Sahula, *Stochastic automata network for performance evaluation of heterogeneous soc communication*, NORCHIP, 2008., nov. 2008, pp. 208 –211.

[6] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara, *Specify-explore-refine (ser): From specification to implementation*, Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE, june 2008, pp. 586 –591.

[7] T. Givargis and F. Vahid, *Platune: a tuning framework for system-on-a-chip platforms*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **21** (2002), no. 11, 1317 – 1327.

[8] P. Grun, N. Dutt, and A. Nicolau, *Mist: an algorithm for memory miss traffic management*, Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on, 2000, pp. 431 –437.

[9] A.B. Kahng, *Design technology productivity in the dsm era*, Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific, 2001, pp. 443 –448.

[10] Jie Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, *Software timing analysis using hw/sw cosimulation and instruction set simulator*, Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on, mar 1998, pp. 65 –69.

[11] O. Mutlu and T. Moscibroda, *Stall-time fair memory access scheduling for chip multiprocessors*, Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on, dec. 2007, pp. 146 –160.

[12] A. Nandi and R. Marculescu, *System-level power/performance analysis for embedded systems design*, Design Automation Conference, 2001. Proceedings, 2001, pp. 599 – 604.

[13] M.K. Qureshi, D.N. Lynch, O. Mutlu, and Y.N. Patt, *A case for mlp-aware cache replacement*, Computer Architecture, 2006. ISCA '06. 33rd International Symposium on, 0-0 2006, pp. 167 –178.

[14] J.A. Rowson and A. Sangiovanni-Vincentelli, *Interface-based design*, Design Automation Conference, 1997. Proceedings of the 34th, jun 1997, pp. 178 –183.

[15] P. Salverda and C. Zilles, *A criticality analysis of clustering in superscalar processors*, Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on, nov. 2005, pp. 12 pp. –66.

[16] S. Sarkar, S. Chanclar G, and S. Shinde, *Effective ip reuse for high quality soc design*, SOC Conference, 2005. Proceedings. IEEE International, sept. 2005, pp. 217 – 224.

[17] P. Schindler, K. Weidenbacher, and T. Zimmermann, *Ip repository, a web based ip reuse infrastructure*, Custom Integrated Circuits, 1999. Proceedings of the IEEE 1999, 1999, pp. 415 –418.

[18] S. Sudharsanan, *Performance evaluation of a dvd processor using transaction level models*, Consumer Electronics, 2004 IEEE International Symposium on, 1-3, 2004, pp. 375 – 380.

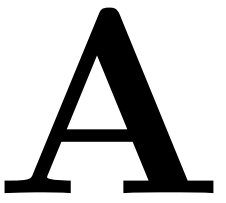
[19] G. Surendra, S.K. Nandy, and P. Sathya, *Redeem rtl: a software tool for customizing soft cells for embedded applications*, VLSI Design, 2001. Fourteenth International Conference on, 2001, pp. 85 –90.

[20] Thin-Fong Tsuei and W. Yamamoto, *Queuing simulation model for multiprocessor systems*, Computer **36** (2003), no. 2, 58 – 64.

[21] Xavier Warzee and P. Kajfasz, *Semantics based co-specifications to design dsp systems*, VHDL International Users' Forum, 1997. Proceedings, oct 1997, pp. 105 –108.

[22] Li Zhou, Tao Sun, Sufen Wei, and Qi Guo, *A multimedia soc performance evaluation method based on memory access model*, Image and Signal Processing (CISP), 2010 3rd International Congress on, vol. 1, oct. 2010, pp. 463 –467.

Algorithm for computing the derived benefit of external memory requests



The Matlab code for the algorithm to compute the derived benefit of individual requests is given below. At the start of the algorithm, the derived benefit of all requests is initialized to zero clock cycles. The algorithm progresses through the entire external memory request trace, analyzing at each request R_i, whether any of the outstanding requests stalls the functional module before request R_i is served. In case a request leads to the stalling of the functional module, the derived benefit for all outstanding requests at that instant are updated. There are two cases when requests get the derived benefit (due to the stalling caused by another outstanding request).

1. If one of the previous outstanding requests R_j stalls the functional module, all outstanding requests (including request R_i, but excluding request R_j) get the derived benefit.
2. If request R_i itself stalls the functional module, all previous outstanding requests (excluding request R_i) get the derived benefit.

As explained in Chapter 2, a given request gets the derived benefit from both the forward and backward direction. The algorithm ensures that the derived benefit is propagated in forward direction through Case 1. The derived benefit is propagated in the backward direction through Case 2.

```
1  %Initialize derived benefit of all requests to zero
2   derived_benefit = zeros(trace_size, 1)';
3
4  %Initialize current_time to zero
5  current_time = 0;
6
7  % Analyze the entire memory request trace
8  for i = 1 : trace_size
9   issue_time(i) = current_time;
10
11   %If no request is issued in the no-stall interval of Request i
12   if (no-stall_interval(i)  < inter_request(i))
13      %Check if a previous outstanding request stalls the functional module
14      for j = i - (Nmax-1) : i-1
15          if (Request j s currently outstanding and will get served ...
                before Request i)
16              if (Request j stalls the functional module)
17          %Then all requests from Request j+1 to Request i get derived ...
                benefit
18                  for k = j+1 : i
19                      Update derived_benefit(k)
20                  end
```

```
21              end
22          end
23      end
24
25      %Check if Request i stalls the functional module
26      if (Request i does not stall the functional module)
27      %Update to the current time
28      current_time = issue_time(i) + derived_benefit(i) + inter_request(i);
29      else
30          %Update derived benefit to all previous outstanding requests
31          for j = i - (Nmax-1) : i-1
32           Update derived_benefit(j)
33          end
34      %Update current_time;s
35      end
36
37  %If a Request j is issued in the no-stall interval of Request i
38   else
39      %Check previous Nmax - 1 requests
40      for j = i - (Nmax-1) : i-1
41          if (Request j is served before Request i+1 gets issued)
42      %Check if Request j stalls the functional module
43              if (Request j stalls the functional module)
44          %All requests from Request j+1 to Request i get derived benefit
45                  for k = j+1 : i
46                      %Update derived_benefit(k);
47                  end
48              end
49          end
50      end
51
52  %Update current_time
53   end
```