Resource Management and Performance Control for Staged design-Based Services

Vorgelegt von

M.Sc.

Mohammad Shadi Al-Hakeem

aus Damaskus – Syrien

von der Fakultät IV – Elektrotechnik und Informatik

der Technischen Universität Berlin

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften

-Dr.-Ing-

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Odej Kao

Berichter: Prof. Dr. Hans-Ulrich Heiß

Berichter: Prof. Dr. Gero Mühl

Tag der wissenschaftlichen Aussprache: 16.12.2010

Berlin 2010

D 83

بسم الله الرحمن الرحيم

In the name of Allah, the Beneficent, the Merciful.

To my wife Amani,

To my daughters Sham and Yamam,

To my son Mohammad Yamen,

To my Mother and to the soul of my Father,

To my Sisters and Brothers,

To my Friends,

To those I love and those who love me…

Abstract

The staged architecture has emerged as an approach to implement highly concurrent Internet services. Staging means that the functionality of the server code is broken down into computational stages with each stage performs some aspect of request processing. A client request would then be processed along a pipeline of these stages. This architecture allows services to behave well and gracefully handle overload, in addition to increase code modularity and simplify service design. However, Staged architecture has introduced other design challenges related to resource management and performance control.

A bottleneck stage in the work-flow of requests processing will limit the overall system throughput even though other stages are isolated from this bottleneck stage and can support a higher performance. For this reason care must be taken to avoid bottlenecks. A solution to adjust the throughput is to allocate more resources to a stage if it is becoming a bottleneck, this in turn may force other stages into becoming a bottleneck. Since all stages are competing for the same resources additional effects may take place and give rise to instability or oscillations.

Staged design was originally introduced as a programming abstraction to improve memory accesses behavior of highly concurrent Internet servers by implementing cohort scheduling policies which batch the execution of requests at each stage. As a result existing scheduling policies in staged design-based applications are mechanisms to increase the benefit from cache locality within the individual stages, rather than to balance resource allocation to avoid bottleneck stages and control the system performance. Consequently, dynamic changes in stages requirements lead to instability and oscillations in performance under different load conditions. In addition, achieving a target performance in such systems is a hard job and often depends on manual parameters tuning by expert administrators or benchmarks experiments.

To address these challenges, this thesis suggests a three-layers control architecture for resource management and performance control of staged applications based on the Staged Event Driven Architecture (SEDA), which is the

state of the art of the staged design. Then an adaptive resource allocation policy and a performance control approach are presented, which follow this three-layers control architecture.

The proposed approach benefits from the advantages of SEDA to support highly-concurrent demands and makes use of feedback-based controllers to manage the system resources and control its performance. The resource controller allocates resources to stages depending on run time observations of stages load and performance, and the feedback based performance controller adapt system parameters to achieve performance targets and guarantee the desired quality of service.

We validate the proposed scheduling policy and compare it with other scheduling policies under different load conditions through a simulation study. Results demonstrate that our approach can allocate system resources automatically and dynamically to achieve a superior performance while avoid performance degradation under overload. We demonstrate also the ability of the performance controller to adjust the system at run-time dynamically and automatically to maintain the desired performance target under a variety of dynamic changes in the system.

Zusammenfassung

Die Staged-Architektur ist als ein Ansatz für Internet-Dienste mit hoher Nebenläufigkeit entwickelt worden. “Staging” bedeutet dabei, dass die Funktionalität des Dienstes in einzelne Stufen zerlegt wird, die jeweils einige Aspekte der Berechnung ausführen. Eine Client-Anfrage wird dann entlang einer Pipeline dieser Stufen bearbeitet. Diese Architektur ermöglicht es dem Dienst, Überlast gut zu behandeln und erhöht zudem die Modularität des Codes und vereinfacht das Design. Allerdings haben sich mit Einführung der Staged-Architektur neue Herausforderungen im Design ergeben hinsichtlich der Ressourcenverwaltung und der Performance-Steuerung.

Eine Engpass-Phase innerhalb des Flusses der Verarbeitung der Anfragen reduziert den Gesamtdurchsatz selbst dann, wenn andere Stufen von dieser Engpass-Phase isoliert sind und eine höhere Leistung ermöglichen würden. Aus diesem Grund muss beim Design sorgsam darauf geachtet werden, Engpässe zu vermeiden. Eine mögliche Lösung ist es, der Engpass-Stufe mehr Ressourcen zuzuweisen, dies kann allerdings dazu führen, dass wiederum andere Stufen zum Engpass werden. Da alle Stufen um die gleichen Ressourcen konkurrieren, können zusätzliche Überlagerungen auftreten, die zu Instabilität oder einem Aufschwingen führen können.

Das Staged Design wurde ursprünglich als eine Programmierungs-Abstraktion vorgestellt zur Verbesserung des Speicherzugriffsverhaltens von hochgradig nebenläufigen Internet-Servern durch die Umsetzung von "Cohort Scheduling"-Ansätzen, die die Ausführung von Anfragen der einzelnen Stufen bündeln. Damit sind die bestehenden Ansätze des Schedulings im Staged Design mehr darauf ausgelegt, die Vorteile der Cache-Lokalität innerhalb der Stufen zu nutzen als die Ressourcenzuteilung zwischen den Stufen zu balanzieren und Engpässe zu vermeiden. Folglich führen dynamische Veränderungen in den Anforderungen der Stufen zu Instabilität und Schwingungen unter verschiedenen Lastbedingungen. Weiterhin ist es kompliziert, in solchen Systemen eine

vorgegebene Performance zu erreichen – dies basiert häufig auf manuellem Tuning der Parameter durch Experten oder anhand der Ergebnisse aufwendiger Benchmarks.

Um diesen Herausforderungen zu begegnen, schlägt diese Arbeit eine Drei-Schichten Architektur für Ressourcen Management und Performance Steuerung von Anwendungen vor, die auf der Stage Event Driven Architecture (SEDA, aktuelle Variante des Staged Design) basieren. Es werden eine adaptive Ressourcen zuweisung und ein Ansatz zur Performance-Steuerung vorgestellt, die dieser Drei-Schichten-Architektur folgen.

Der vorgeschlagene Ansatz profitiert von den Vorteilen der SEDA zur Unterstützung massiv nebenläufiger Nachfragen und nutzt eine Feedback-basierte Steuerung zur Verwaltung der System-Ressourcen und der Steuerung der Leistung. Der Ressourcen-Controller weist die Ressourcen den einzelnen Stufen in Abhängigkeit von zur Laufzeit getätigten Beobachtungen der Last und Performance der Stufe zu. Der Feedback-basierte Performance-Controller passt die Systemparameter dynamisch an, um Performance-Ziele zu garantieren sowie die gewünschte Qualität der Dienstleistung zu erreichen.

Der vorgeschlagene Scheduling-Ansatz wird validiert und mit anderen Ansätzen unter verschiedenen Lastbedingungen mittels einer Simulation verglichen. Die Ergebnisse zeigen, dass der Ansatz Ressourcen dynamisch zuweisen kann und eine höhere Performance erreichen kann, während eine Herabsetzung der Performance unter Überlast vermieden wird. Weiterhin wird die Fähigkeit des Performance-Controller demonstriert, die Systemparameter zur Laufzeit dynamisch und automatisch anzupassen, um das gewünschte Performance Ziel auch unter einer Vielzahl dynamischer Veränderungen im System zu erreichen.

Acknowledgment

Living in Berlin and doing this PhD has been a wonderful experience for me. I feel deeply indebted to a number of people who have contributed to my success, and I want to express my gratitude to them here.

First and foremost, I would like to thank my supervisor Prof. Dr. Hans-Ulrich Heiß for giving me the opportunity to be in his research group and for his guidance and patience until finishing this thesis. I hope to be able to emulate him as a teacher and as a researcher. I am also grateful to Prof. Dr.-Ing. habil. Gero Mühl, and I would like to thank him for taking interest in my work and providing valuable comments. Special thanks to Dr.-Ing. Jan Richling for supporting me in the different phases of this thesis and for the many fruitful discussions. I owe a great deal of thanks to Prof. Dr. Bernd Mahr. I will not forget our first meeting in Damascus, which opened the door for me to complete my PhD in Berlin. I like also to thank all my other colleagues in the Communication and Operating Systems Group. In particular, I would like to thank Gabriele Wenzel and Jessica Krueger who are not from the researching stuff, but they was very kind and helpful to me.

The biggest thank is to my wife and children for supporting me in the many hard times during my work and encouraging me to pursue. I love you all and thank you very much.

Mohammad Shadi Al-Hakeem.

Contents

List of Figures

1 Introduction

1.1 Motivation

Servers[1] are broad class of computer programs, that are combined with a hardware platform in order to provide and manage accesses to special services and shared resources on behalf of clients; such as file systems, databases, mail stores, web sites, etc.. Servers receive a stream of clients' requests, process each request and produce a stream of results. The performance of these servers is very important, as it determines the latency to access the resources and to reply clients requests, and constrains the server's ability to serve multiple clients. For this reason, the performance of commercial servers, such as database and file servers, has been the focus of considerable research, which improved the underlying hardware, algorithms, and parallelism of these servers, as well as considerable development which improved their code.

However, the phenomenal growth of the World Wide Web "the explosion of the Web", the rise of modern Internet services and the characteristics of these services give network servers performance additional importance and present a number of new design challenges.

Internet servers demand massive concurrency and have to be well-conditioned to load, as they are subject to large load oscillations with load peaks that are multiple of the average. In addition, the critical nature of many on-line services and the increasing interest of servers administrators to maximize their clients satisfaction while efficiently using existing resources increases the complexity of performance requirements. Servers must maintain peak throughput, avoid degradation of performance and behave predictably even when demands ex-

[1] This term "server" is also used widely today to refer to the computer that runs the server (software), although it gets the name from the task of the software not from its special architecture [178]. To avoid confusion, the term server-machine or server-computer will be used to explicitly designate the hardware platform used to run the server software.

ceed available resources capacity.

To cope with the high concurrency in Internet services, servers must accommodate thousands or even more of simultaneous client connections. Designing such massively-concurrent systems is difficult especially when high and controlled performance requirements must also be met. These challenges have made the design of high performance servers a recent research thrust, to meet the increasing popularity of Internet-based services.

As mentioned, Internet servers have to be designed to handle very large numbers of clients connections and even larger numbers of messages or requests per second. In order to serve these clients simultaneously a server must be able to process their concurrent requests in parallel on the server hardware resources by time and space sharing the available resources (CPU Cycles, Memory, Network I/O, etc.) among requests.

Many approaches for building and managing such concurrent systems have been proposed, which can be broadly categorized as event-driven programming approaches and thread-based programming approaches (also referred to as thread-per-connection). Event-driven and Thread-based are two traditional and prevalent implementation strategies which have been successful for building concurrent systems[2]. Which of them is “better”, that is a debate which has waged for many years, with almost no resolution [104, 126, 167].

However, although several techniques have been proposed and adopted to enhance both approaches and improve their performance, researchers have argued that both concurrency models show many drawbacks and limitations. For this reason, these approaches are unable to supply the requirements and fail to introduce the controlled performance that are needed in today Internet applications. Both approaches have their advantages and disadvantages; threads and events are the two opposite ends of a design spectrum, and the best implementation strategy for today applications is somewhere in between. This fact gives the rise to hybrid systems which appear in the recent few years and exploit properties of both approaches. These hybrid approaches operate

[2]They are called also “Message-oriented System" and “Procedure-oriented System", respectively [104].

in the middle of the spectrum and utilize both threads and events as a tool to develop the high concurrency which is required for today Internet services [107, 173].

In order to provide services, servers have to be combined with a hardware platform. The structure of this platform is another important issue that must be considered in the process of servers design, and which has a great influence on server's performance. As any application specific design should exploit the novelty of the architectural trend of the state of the art hardware, the trends in computer architecture technology have to be taken in account in order to benefit from these new architectures. Hardware systems are only as effective as the software's ability to take advantage of these systems. However, the architecture of today's servers do not give us what we pay for and do not show the expected gains in performance [39]. A big gap exists between the theoretical and the actual performance of server computers, and it is the task of the software community to develop techniques that improve resource utilization and to introduce architectures for server applications which can bridge this gap.

Today, web content is increasingly generated dynamically, secure connections are required, network bandwidth increases and main memory is becoming cheaper and large enough to replace disks as the storage unit for active server data. As a result, processing resources arise as a performance bottleneck in comparison to network transfer and disk I/O [17, 73, 119]. On the other hand, the processor-memory performance gap continues to grow and the emergence of multi-core and multi-threaded processors decreases the efficiency of cache hierarchy [29, 78, 156]. Altogether, this makes memory accesses one of the most dominating factors in server performance.

Many researchers have reported about a performance bottleneck related to memory access behavior in both software architectures; thread-based approach [51, 78, 103] and event-driven approach [29, 30]. Switching to another thread in the thread-based concurrency and handling another event from the event queue in the event-based concurrency, both result in frequent control transfers between unrelated pieces of code which decreases instruction and data locality and therefore reduces the effectiveness of hardware caches. This results in limiting scalability and performance of servers as often only a fraction

of modern processor's computational throughput is utilized. Eliminating these effects of memory accesses is an important issue to design highly concurrent and high performance systems for Internet services.

Staged architecture is one of the hybrid approaches that has been presented as a general purpose design framework for building highly concurrent systems, which encapsulates the concurrency, performance, and software engineering benefits of both threads and events. In addition this architecture introduces a programming abstraction to implement cohort scheduling policies which are potential to avoid misses in cache hierarchy and eliminate the effect of memory accesses behavior on the performance of Internet services [103]. Cohort scheduling policies increase code and data locality by batching the execution of similar operations arising in different server requests.

In the staged design, a service is implemented as a network of computational stages connected with explicit queues. Cohort scheduling policies can be implemented in this design by batching the execution of requests at each stage. Although this combination of the staged architecture with cohort scheduling policies has the potential to improve the cache behavior, it introduces new challenges related to resource management and performance control for staged design-based applications, which make this architecture recently an active research field [70, 109, 110, 172, 173].

A client request in a staged service is processed along a pipeline of stages. A bottleneck stage in the process flow will limit the overall system throughput even though other stages are isolated from this bottleneck stage and can support a higher throughput. For this reason, care must be taken to avoid such bottlenecks. A solution to adjust the throughput is to allocate more resources to a stage if it is becoming a bottleneck, this in turn may force other stages into becoming a bottleneck. Since all stages are competing for the same resources, additional effects may take place and lead to instability or oscillations.

Another concern is to adapt the system to achieve performance targets, taking into account the characteristics of the staged design and the target of improving the behavior of memory accesses. An important aspect here is to control the dependencies between the number of requests processed as a batch at

each stage and the different performance goals.

However, existing cohort scheduling policies do not take the characteristics of the staged system and these performance objectives into account. They are usually heuristic approaches, aimed at increasing the benefit from cache locality within a stage rather than optimizing the overall system performance by managing the available system resources to avoid bottleneck stages in the work-flow.

As a result these policies introduce coarse grained resource allocation approaches which increase the difficulties related to controlling the system performance. Configuring such systems to generate a target performance requires experienced administrators to correctly set the control parameters or determining these parameters experimentally using benchmarks which is a very difficult, time consuming, and error-prone manual operation.

Motivated by the above comes this thesis...

1.2 Thesis Summary and Contributions

Considering the staged design and the previously mentioned challenges that appear in this design, the main contribution of this thesis is to present a resource management scheme that combines the staged design with a "Cohort Scheduling" policy, in order to reduce the effects of memory accesses behavior. The proposed resource management scheme has to take many consideration that are related to staged design into account. Differentiations in the requirements of the individual stages, dynamic changes in these requirements and other dynamic changes in the system have to be addressed. In addition, the characteristics of today hardware platforms have to be considered to maximize the benefit from performance improvements that are presented by the architecture of these platforms. Furthermore, performance control and performance level guarantees that are required in today services have to be provided.

To address these requirements, the thesis firstly suggests a control architecture that consists of three layers of controllers. After that, the thesis presents a resource allocation policy and a performance management approach that fol-

low these three-layers control architecture.

The proposed resource allocation policy depends on batch processing to benefit from locality within stages, and allocates the available system resources depending on run-time observations of the requirements of the individual stages. The policy considers also the characteristics of the parallelism hierarchy in the underlying hardware platform, in order to optimize the allocation of processing resources.

The performance management approach depends on a feedback-based controller to adjusts the system at run-time to achieve performance targets and to guarantee the specified performance levels. We depend on the proposed resource allocation policy to present a robust system model, then we implement this model in a control-theoretic feedback-based controller to adapt system parameters, in order to achieve performance targets.

We evaluate the proposed approach using a simulation study, by comparing its performance with other resource allocation policies that have been presented for staged design based services. We also present performance evaluation results that demonstrate the ability of our performance controller to maintain performance targets under a variety of unpredictable changes in the system.

We argue that the proposed approach effectively provides resource management and performance control for staged design based services, while reducing manual operations and off-line management overheads that characterize existing approaches.

1.3 Dissertation Road Map

The dissertation is organized as follows.

In Chapter 2, we present a background about the characteristics of modern Internet services, an overview of the characteristics and the trends in today server machines, a brief explanation of conventional server design techniques and the challenges that have to be faced by these techniques. Chapter 3 explains the staged design in details and presents the challenges and limitations

in the staged design that we address throughout this thesis. In Chapter 4, we review related work on the Staged design and other similar programming models. In addition, we present a review of Cache Conscious Scheduling, performance control mechanisms for Internet servers and related researches that have been introduced for staged design-based services.

Chapter 5 presents a general three-layers control architecture for staged design-based applications, explains the layers of this control architecture and introduces an overview of the proposed approach for resource management and performance control which follows this three-layers control architecture. In Chapter 6, we present a strategy to allocate processing resources in staged design based services, which adapts the resources assigned to each stage based on observations of the changes in the system. The chapter presents also a validation and evaluation of the strategy through a simulation study. Chapter 7 demonstrates an adaptive control approach that automatically manages system resources in order to control the performance for staged applications, and finally, Chapter 8 summarizes our results, presents many directions for future work and a variety of potential implementation areas.

2 Background and Overview

Internet has been showing a phenomenal growth in its size and an increasing popularity of its services. These characteristics, in addition to the increasing demand for performance and service quality, give rise to unprecedented system design challenges for service's providers and designers. This chapter presents an overview of the characteristics of modern Internet services, an overview of the characteristics of today server's hardware platforms, a brief explanation of conventional server design techniques and the challenges that have to be considered when such systems are to be employed.

2.1 Internet Services - Characteristics and Trends

2.1.1 The Growth of Internet and the Rise of its Services

After decades of the Internet evolution, and after the advent of the World Wide Web in the early 1990s, there has been a phenomenal growth of the Internet in size and use.

Netcraft Web Server Surveys have covered the dramatic increase in the number of Internet sites since August 1995, see Figure 2.1. A recent survey of February 2009 has reported receiving responses from more than 215 million sites with a month's gain of more than 30 million sites [124]. DomainTools statistics have also shown that more than 110 million active domains exist [57], and Google reported the existence of more than 1 trillion unique URLs on the web at ones [16].

The number of Internet users and households with access to the Internet is also growing at amazing rates worldwide. According to a statistic from Internet World Stats [92], the world Internet users growth since the year 2000 was more than 340 percent[1]. The same source reports that currently almost 1,574

[1]This number increases to 11000 percent in Syria [92].

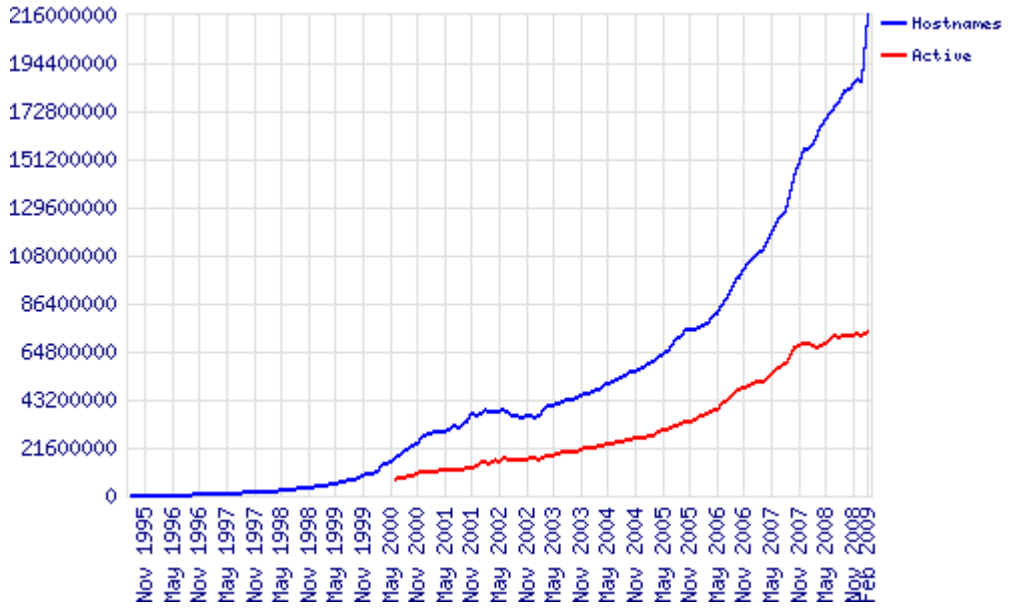


Figure 2.1: Total Sites Across All Domains August 1995 - February 2009

billion people are using the Internet world wide, which is more than 23 percent of the total world population. A study from Parks Associates reported that homes with broadband connections worldwide had grown by over 18% in 2008 exceeding 400 million. The firm claims that by 2013 the households globally with broadband Internet access will be over 640 million [130].

In fact, the expansion of the Internet infrastructure and the ease with which resources on the Internet could be published and accessed increase the popularity of Internet-based services, causing them to experience extremely fast growth. Today, business and individuals are increasingly depending on these Internet services for day-to-day operations. Many of these services, such as e-mail, on-line news, social networks, on-line auctions, e-commerce, etc., have become a vital resource and considered indispensable for many people – "as water and gas" [36]. For instance, Windows Live Hotmail has over 260 million users worldwide since February 2008 [147], Facebook has more than 200 million active users, and eBay and Amazon has more than 50 million unique visitors [150].

2.1.2 Internet Services Properties

As a result of this "explosion of the web" and the enormous, continuously increasing popularity of services over the Internet, these services have shown many characteristics that distinguish their behavior from traditional computer systems.

The rising huge number of Internet users causes Internet sites to be subjected to extremely large number of clients visits received per day, which are translated into an even greater number of concurrent requests and operations. As a site becomes popular, users accesses increase dramatically, and demands can reach hundreds of millions of requests a day. For example, in December 2008, more than 5,4 billion search queries were conducted at Google search [158]. In the same month, Facebook and MySpace had 80 billion and 43 billion monthly page views, respectively [21].

Compounded with the randomness associated with the way users visit Internet sites and request Internet services, this huge volume of clients can cause the incoming service's workloads to vary significantly and unpredictably, even within the same business day [41, 50]. The resulting peak workload experienced by a service may be many times that of the average. Figure 2.2, for example, which is taken from Alexa.com [12], shows how the daily reaches on the web site of "Aljazeera news channel" have been increasing continuously over time. The figure shows also large oscillations in load over a variety of time scales, with a very huge load peak on march-2003, that was caused by the large number of visitors during the "Iraq occupation war".

Moreover, the impact of these massively concurrent requests has been magnified even further by many other features and characteristics of today Internet services.

In the early days, the Web was dominated by the delivery of static contents, mainly in the form of static HTML pages and images files. Recently, the advent of new technologies, including on-line banking, on-line trading and more recently Web 2.0 applications [180], gives rise to a variety of new Internet-based services. The content of these modern services is often generated dynamically on-the-fly, and it may include server side scripting (like CGI, PHP[159],

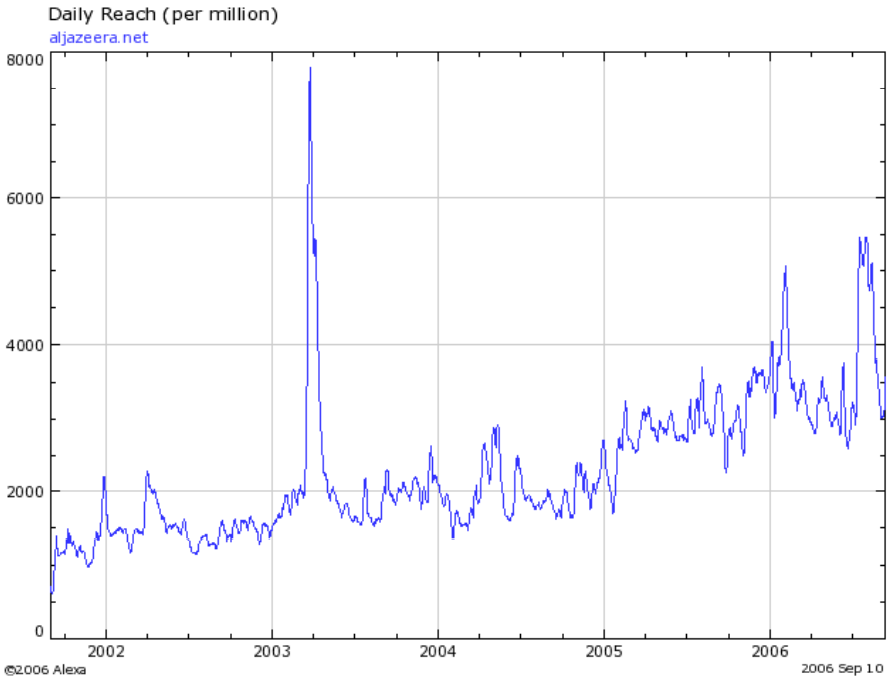


Figure 2.2: Traffic History Graph for Aljazeera.net

ASP[118], Java EE [154], etc.) and/or database accesses. For each request, a server application executes the service's corresponding code, generates the result and assembles the response, which is returned to the client. In comparison to static content, dynamic content requires more operations, greater resources and significant amounts of computation and I/O to be generated.

In addition, the rapid expansion of e-commerce and the corresponding security concerns give the rise to security requirements when accessing sites contents and exchanging sensitive information. All information that has market value or is considered confidential must be carefully protected when transmitted over the open Internet. Consequently, Internet servers need to provide certain levels of security so that the user feels comfortable when running the applications that provide the requested services. Although, most of the used solutions[2] do not introduce additional complexity in the applications structure, they increase the computational demands on the servers resources remarkably [72, 94]. Figure 2.3 taken from the Netcraft SSL survey [132], shows how the number of

[2]The most used solution is Secure HTTP (called HTTPS). It uses SSL for security and to protect sensitive transactions (e.g., order placement, payment).

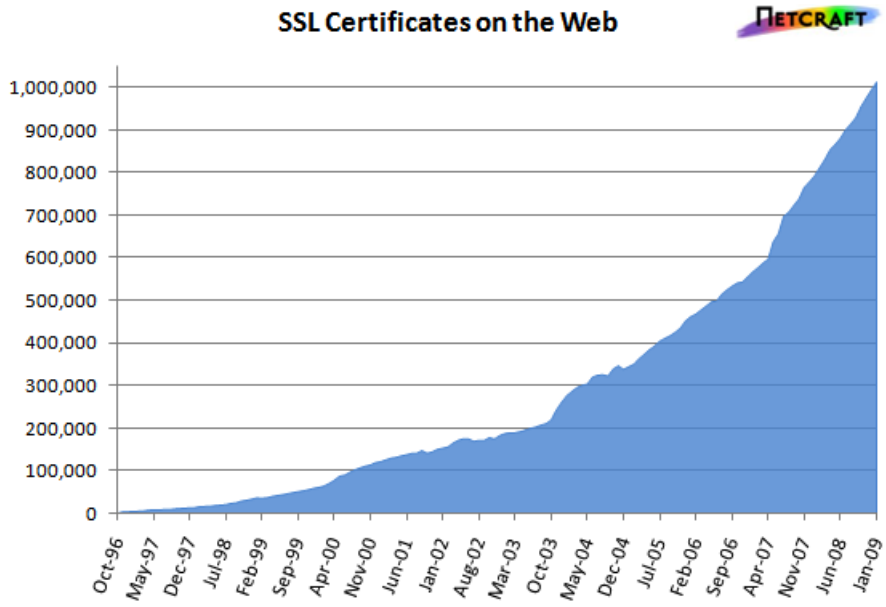


Figure 2.3: SSL Certificates on the Web

SSL certificates has increased since October 1996. The survey reported an average growth of more than 18,000 certificates per month in the last year.

Another special feature of the Internet is the presence of a large population of robots that interact with servers and web sites, like crawlers, price-bots and other autonomous software agents. These robots increase server loads and consume significant amount of system resources as they dig out the information [14, 15]. Search engines, for instance, demand exhaustive crawling work to maintain and update their indices to the very large collection of documents on the web. Unfortunately, there are also some malicious robots or spiders. The most common of these are ones designed to implement Denial-of-Service (DoS) attacks, that are accomplished by flooding a server with a very large number of concurrent requests such that the server becomes too busy and the provided service is no longer available for normal users. Many such attacks have been recently reported on very popular sites, like Twitter, Facebook, and the White House website [49, 139].

As a result of these characteristics, demands on Internet servers are unpredictable and exhibit fast growth, which raise a variety of problems in designing, building and operating on-line services. Providers and designers of such

services face challenges to meet clients expectations, like availability, reliability and trustworthiness, and in recent years many high-profile companies that provide on-line services have experienced operational failures [23, 58].

2.1.3 Internet Servers Design Challenges

Today, when providing an Internet service, server's scalability and performance are key attributes that should be considered. Scalability means the ability of the server to maintain the service availability, reliability, and performance as the amount of load, or simultaneous requests, hitting this server increases [32, 75]. However, as a result of the previously stated characteristics of Internet services and their workloads, meeting these requirements is very difficult and presents system design problems with unprecedented challenges.

To cope with the rising number of Internet users, servers that host popular Internet services, have to support large numbers of clients connections and scale to high levels of concurrency. These servers must have enough resources capacity in order to handle the massively concurrent requests in a reliable, responsive and always available manner. In addition, since demands increase continuously, providers need to adjust their infrastructure periodically in order to accommodate these additional demands. Otherwise, if the available resources remain unchanged, requests have to be rejected, and corresponding revenues are lost [13].

At the same time, Internet services are subject to enormous variations in workload, which happen over a variety of time scales, making forecasting the needed resources a difficult task. Although, certain workload variations such as time-of-day effects could be predicted and handled by appropriate capacity provisioning [82, 115], other variations, such as flash crowds or "Slash-dot Effect"[3], can cause huge load spikes, which cannot be accommodated with traditional capacity planning practices and may affect the availability of the service. See for example what happened at the first launch of the Photosynth site [23, 64].

It is not uncommon for a site to experience orders of magnitude increases

[3] Also known as slash-dotting, the term is often used to describe the phenomenon when a site is suddenly hit by heavy load. This term refers to the technology news site slash-dot.org, which is itself very popular and often brings other smaller sites to slow down or even temporarily close when linking to them from its main page.

in demand when it becomes popular. Even popular sites could be subjected to unexpected load peaks that are orders of the average and usually coinciding with times, those are the most important to be able to get the service [83]. For example, the load on e-commerce retail Web sites can increase dramatically during the final days of holiday seasons [47], and it may be very difficult to access a major newspaper or TV site when big news breaks due to site overload. An important event can cause services to experience huge and unexpected volume of crowds, like the share market's black Tuesday on 27 Feb 2007, which caused many electronic stock trading sites worldwide to clash for hours [143], and the death of Michael Jackson which caused a record workload on many news sites and resulting in many problems [131]. Another example is the heavy snowfall in December 2009, which caused the website of the National Rail in Great Britain to be unavailable as it has been unable to cope with the large number of visitors eager to know which train services may be delayed or canceled [122].

To avoid overload effects which may result from these load variations and cause the service to behave erratically or even crash, many services rely on over-provisioning of server resources to handle load peaks. However, it is clearly infeasible to over-provision a service to handle spikes in load that are multiple orders of magnitude greater than the average, especially when considering budget constrains which limit the space of possible solutions.

Moreover, the widespread of services with dynamic contents, like on-line stores, auction sites, bulletin boards, etc., reduces the server capacity and increases the time to serve clients. In comparison to servers whose workload is dominated by static files (HTML pages or images), servers with large proportion of dynamic content perform worse. For this reason, types and compositions of workloads are also needed to indicate the server load correctly. In addition, such services are increasingly compounded with mechanisms needed for supporting secure communications between clients and servers, like SSL protocol which provides communications privacy over the Internet and is widely used in e-commerce environments. These mechanisms increase the demand on server computational resources remarkably, due to the use of cryptography to fulfill their objectives [72, 94]. Taking into account the increasing performance demand, and that security issues rise considerably amongst the world's busiest

services [127], altogether, that magnify the scalability problem and make the capacity planning task more challenging, as the resource requirements for a given user load are more difficult to predict. Either under-estimating or over-estimating a server's capacity could cause unnecessary expenses, delays or potentially disastrous consequences.

In summary, Internet services that scale to on-line rates of simultaneous unpredictable client connections, and accommodate their massively concurrent requests, are difficult to implement and present a number of unprecedented system design challenges, especially when other requirements such as performance and security issues, must also be met.

2.2 Server's Hardware Characteristics

Internet services are provided by a combination of software components and hardware platform. Each of these has a great influence on the scalability and other performance aspects of a service. From the hardware standpoint, a service performance is affected by the number, structure and speed of the underlying server machine processors; the amount of its main memory; the capacity and the bandwidth of its storage sub-system and the bandwidth of its network connections.

This section presents an examination of the characteristics of today server computing systems, as they relate to the performance aspects of Internet services.

2.2.1 Bottleneck Resources

Serving a request consumes different server resources, such as memory, disk bandwidth, communication bandwidth, and processing cycles. The ability of the server to handle multiple of these requests and its performance is limited by the bottleneck resource in the system.

Today, network bandwidth increases, as a result of the advent of more sophisticated networking technologies [45], and the growing investments in the field of high-speed communications [22, 133]. Although many experts and observers warn of potential Internet capacity problem because of certain bandwidth-

intensive applications [145], like high-quality video transport, peer to peer file sharing, massive multi-player on-line games, etc.; Statistics still show good performance, concerning Internet traffic speed and loss rates [18], and concerning Internet bandwidth utilization [134].

Main memory is becoming cheaper and larger in size [97, 125], modern server machines can support more memory slots per processor socket [46], and systems with more number of sockets are presented. Consequently, modern server platforms can have enough main memory space for active server data [87], especially when many considerations related to the characteristics of Internet workload are taken in account. The concentration of references [20], particularly in the case of flash crowds, and the fact that the relative frequency of clients request accesses to the web content follow Zipf distribution [35], that decreases the size of active data. For example, although the database of an auction website itself may be relatively large, the concentration of accesses to the information in the database causes the active data size to be relatively small. Clearly, not all the documents of a web site are equal. Some are extremely popular, accessed frequently and at short intervals by many clients. Other documents are accessed rarely, if at all. Some documents receive thousands or even more of requests, while others receive relatively few requests. It have been reported that in average less than 10% of the distinct documents of a web site are responsible for 80-95% of all requests received by this site [20, 43, 157].

In addition, as previously stated, web content is increasingly generated dynamically, which, in comparison to static HTML pages, uses more processor cycles [119]. Secure connections are increasingly required, using SSL –for instance, as a mechanism to implement secure connections, which can increase the processing overhead by a factor of 5-7 [72, 94].

All together that give the rise for processing as a performance bottleneck in comparison to network transfer and disk I/O, in a variety of Internet applications, as reported by many studies [17, 72, 121]. For this reason, and in order to obtain the processing power that is needed to cope with the massive concurrency and the increasing demands for performance aspects, today's servers often employ a variety of hardware techniques, like Multi-Processor systems,

Clusters [26], etc.. However, in addition to the budget constrains problem, using these techniques increases the complexity of system design as other issues, like parallel processing related challenges, must be taken into account.

2.2.2 Processor-Memory Speed Gap

From 1986 to 2000 micro-processor design had moved rapidly, doubling clock-speed and performance of CPUs almost every two years. This rate of improvement in micro-processor speed exceeded the improvement rate in Dynamic Random Access Memory (DRAM) speed, since the memory arena, which had focused on increasing density and lowering cost, could not achieve the same performance increases. This disparity resulted in a performance gap between processor and memory, which causes considerable wait time that execution threads have to incur when accessing memory, and as a result impacts processor efficiency and the overall system throughput dramatically. Hence, this processor - memory performance gap was considered one of the primary obstacles to improve computer systems and it was expected that memory latency would become a bottleneck in computer performance [113, 181]. Currently, CPU speed improvements have slowed significantly due to many causes that are summarized by Intel in their Platform 2015 documentation [136], and which are majorly related to physical barriers. However, this performance gap still plays a main role in computer systems performance, which is growing now as a result of the chip multi-processing trend –See the next sub-section 2.2.3. In Figure 2.4 from [84], the gap in performance between memory and processors is plotted over time, starting with 1980 performance as a baseline. The memory baseline in the figure is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency. The processor performance line shows a 1.25 improvement per year until 1986, a 1.52 improvement until 2004, and a 1.20 improvement thereafter.

Many hardware techniques have been implemented, such as caches, translation look-aside buffer (TLB) [148], branch prediction and out of order execution, in order to bridge this performance gap and to hide the latency of long memory access operations. These techniques are based on cache memory and exploit the principle of locality, which characterizes memory accesses [175]. Cache memory is a smaller, more expensive, but faster memory in comparison to DRAM, which stores copies of data from the most frequently used main

memory locations. As long as most memory accesses are to the cached memory locations, the average latency of memory accesses will be closer to the low cache latency than to the high latency of main memory.

However, benefits of these conventional techniques are limited and they fail to work well for modern applications, such as application servers, web services, and on-line transaction processing systems. These applications usually include multiple threads of control that execute short sequences of operations, with frequent dynamic branches. The behavior of these structures decreases cache locality and branch prediction accuracy and as a result causes frequent processor stalls, resulting in very poor processor resources utilization and wasting significant processing time [9, 27].

The latency difference between main memory and the fastest cache has become larger, because of the evolution of cache memory. For this reason, processors have begun to utilize multiple levels of cache, with each level takes considerably longer to be accessed than previous levels. The multiple level cache has been also presented as a solution to address the trade off between hit rate and cache latency. A larger cache may have a better hit rate but it will have a longer latency. In a system with multiple cache levels, small fast caches will be backed up with larger but slower caches. Modern processors have as many as three levels of cache. For example, AMD Opteron has 64+64 KB L1 data and instruction cache and 512 KB L2-cache (private cache per core) and 12 MB L3-cache (shared between cores) all on chip caches [7, 69]; Intel Xeon has also three on chip cache levels with 32+32 KB L1-cache and 256 KB L2-cache (private cache per core) and 12 MB L3-cache (shared between cores) [69, 91]; and IBM Power7 has 32+32 KB L1 instruction and data cache per core, 256 KB L2- cache per core and 32 MB L3-cache that can be used either as shared cache or separated into dedicated caches for each core [86, 123].

Recently, other techniques have been also developed to cope with modern applications needs and to improve processors performance even further, like multi-core chips and hardware multi-threading. Multi-threaded processors interleave the execution of instructions from different threads, so that if one thread blocks on a memory access or some other long operation, other threads can continue execution and make forward progress. Numerous studies have

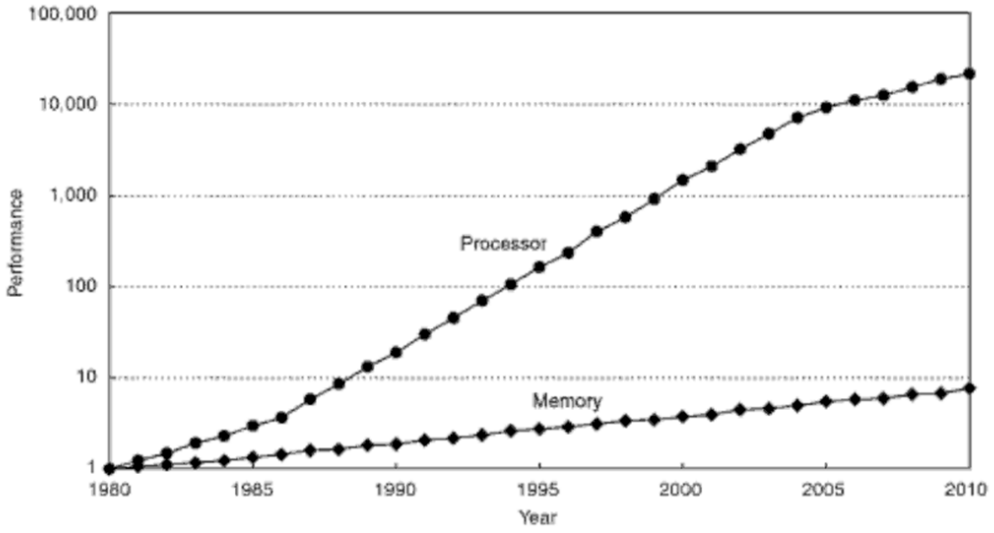


Figure 2.4: The Memory Wall.

demonstrated the performance benefits of these techniques [33, 114, 153, 163]. However, as multi-core and multi-threaded processors include multiple thread contexts on a single chip that are simultaneously active, the competition for shared resources, which typically include shared caches, is more intensive [63]. As a result, that increases the problem of waiting for memory, which exists in processors that have a single execution thread.

Although processors are increasingly equipped with larger caches, this growth in size causes an increase in latency which means slower caches. On the other hand, as the cache is shared among more contexts (cores and threads) that means also smaller cache per execution thread. For instance, Intel Core i7 has 8 MB L3-cache shared across four cores with a latency of 35 ns, rather than 6 MB L2-cache shared across just two cores with a latency of 15 ns in Intel Core2 [155].

Moreover, in multi-processor systems, which are the typical platforms for server's machines today, cache misses (data footprint and coherency misses) increase even further with the number of processing units, and with the advent of system growth beyond single system (CPU/memory) boards Uniform Memory Access (UMA) could no longer be guaranteed –this issue is discussed in more details in the next section.

2.2.3 Chip Multi-Processing Era

As mentioned earlier, processor's performance have been doubling approximately every two years according to Moore's law [120], by increasing the count of transistors on a chip. Many other techniques have been also employed to increase the throughput, improve the efficiency and better utilize the resources provided by the processor architectures. Examples of these techniques are caches, pipe-lining, super-scalar architectures, and simultaneous multi-threading technology, which allows multiple threads to execute in parallel on the same processing unit, with instructions from multiple threads able to be executed during the same cycle [114, 163, 164].

Recently, as the potential improvement in processors clock speed is achieving its limits, because of the physical limitations of manufacturing, the concerns of energy consumption and the related heat issues [136], processor chip manufacturers have turned towards multi-core processors[4] [65], like AMD [6], Intel [89], IBM [85] and Sun Microsystems [152]. Higher-frequency processors waste a lot of power, generate so much heat and as a result these faster processors need new, usually more expensive, techniques to properly cool the systems in which they are running. These concerns are becoming increasingly very pronounced in green computing campaigns. For these reasons sharing some of chip resources among multiple core appeared as a necessity for efficiency and economy.

The idea behind multi-core technology is to change from the trend of just increasing the speed of processors to a new design strategy that is to include two or more processors (processing cores) together on a single chip. This technique allows more than one thread to be active at a time, which increases on chip parallelism and creates an on-chip network or an SMP-like system on the chip. As a result, that improves the utilization of chip resources, obtains further performance gains, and reduces energy consumption.

As designers are fast moving towards multiple cores on a chip to achieve new levels of performance improvements, all processor vendors offer CPU models of

[4] On October 1989 Intel published a paper with the title " Microprocessors Circa 2000" [67], where it previewed that for the end of the year 2000 there will be offered multi-core processors on the market, which becomes a reality fifteen years later.

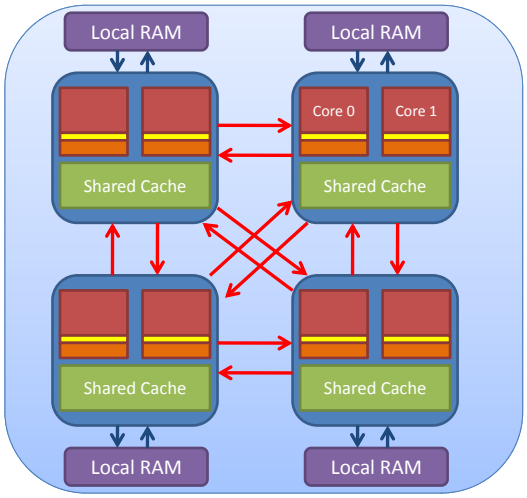


Figure 2.5: A Typical Topology for an SMP Server that is based on NUMA.

this design, and the majority of newly released CPUs are chip multi-processors. Consequently, multi-core processors have become the dominant architecture for a wide spectrum of platforms, especially for server class machines, and they are expected to be so in the coming years. This trend gives the rise to a hierarchic parallelism in computing systems, which consists of a number of processing chips, with each chip contains a number of cores (that are likely to increase per chip) and maybe multiple hardware threads within each core.

Moreover, the depth of this parallelism hierarchy becomes even larger, by the Non-Uniform Memory Architecture (NUMA) of today systems. As the number of processing units increases, chips per machine and cores per chip, memory-bandwidth utilization increases dramatically, and the scalability of the memory controller becomes an important issue that limits the performance of these processing units. NUMA presented a solution for this effect. In modern systems each socket in a multi-socket system has its dedicated main memory, and recently memory controllers have been moved on chip, in order to improve memory accesses for multi-core processors. An integrated on-chip memory controller is more efficient and more compatible with this NUMA memory architecture.

Figure 2.5 shows the typical topology of an SMP server that's based on the NUMA architecture and Figure 2.6 shows the corresponding parallelism hier-

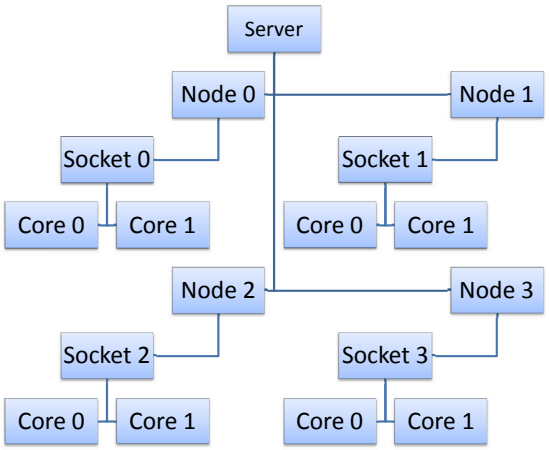


Figure 2.6: Parallelism Hierarchy

archy. The server here contains four multi-core chips (dual-core in this Figure) and each socket is connected to a local random access memory (RAM).

A key difference in these systems is that they introduce a non-uniform data accessing overheads, which differ depending on the physical location of data. Modern server machines, like those based on AMD Opteron or Intel latest generation Xeon processors, are NUMA multi-socket systems with a multi-core processor on each socket that has an on-chip cache which is shared among the cores (L3-cache – inclusive or exclusive cache), and a per core private cache (L1- and L2- cache). In this architecture, it is more expensive for a processing unit to access data that resides on the shared cache than to access data that resides on the private cache. Furthermore, it is significantly more expensive to access data that reside in the cache of other chips or in the main memory, and it is even more expensive to access data that reside in the main memory of another chip.

Looking in the future, the depth and complexity of this parallelism hierarchy will continue to increase as future systems are expected to have more processors, more cores and more threads per core. Examples of these future processing system generations are presented by the “Intel’s 80-core Polaris research processor”, the “Tera-Scale Computing Research Program” [80, 137] and the recent 48-core Single-Chip Cloud Computer [90].

Considering this trend, the problem is that, to ensure that our systems run well in the future, it is important that software community develops techniques to improve resource utilization of these systems and support this kind of parallelism.

2.3 System Design Techniques

In order to address the scalability, robustness and other performance challenges faced by Internet services, servers have to be designed to handle a very large number of concurrent clients connections and even a larger number of requests, that changes over time. To serve these clients, a server must be able to process their concurrent requests simultaneously on the server machine by time and space sharing the available resources (CPU Cycles, Memory, Network I/O, etc.) among requests. However, supporting concurrency for a few tens of clients is fundamentally different than for many thousands of service requests. A key aspect here is the means by which requests concurrency is represented.

Several different server design approaches have been proposed in the literature to overlap multiple clients requests and for managing these high levels of concurrency. The major strategies used to construct high performance servers can be broadly categorized as event-driven concurrency and thread-based concurrency. Many researches have compared these different concurrency alternatives [8, 96, 104, 126, 129, 167], and the debate over which approach is better has waged for a long time with almost no resolution.

2.3.1 Thread-Based Concurrency

A thread is a context of execution. Using multiple of execution threads has emerged from a long time as a leading solution for the development of applications with demanding performance requirements. This design was primarily derived from the multi-programming paradigm, which allows multiple applications each with distinct resource demands to efficiently share a set of resources, by presenting threads as an abstraction of processors.

A large number of popular computing system designs in current use are developed depending on the thread-based concurrency, and this model is the most commonly used in today server applications in order to provide the perfor-

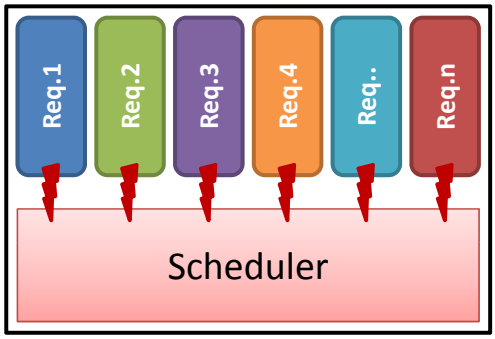


Figure 2.7: Thread-Based Concurrency

mance requirements of the highly concurrent Internet services. For instance, thread-based concurrency form today the basis for conventional web servers such as Apache [19] and Microsoft Internet Information services (IIS) [116], which are running on more than 80 percent of the world's web sites – as reported in the recent Netcraft surveys [124].

In the thread-based concurrency approach (Figure 2.7), a worker thread is assigned to each accepted request. This thread is consumed by the request and performs all the steps associated with request processing independently, with synchronization mechanisms, using locks, condition variables, or other synchronization primitives, to protect shared resources. Each thread executes until it either blocks on a synchronization condition, or an I/O operations, or until a predetermined time quantum has elapsed. Then, the execution switches to a different thread. Since multiple threads are employed, many requests can be served concurrently, enabling the system to overlap I/O operations with computations as the operating system switches among threads transparently. As a result, that increases resources utilization, and gives the opportunity to fully exploit additional speedup when multi-processor platforms are used.

Although straightforward to implement and relatively easy to program, as it is well supported by modern languages and programming environments, like Java, many researchers argue against the scalability of the thread-per-request design when handling large concurrent loads. This concurrency model has severe performance drawbacks due to the overheads associated with resource

contention and threading, which include cache and TLB misses, context switches, scheduling overheads, lock contention, etc. [126, 128, 170, 173]. Given the extreme degree of concurrency required for Internet services, these overheads and the overhead caused by the management of a high number of threads that present in the system will limit the maximum achievable throughput for a threading-based server. Moreover, these overheads typically lead to an overload behavior such that as load increases the system performance first increases, reaches a maximum and then declines. When the number of concurrent threads in the system increases over a certain degree, application throughput degrades severely and response time increases dramatically, thereby limiting the system's capacity and ability to support highly concurrent requests.

To avoid this effect of the over-use of threads, a number of systems associate a pool of execution threads with a service that continuously picks requests from the network queue and adopt a coarse form of load conditioning that serves to bound the size of this thread pool. When a request arrives, the server uses a free thread in the pool to serve this request, and returns the thread to the pool after finishing the request. When the number of requests in the server exceeds the thread pool size limitation, no more extra threads are allowed to be added into the thread pool. In this case, additional connections are not accepted and need to stay in the waiting queue until threads have been released from the current request processing. By limiting the number of concurrent threads, the server can avoid throughput degradation, and the overall performance is more robust than the unconstrained model. As it uses a pool of preliminarily created threads, this design can also avoid the cost of creating a thread per request arrival. However, setting the thread pool sizes to the number that yields the optimal performance in advance is usually not possible due to the dynamic characteristics of the workload, which may change over time. For this reason, server administrators are responsible for adjusting the thread pool size. A large number of threads may lead to performance degradation as it wastes resources. On the other hand, too few threads may restrict concurrency, since all threads may block while there is work the system could perform, resulting in lower performance.

Another disadvantage of the thread-based concurrency model is that scheduling and resource management decisions are taken by the operating system

at thread level, in a way which is transparent to applications. As a result, applications are rarely given the opportunity to participate in system-wide resource management decisions, or given indication of resource availability in order to adapt their behavior to changing conditions [170]. When each request is handled by a single thread, it is difficult for the operating system to identify internal performance bottlenecks in order to perform tuning and load conditioning.

2.3.2 Event-Driven Concurrency

Limitations that exist in the thread-based approach have led many developers to favor an event-driven programming approach, which has emerged as a solution for large loads and massive concurrent demands that arise in the increasingly popular Internet services [25, 28, 128, 183]. Event-driven programming is a generic term that is used to represent a programming architecture, which is based on detecting events and then responding to these events using a collection of cooperative tasks that are organized as event handlers.

In a server that follows this design approach, processing each of the concurrent requests in the system is implemented as a Finite State Machine (FSM). A state machine is a collection of states, input events and transitions that map states and input events to states. It is typically drawn as a directed graph where the nodes of this graph represent states, the directed arcs represent transitions and the arc labels represent input events – Figure 2.8 shows the finite machine of a simple HTTP server request. A state stands for a set of processing steps to be performed on the request. Events (messages, or whatever they might be called in a particular system) are a core concept in the event-driven concurrency, which are originating from the operating system or within the application it self, such as a disk I/O readiness, receiving a network message or a notification of the completion of some task execution. A special part of the application should deploy a mechanism to detect or to be notified of the occurrence of events. The detection could be in form of an approach where a small piece of software polls and waits for some thing to happen or it could be based on a notification from the event generator – see [96] for a comparison between available solutions. For each new request the server creates a new FSM and associates it with a descriptor. This FSM begins execution in some initial state, and each input event triggers the execution of a specific event handling

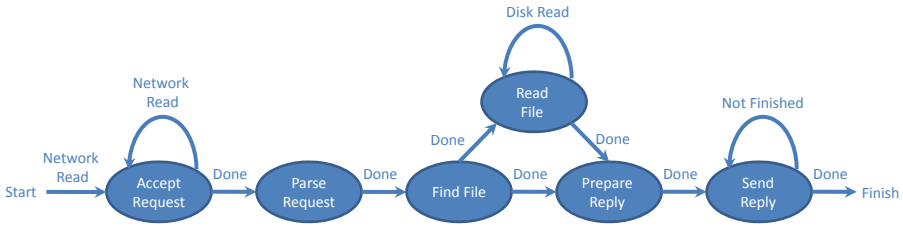


Figure 2.8: Finite state machine for a simple HTTP server request

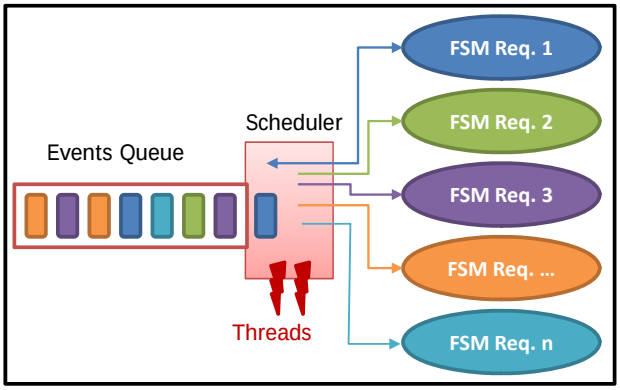


Figure 2.9: Event-Driven Concurrency

routine, and as a result triggers a transition from the current state to the next state [37].

In this form of concurrency, servers explicitly schedule their own work flows based on the detected events. A small number of execution threads is used, typically one or two per CPU in the system. These threads loop continuously dispatching events of different types from a shared queue, determining which FSM should be chosen to service each event and processing the corresponding event-handling routines. Then event handlers yield control by returning the control again to the event scheduler. In contrast to the thread-based approach, in this design just this small number of threads could attend a high number of requests simultaneously – see Figure 2.9. A thread is only needed to handle an event at a state and then move to handle the next event regardless to which request it belongs, instead of being assigned and responsible for the processing of the whole request.

Servers that depend on the event-driven concurrency model are typically more scalable in comparison to those of the traditional thread-based model. They tend to be robust to load variations, as their performance does not degrade with increased concurrency. In these servers, as the number of incoming requests grows, the server throughput increases until the bottleneck resources in the system become saturated. If the number of requests grows further, excess work is absorbed in the server's event queue. These servers show also a high flexibility and low operating system overhead, as the event-driven programming model can simplify concurrency issues. Because of the small number of used threads, this concurrency model can reduce opportunities for race conditions and deadlocks, and can avoid the overheads of context switching and synchronization among execution threads.

However, an important limitation of this approach is the assumption that event-handling threads do not block. For this reason non-blocking I/O mechanisms have to be employed, which are not well-supported by most environments and operating systems. Moreover, event-processing threads can block regardless of the I/O mechanisms used, due to many other reasons, like interrupts, page faults, or garbage collection, which are common sources of thread suspension that are generally unavoidable [28]. In addition, the primitives of event-driven programming raise a number of difficulties and challenges for application developers. Events scheduling and ordering is probably one of the most important concerns, since the application is responsible for deciding when to process each incoming event and in what order to process the FSMs for multiple work-flows. Modularity is also difficult to achieve, as the code implementing each state is directly linked with others in the flow of execution, and it must be trusted not to block or consume a large number of resources that can stall the available few threads. Another drawback in this concurrency model is that in this design applications generally cannot take advantage of multi-processor systems for performance, unless the needed modifications to support these systems are made and multiple event-processing threads are used [182].

2.4 Memory Accesses Problem

To meet the increasing demands presented by today systems, especially the highly concurrent Internet services, processors that power these systems have to deliver huge improvements in performance and throughput. As stated earlier, a limiting factor is the gap between processor performance and memory accesses performance. Although many hardware techniques attempt to alleviate the performance mismatch and bridge this gap, as the effects of this gap are expected to continue growing, an improvement in these techniques is needed to insure that increases in the performance of processoring resources result in corresponding increases in system performance. These techniques, such as caches, TLBs, and branch predictors [84], are based on a hierarchy of high speed cache memory and exploit the concept of locality, spatial and temporal reuse of code and data, which is a well-known property of computer programs. Depending on this locality, the cache hierarchy is used to predict the future behavior of the program and to keep data, which is likely to be reused quickly, close to the processing unit, in the fastest possible cache-memory level, in order to avoid memory stalls.

Today's systems employ larger and deeper cache-memory hierarchies, more sophisticated branch predictors, in addition to software/hardware prefetching mechanisms. However, the structure and the requirements of modern applications decrease the benefits of these techniques. Consequently, only a fraction of modern processors computational throughput is utilized. A significant part of CPU time is wasted because of the memory accesses behavior of the implemented designs and concurrency management approaches, and as a result of the high penalty of a cache miss that is often several tens of cycles in current machines, wasting CPU time during which many instructions could be executed.

Considering server applications, these applications are commonly organized to execute the code that is necessary to process multiple concurrent requests from multiple clients, using the event-driven approach (multiplexing a single execution thread among these requests), or using the thread-based approach (assigning a thread to execute each request). When a client request resumes execution, the code, all variables, along with the data structures that are frequently accessed consist the working set of this request. As a piece of data

from this working set is referenced for the first time, this data will be loaded into the cache of the processing unit on which the request is executed, which may take hundreds of CPU cycles. If it is referenced frequently enough to not be evicted, each subsequent reference to this data will find it in the cache, and will take only few cycles. Such data is called “hot in cache data”. However, the used server’s concurrency models disturb this behavior and fail to reuse cache contents. They cause frequent context switches from request to request in a random way, as switching to another thread in the thread based approach or handling another event from the event queue in the event-driven approach. This results in frequent control transfers between unrelated pieces of code and data, which happen during the execution of the different concurrent requests. Consequently, each request will tend to evict the working set, which was hot in cache for an other request. When a request is rescheduled again on the same processing unit and starts execution, it will need to reload its working-set’s evicted data at the cost of hundreds of cycles for each cache miss. These additional delays that a resumed request often suffers after each context switch, as caches have to be repopulated with the memory footprints of this request, are becoming significantly more expensive, especially when they are viewed in relation to the shortened dispatch time of each request, the deep cache-memory hierarchy of modern server machines and the higher costs of cache misses.

Since these context switching points tend to occur very frequently in highly concurrent systems, like Internet servers, this behavior disrupts the operation of the locality principle, on which cache hits depend, as it causes the loss of instruction and data accesses locality, and consequently decreases the effectiveness of the cache hierarchy. Considering this structure, cache techniques, which may improve the locality within each request, have limited effects on the locality across the multiple concurrent requests. These effects of memory accesses behavior and the poor cache utilization of server applications has been reported by many researchers in both software architectures, the thread based concurrency model [52, 77, 103] and the event-driven concurrency model [29, 30, 103].

On multi-processor systems, the memory accesses problem is even worse, as additional issues come into account. Problems like data sharing and thread migration emerge in such systems and disrupts locality and cache content

even further [156]. Servers, especially for commercial applications such as databases and Web servers, constitute today the largest segment of the market for multi-processor systems. In addition, the recent emergence of multi-core and multi-threaded processors presents a more wide spectrum of these systems with more complex structures.

In multi-processor systems, applications can execute well only when all processors in the system are effectively sharing the presented workload and when code and data structures that are accessed by an execution thread are located close to this thread. However, as resource management strategies in these systems usually promote load balancing in order to insure that all processors service an equal workload, they frequently migrate the execution of descheduled requests onto processors on which they may have not recently executed. Although this load balancing is commendable to increase utilization, migrations will often cause a performance degradation. During migration there are phases without normal execution as a newly dispatched request is unable to find its working set and data footprint in cache, since it is executed on a different processing unit that has a different cache at all [48, 51, 52]. The wait time that a thread of execution must incur to access the data of a processed request, coupled with the additional latency required to access memory across the system interconnect (not physically local to a CPU, but rather on another CPU board or memory bank) can impact the processor efficiency and the overall system throughput dramatically.

These effects are also compounded with data sharing and coherency protocols overheads [176]. When two or more execution threads are working on a set of shared writable data structures, these data structures have to be protected from uncoordinated accesses to prevent race conditions and other effects. In addition, as the execution threads that share these data structures may run in parallel in a multiprocessor system the integrity of the accessed data must be guaranteed as multiple copies of them may exist in different places in the cache-memory hierarchy. Coherency protocols are responsible for this data integrity, and as the number of processing units in the system increases the wait time which is incurred by these protocols increases. Although processor architectures which depend on an exclusive cache hierarchy rather than an inclusive cache hierarchy, like AMD processors, can reduce the effect of the

coherency protocol, they are affected by the cost of moving the shared data structures to the cache of the processing unit on which the request is executed. The overhead of this effect may increase even further in the case of false data sharing[5] which can cause moving the data structures between processing units even they are not really shared.

As future systems are expected to have more processing units and even deeper memory hierarchies, and as the effects of the processor-memory speed gap are expected to grow in these systems, more adaptive design and management solutions become necessary to best utilize available hardware resources and sustain high performance under massive concurrency.

2.5 Scheduling and Resource Management

Scheduling is a key concept in computing systems that refers to the way tasks are assigned to be executed by available resources in the system, and it is carried out by a special software known as scheduler. The set of rules that are used to determine which tasks would be processed, when to process them, and where to process these tasks – in the case of multiple processing units, is called scheduling policy. Typically, a scheduler is mainly concerned with: CPU utilization (keeping the processing units as busy as possible), throughput (the number of tasks that complete their execution per time unit), in addition to other application/service level performance metrics, like response time, fairness (the parties utilizing the resources receive fair service in terms of equal service time and the equal opportunity to be serviced.), etc. [34].

However, implementation issues and workload characteristics can have a large impact on these metrics and on the measured performance of a scheduling policy. Especially in Internet servers, scheduling of requests is a very critical factor that plays a major role in determining the performance of these servers. Multiple aspects have to be considered in these systems in order to fulfill the goals of a specific scheduler, as these goals often tend to conflict. Particularly, when the view of both clients and service providers have to be taken in account, the popular scheduling algorithms that are commonly used in Internet

[5] When threads that execute on separate processors repeatedly write to memory addresses on the same cache line.

servers include round robin, earliest deadline first, weighted fair sharing, etc., are not adequate to provide the requirements of these systems.

Considering the discussion in the previous section (Section 2.4), scheduling policies have to be implemented that avoid cache misses and context switches effect, in order to increase the system throughput and optimize the utilization of the available resources. In conventional scheduling policies, context switches occurrence typically relies on events that are generated in the system instead of the current state of the executed program. While this model is intuitive, as these systems depend on the two traditional concurrency models, it has several shortcomings and it can introduce high performance overheads [111]. As a result, these scheduling policies affect memory accesses locality and impact the system's performance dramatically, especially in the case of multi-processing systems.

Context switches effect is determined by the frequency of context switches as well as the number of cache misses that occur after each context switch. These amounts depend on the level of processor sharing, i.e. the number of executed tasks that share a single processing unit, how long these tasks are processed, and how fast they bring in new cache blocks to cause changes in the current cache content. Considering the massive concurrency in today's Internet servers, they are subjected to highly frequent context switches, with dramatic effects on their performance.

In order to reduce these effects, executed tasks should reuse their cached state more. One way to encourage that is to schedule each task based on its affinity to individual caches, that is, based on the amount of state that the task has previously accumulated in the caches of an individual processing unit. This technique is called cache affinity scheduling. Affinity scheduling explicitly routes tasks to processors with relevant data in their caches. However, the structure and the behavior of server applications reduce the benefit from this scheduling policy. The frequent switching between different requests, even if they are for the same service and execute the same program, interleaves unrelated memory accesses, thereby reducing locality. This gives rise to the problem of memory performance and improving locality from a single application's point of view, which needs cache-aware scheduling policies at a finer

granularity and is difficult to address using existing affinity scheduling policies.

Furthermore, the previous challenges are more intensive when the parallelism hierarchy of today's server machines is considered – see Section 2.2.3. In these systems, when a cache-aware scheduling policy is to be implemented, data sharing patterns among the executed tasks have to be considered too, in order to utilize the available resources optimally. If two tasks are processed on two processing units that reside on the same CPU core (i.e., hardware execution threads), communication typically occurs through the core private cache, with a latency less than 10 cycles. If the two processing units do not reside on the same CPU core but reside on the same chip, communication typically occurs through a shared cache level, with a latency of 30 to 50 cycles. If the processing units reside on separate chips communication will be across chip with an average latency of hundreds of cycles. As an example, consider the Intel Xeon (Nehalem EX) latencies [66]. According to that, tasks that communicate frequently will be better executed on processing units that communicate through a cache-memory level with less latency. At the same time, non-communicating tasks with high memory requirements may be better placed onto different chips, helping to reduce potential cache capacity problems. Although today schedulers have become increasingly cache-aware, they do not take these non-uniform sharing overheads into account.

In addition, as cache-aware scheduling policies traditionally utilize a per processing unit run queue, a conflict between load balancing and locality is evident. Locality conscious scheduling would opt for tasks to spend their entire execution lifetime on the same processin unit. Fixing tasks onto processing units, while allowing tasks to reuse more cache state and to benefit from locality, it could cause load imbalance, as it may neglect that some processing unit might be lying idle. Conversely, a scheduling algorithm that complies with the load balancing policy would attempt to migrate tasks to balance work load evenly, which may disregard the task's last dispatch locale. For this reason, ideally, scheduling algorithms should aspire to include both policies.

To be noted here are the differences in static approaches, which make use of long-term trends and off-line optimizations, versus dynamic approaches which

exploit current changes in the system in order to allocate resources and provide load balancing. System dynamics and the rapid development make static approaches, which are typically based on worst-case assumptions inadequate and increasingly conservative. For this reason, research is needed on design and implementation techniques that allow dynamic run-time flexibility with respect to variations such as changes in workload and resource utilization patterns.

2.6 Performance Management

The Web was traditionally considered as a non-critical application for which no performance guarantees are required. However, with the fast growing number of Internet users and the increasing pace of information being exchanged, Internet service providers are increasingly expected to provide the information with an ever faster speed. A measure of this trend is the "Latency Tolerance", which is a measure of how long a user will wait for a web page to load before taking some action (e.g. hit reload or abandon the web site). For many years, there was a widely used rule of thumb which says that most users will wait approximately 8 seconds for a web page to load and after that time they will start thinking about taking some other action [142]. In 2006, the threshold for web page loading decreased to 4 seconds, and according to a recent research, Internet users expect a web page to load in two seconds only [56].

Today, the need for highly reliable and highly available Internet services increases, as businesses, government agencies, and individuals are increasingly dependent on the Internet resources for their day-to-day operation in order to satisfy business and personal needs. In addition, the Internet is evolving rapidly into a provider of new categories of modern, globally accessible, digital services that require performance guarantees, such as on-line banking, e-trading, business transactions, real-time databases, on-line games, distance learning, etc.. As the popularity of these new services grows extremely, there is an increasing demand for Internet servers to meet customer's expectations in terms of the quality of service delivered, in order to increase clients satisfaction and as a result achieve more profits. Therefore, service providers increasingly need to control and guarantee the performance of their services to provide a variety of performance characteristics, such as response time, throughputs,

availability, costs, resource utilization, etc., which can be analyzed from different viewpoints. For instance, a user's perception of performance has to do with fast, predictable user-perceived response time, no connections refused and 24 $\times$7 (24 hours/day, and 7 days/week) up-time. On the other hand, from management's viewpoint performance also includes high throughput (the rate at which the system can perform work) and high availability with low operation costs.

In such services, what the customer sees is the level of service provided by the company, and any degradation in this service level can be noted in real time. Quality of service (QoS) indicators are usually used to define the quality of service provided by sites. These indicators represent the level of service provided to customers, may be at a given cost. QoS of a service provider plays a crucial role in attracting and retaining customers and in determining their satisfaction. Failure to meet performance specifications or desired performance levels may result in loss of customers, financial damages, or liability violations. For instance, frustrated customers leave e-commerce sites and do not return, causing revenue to be lost. Unacceptable performance and availability can cause serious harm to a company's bottom line and market value in the extremely competitive world of the Internet.

However, as these services execute in an unpredictable general-purpose environment, which is the open Internet, that brings many big challenges. This environment has many unique characteristics, which have a profound impact on service performance. As stated earlier, the web community is growing day after day, increasing exponentially the load that sites must support. In addition, current sites are subject to enormous variations in demand, often in an unpredictable fashion, including flash crowds that cannot be processed. Accordingly, servers that are attending requests from thousands of clients, simultaneously, need to perform really efficiently if they want to offer a good Quality of Service. The more requests a web site gets the higher the probability that users will wait too long for a response, and in many cases web users or customers will become frustrated and switch to another site. For this reason, Internet servers must deal with the Internet load characteristics to achieve target QoS and performance guarantees in the face of unpredictable server overload situations (i.e. when the volume of requests for content at a site tem-

porarily exceeds the capacity for serving them and renders the site unusable).

Traditional approaches for designing performance guaranteed computing systems depend on quantifying hardware resources and software execution requirements, then apply an appropriate combination of pre-run-time analysis, capacity planning and admission control to ensure that the system is not overloaded and that the desired performance is achieved. Capacity planning is the process of predicting when future load levels will saturate the system and determining the most cost-effective way of dealing with system saturation. However, traditional capacity planning based approaches are not adequate, because of the exponentially growing demands that the services must support resulting from their increasing popularity, the huge variations in these demands and the dynamic nature of these services. These approaches rely on a prior workload and resource knowledge and are therefore inapplicable as both the processing costs and requests arrival rate are highly unpredicted. Leaving the management of dynamics to operators is also not acceptable because many changes occur too rapidly for human to be able to response in a timely manner. For example, e-commerce sites frequently contend with workloads that change so rapidly that service degradations and failures result. Instead, the predominant practice for providing QoS assurances today is over-provisioning based designs. A drawback of this practice is that it results in costly systems with uncertain assurances regarding performance. For these reasons, automatic self-management techniques are needed that cope with the dynamics of these computing systems and adapt the system in a more reliable manner.

3 Problem Addressing

Considering the challenges presented in the previous Chapter, Internet services must not only support high concurrency, but also provide quality guaranteed service, in a dynamic, unpredictable environment. This thesis is based on the advantages of a recently rising programming paradigm, that is called the staged design, and improves over this approach to provide services which are to be qualified to address these challenges. This Chapter explains the staged design in details and presents the challenges and limitations in the staged design that we address throughout this thesis.

3.1 Staged Design & SEDA

3.1.1 Staged Design

Modern applications are getting larger and more complex, which is expected to continue in the future, leading to complicated implementations. As a result, applications are increasingly difficult to extend, tune or evolve, especially those applications which have additional requirements related to scalability and performance issues, like Internet services and server applications. To cope with this increasing complexity, a recent trend is to identify the independent units of work that construct an application, and to build this application explicitly from sub-tasks or stages that represent these units. This trend gives the rise to a shift in the view from traditional event- and thread-centric programming models to task- or stage-centric programming models [24, 42, 103, 138, 173].

Staging applications explicitly appeared in the seventies as a solution to the limited size of main memory [151], and recently this design emerges again as a programming model to provide solutions at hardware and software engineering levels. Hence, a broad wide spectrum of applications have been implemented using this design paradigm, like Operating systems [102], data-base management systems [77], web servers [70, 173], network applications [101, 121, 160],

etc..

Staged design introduces a hybrid programming abstraction that combines ideas from the two traditional programming paradigms, event-driven programing and thread-based programing, and presents stages as the constructing blocks underlying applications. This design offers the best of both worlds, the ease of use and expressiveness of threads and the flexibility and performance of events. In this design model the code of a complex program is broken down into multiple independent self-contained stages, by introducing queues as stage's boundaries. The program is constructed as a graph or a network of these stages and the stages are connected to each other through the queues. A stage in this model embodies a robust, reusable software component that consists of a local state, private exclusively owned (to the degree possible) executable code and data environment, and a group of exported operations that perform some aspect of the service provided by the program. These operations are invoked in an asynchronous manner by queuing a message (request, event, etc.) at the queue of the stage, so that their invocation, execution and reply, when necessary, are decoupled. Every stage is capable of receiving messages, performing work on these messages, sending the same or newly created messages to other stages and controlling its local performance. When the tasks of a message processing at a stage are finished, a new message is generated and passed to the next stage, so that communications between stages can be considered as a matter of data switching between threads, each executes within a different stage. Each stage is serviced by one or more threads, which can be part of per stage or shared thread pools.

In comparison to the object oriented programming, we can speak about a stage oriented programming model [103], to the extent that it is an abstraction to structure programs that provides local state and operations for a stage. However, stages still differ from objects in three major aspects. First, operations within a stage are invoked asynchronously –by enqueuing a message (event) at the stage's queue, so that a caller does not wait for the computation to complete at the called stage. Second, a stage has autonomy to control the execution of its operations, and this autonomy extends to deciding when and how to execute the computations associated with the invoked operations. Finally, stages are a control abstraction used to organize and process work, while

objects are a data representation acted on by other entities, such as functions, threads, or stages.

Staged design-based computation can support a variety of programming styles, including software pipelining, event driven state machines, bidirectional pipelines, and fork-join parallelism. For example, in this design a server application can be arranged as a pipeline in which requests arrive at one end and responses flow out from the other. This form of computation is easily supported by representing a request as an object passed between stages. Linear pipe-lining of this sort is simple and efficient, because a stage retains no information on completed computations.

3.1.2 Staged Event-Driven Architecture

The Staged Event-Driven Architecture (SEDA) is the state of the art of the staged design, which has emerged as a design framework deployed for supporting the massive concurrency demands of large-scale Internet services, as well as to exhibit good behavior under heavy load [173, 174]. In SEDA, a complex service logic is broken down into a set of basic event-driven tasks. The tasks are separated by event queues and a request is processed by a sequence of these tasks. Each task is implemented within a stage and the whole service is a network of these stages communicating using the event queues. A stage, which is the fundamental unit in SEDA (Figure 3.1), consists of an event handler which implements the basic functionality and performs some aspect of request processing, an incoming event queue and a pool of threads to process events entering the stage. Threads within a stage operate by pulling a number of events off the event queue and invoking the event handler which processes the events and dispatches the results by putting them as events in the event queues of other stages in the system. Figure 3.2, which is taken from [174], shows the structure of a simple HTTP server that is based on the staged event-driven architecture.

SEDA allows stages to have private thread pools or to share thread pools among stages. These threads are used to handle blocking I/O operations and to utilize CPUs in multi-processor systems. To avoid over-using threads, it is important that blocking operations be short or infrequent. For this reason, SEDA provides non-blocking I/O primitives to eliminate the most common sources

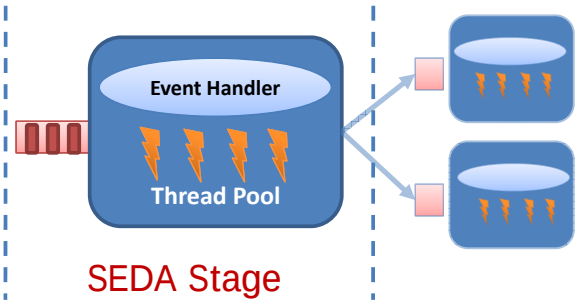


Figure 3.1: SEDA Stage

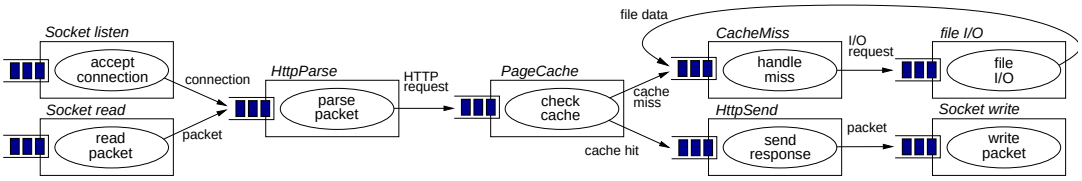


Figure 3.2: SEDA-Based HTTP Server

of long blocking operations [169]. The explicit event queues between stages prevent demands from overcommiting the resources when they exceed service capacity. In addition, they separate the stages and act as execution boundaries between stages and as a mechanism for controlling the flow of requests in the whole system. Therefore, Stages are isolated from each other and each stage is responsible only for processing the subset of requests existing in its queue to avoid holding resources by single request/thread for too long time. Moreover, a stage has scheduling autonomy, which enables it to control the order and concurrency with which its operations execute. Each stage can be independently managed and stages can be run in sequence or in parallel, or a combination of the two. As a result, this design avoids resource contentions, scalability limits and other overheads that are associated with thread-based concurrency. It avoids also the complexity of the standard event-driven programming model, and allows applications to be well-conditioned to load by making informed decisions based on the inspection of pending requests in the event queues. In addition, SEDA promotes stage autonomy, data and instruction locality, and minimizes the usage of global variables.

In a SEDA-based system dynamic resource management, adaptive overload control and other self-tune control techniques are implemented through con-

trollers that are in each stage. By monitoring the queue length and the service rate, the resource controller employs a heuristic control approach to adjust the number of threads in the thread pool of the stage. The overload controller makes use of admission control at the source of the event queue to control the number of requests accepted by the system. This approach have been also extended to perform class-based service differentiation [172, 174].

3.1.3 Advantages of the Staged Design

While the staged design model is conceptually simple, it yields a number of benefits which directly address the needs of today Internet services. This design has a number of desirable properties for high-concurrency management and it is one of the best means for structuring applications that require accurate control. In addition, it provides many solutions at hardware level and at software engineering level. This Section presents a survey of the benefits of this design, which have been reported by many researches [77, 108, 141, 170, 174].

At the hardware level the staged design provides many solutions to challenges that appear in the traditional concurrency models. These solutions can be outlined as follows:

- Cohort Scheduling: The staged design can optimally exploits the underlying memory hierarchy as it is particularly amenable to cohort scheduling [103]. Cohort scheduling means to group the execution of multiple requests that have potential to show the same behavior and locality, in order to benefit from caches content reuse (see Section 3.2.2). In this design the scheduler can repeatedly execute requests queued up in the same stage, thereby exploiting stage affinity to the processor caches. The first request's execution fetches the common data structures and code into the higher levels of the memory hierarchy while subsequent request executions experience fewer cache misses. This type of scheduling cannot easily apply to existing systems that are based on the traditional programming models, since it would require annotating the execution threads with detailed application behavior.

- Context Switches: Context-switches that occur in the middle of an operation evict the working set of this operation from the higher levels of the cache hierarchy. Each context switch requires that the processor save all

its registers for the previous operation that it was executing and load its registers for the next operation that it runs. As a result, each resumed operation often suffers additional delays while re-populating the caches with its evicted working set and restoring the processor state. The staged design can reduce the overheads of the frequent context switches that are existed in the traditional thread-based model through the following mechanism: A stage contains code with one or more logical operations. Instead of preempting the currently executed operation at a random point of the code (for example, whenever a time quantum elapses), a stage thread that executes this operation voluntarily yields the CPU at the end of the stage code execution, as a request's processing at each stage is typically very short and runs to completion. When an operation runs to completion, it does not require its own stack or an area to preserve processor state. This way the working set, that is evicted from the cache, would be at its shrinking phase and the time to restore it by the next execution (maybe at another stage) is greatly reduced, which eliminates much of the cost of context switches.

- Parallel Processing: The staged design can also take advantage of parallel processing systems, as it uses multiple execution threads and as it is capable of exploiting irregular parallelism in the presence of complicated control structures [24]. As stated in the previous section, stages are isolated from each other by event queues, for this reason each stage can be executed on a separate processing unit, "almost" independently of other stages execution. Moreover, as it innovates from the event-driven programming model and depends on asynchronous operations, the staged design provides low-cost parallelism and can simplify concurrency issues by reducing opportunities for race conditions and deadlocks. In addition, the properties of this design enable the implementation of a staged design-based application as a distributed system, which also enables the benefit from the advantages of such systems, like the additional capacity and the fault-tolerance properties [26].

At the software engineering level, it have been shown too that applications which are organized as a network of stages connected by event queues offer a variety of advantages over traditional architectures. The key advantages and the solutions to software-complexity problems that are presented by the staged design can be outlined as:

- Code modularity, flexibility and extensibility: The staged design innovates from the traditional event-driven design patterns. However, in contrast to the "monolithic" traditional event-driven design, in which the states of the request-processing state machine are often highly inter-dependent, the staged design allows stages to be developed and maintained independently. An application that is based on this design model consists of a network of inter-connected stages; each stage can be implemented as a separate code module in isolation from other stages; and the operation of two stages is composed by inserting a queue between them, thereby allowing events to pass from one to the other. As these stages provide a well-defined functionalities, this design makes it easy to replace a stage with a new one (e.g. a faster algorithm), or develop and plug stages with new functionality. The programmer needs only to know the stage functions and the limited list of global variables that are accessed within this stage. Accordingly, this design will significantly enhance the long-term value of implemented applications, and will allow implementers to follow an evolution-upgrade approach, rather than a big-bang approach [141].

- Easy to tune: As stage autonomy eliminates inter-dependencies with other stages, each stage can provide its own monitoring and self-tuning mechanism. Each stage is responsible for adjusting all stage related parameters, such as the number of threads, scheduling policies, buffer space, slice of CPU time, etc, which make it easier to build auto-tuning tools. In addition, the utilization of both the system's hardware resources and software components at a stage granularity can be exploited during the self-tuning process.

- Testing and debugging support: Few tools exist for understanding and debugging a complex event-driven system, as stack traces do not represent the control flow for the processing of a particular request. The staged design facilitates debugging and performance analysis, as the decomposition of application code into stages and explicit event delivery mechanisms provide a means for direct inspection of application behavior. For example, a debugging tool can trace the flow of events through the system and visualize the interactions between stages. In addition, using the staged design, independent teams can test and correct the code of a single stage without looking at the rest of the code.

- High concurrency and scalability: As with the traditional event-driven design, the staged design is based on asynchronous facilities to perform I/O operations and communications between stages [169, 173]. Consequently, that eliminates the need for multiple threads to overlap pending I/O requests. Staged design-based applications make use of a small number of threads to process requests within each stage, while the requests that are waiting to be executed are pending in the queues between stages. This addresses scalability issues in a very comprehensive manner, and avoids the performance overhead of using a large number of threads for managing concurrency. In the Staged Event-Driven Architecture (SEDA) which is the state of art of the staged design, the number of threads can be chosen at a per stage level; this approach avoids wasting threads on stages which do not need them. Each stage is allocated a number of worker threads based on its functionality and the I/O frequency, not on the number of concurrent clients. This way there is a well-targeted thread assignment to the various execution tasks at a much finer granularity than just choosing a thread pool size for the whole system.

- Application-specific load conditioning: The use of explicit event queues allows applications to implement load conditioning policies based on the observation of pending events. A stage can drop, filter, or reorder incoming events in its queue in order to implement some policy, such as event prioritization, for example. During overload, a stage may prioritize requests requiring few resources over those which involve expensive computation, or a per queue fine-grained rejection can be implemented by having a queue reject new entries when it becomes full. Alternately, an application-specific form of service degradation can be implemented, by routing events to alternative, faster stages (e.g. trading off accuracy for speed) and thus momentarily increase server capacity. These policies can be tailored to the specific application, rather than imposed by the system in a generic way.

3.2 Challenges in the Staged Design

Staged design has been presented as a programming paradigm to implement high performance Internet services and to support highly concurrent demands. This design takes benefit from both traditional concurrency approaches – thread

based concurrency and event-driven concurrency. However, although this design avoids the pitfalls related to conventional concurrency models, it introduces many design challenges. This Section presents challenges and limitations in the staged design which are addressed throughout the thesis.

3.2.1 Resource Allocation in Staged Internet Services

An application that is based on the staged design consists of a network of stages, so that a client request is processed along a pipeline of these stages. Considering this structure, the overall performance of the system is limited by the performance of the bottleneck stage in this pipeline. That means, in order to increase the system performance we have to increase the performance of this bottleneck stage, e.g. raising the stage's service rate, which could be done by allocating more resources (for example, more CPU time) to this stage. However, the increase in performance of one stage may cause an increase in requests accumulation at the queues of other stages, which in turn would result in request drops at these queues, e.g. because of buffer space or timeouts. This domino effect disrupts the normal request transition flow into other queues in the staged service, and eventually leads to an even lower system performance.

In addition, since all stages are competing for the same resources, giving more resources to one stage will decrease the resources allocated to other stages, which may result in the creation of a new bottleneck stage. Moreover, the complexity of this problem is increased even further due to the dynamic nature of both system load and resource requirements of the individual stages.

To solve this problem, we need to allocate the available resources in the system, so that we balance and coordinate the performance of the different stages in the staged service. This resource allocation has to take in account the different demands of the individual stages which are changing during execution in order to avoid wasting resources and to achieve the optimal system performance.

3.2.2 Cohort Scheduling

Cohort scheduling is a technique for organizing the computations in server applications to improve program locality [103]. The key insight is that distinct requests on a server execute similar computations. The server can defer

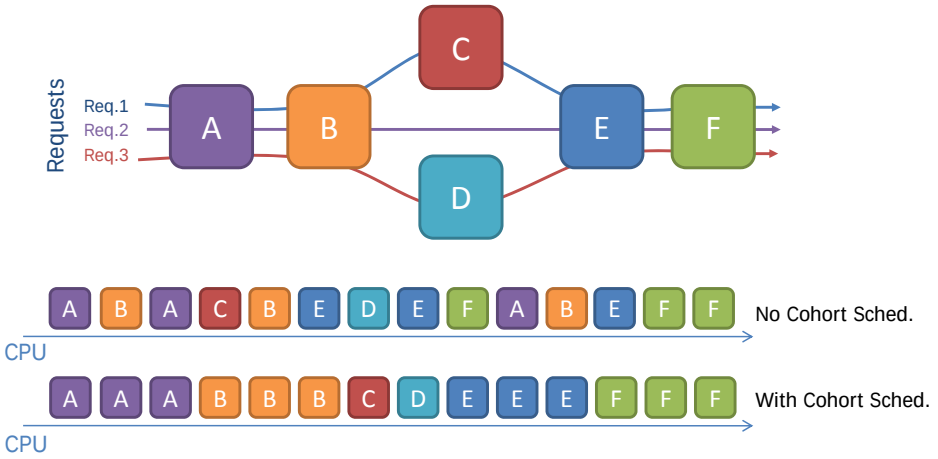


Figure 3.3: Cohort Scheduling.

the processing of a request until a cohort of computations arrive at a similar point in their processing and then execute the whole cohort consecutively on a processor –See Figure 3.3. This policy increases opportunities for code and data reuse, by reducing the interleaving of unrelated computations which may cause cache conflicts and may evict live cache lines.

The staged design was originally introduced as a programming abstraction to implement these cohort scheduling policies in order to improve memory access behavior of high concurrent servers. In the staged design, a stage facilitates cohort scheduling because it provides a natural abstraction for grouping operations with similar behavior and locality and the control autonomy to implement cohort scheduling. Operations in a stage typically access local data, so that effective cohort scheduling only requires a simple scheduler that accumulates the execution of pending operations at a stage to form a cohort.

Therefore, existing cohort scheduling policies for staged applications aim to increase the benefit from locality and optimize the performance of the individual stages rather than optimizing the overall system performance. As a result, these policies introduce coarse grained resource allocation approaches which increase the difficulties related to controlling the system performance.

Since this cohort scheduling is commendable, a resource allocation scheme is needed which provides such policies and at the same time takes the require-

ments of the different stages and the whole system in account.

3.2.3 Parallelism Hierarchy

As stated in Section 2.2, today's Internet services need increasingly more processing resources. For this reason, parallel processing techniques are employed to cope with these increasing demands. In addition, the recent trend of chip multi-processing and multi-core systems presents a set of new characteristics in today's server machines. These characteristics, especially the hierarchic parallelism in these systems, have to be considered when the system resources are to be allocated to individual tasks.

Typically, in a multi-processor system, processing multiple requests in parallel on different processing units may cause additional CPU time overhead related to the contention of the concurrently processed requests for shared resources (shared data structure, locks, bus, shared caches etc.). In addition, as multiple copies of the shared data structures are existed in different places of the memory hierarchy, coherency protocols have to be used in order to insure the integrity of these data structures, which increase the time overhead.

In the staged design, this processing time overheads are affected by the type of requests that are processed in parallel. If the requests that are processed in parallel are at the same stage, they will share more data structures than requests that are processed at different stages. For this reason the overhead of the contention for the shared data structures and the overhead of the coherency protocol will be larger. On the other hand, if the requests that are processed in parallel are at different stages, each request will need to load the code and data structures of the stage at which it is executing into the caches of the processing unit. As a result the concurrent requests will content for the shared cache level, evicting each other working set from the shared cache, which causes additional processing time per request.

Another aspect that is to be considered in such parallelism hierarchy is that when requests are processed on different cores at the same chip they share data that exist in the caches of this chip. While, in the case of processing requests on other chips, requests will need to access data structures that reside on other chips, which is significantly more expensive.

Considering the previous discussion, as we allocate processing resources to the individual stages in the staged design, we need to make a trade-off between processing multiple requests from the same stage and processing multiple requests from different stages in parallel, in order to benefit from locality and at the same time not to be over-committed by parallel processing overheads. Processing requests from the same stage in parallel will increase locality as requests reuse data that exist in the cache, but it will increase the overhead of locking and coherency protocols, too. On the other hand, running requests from different stages in parallel will decrease contention for shared data structures and decrease the effect of coherency, as less data is shared, but this will cause more contention for shared cache-memory levels and memory bandwidth.

3.2.4 Performance Control for Staged Services

There is a growing demand for Internet servers to provide quality-guaranteed service for highly concurrent requests. To be qualified to meet this requirement, a system must not only support high concurrency, but also the performance of this system has to be controlled, see Section 2.6 .

As demonstrated by many researches, the staged design has a simple structure, can greatly benefit the system in massive concurrent loads and is potential to handle dynamics in these loads [108, 110, 172, 173]. However, this design presents many challenges when the system is to be tuned to achieve a desired performance target or to guarantee a determined service level. For many reasons, quality guarantees for requests processing in a staged design-based system are difficult to maintain. First, requests are continually generated in large volumes by external and internal sources such as clients and I/O operations. Second, the processing costs of requests vary dynamically along the time and are difficult to predict. The last, fluctuations of loads and resources usage of the individual stages may cause overloading that interferes with the robustness of the system or may under-utilize the available system resources. To deal with these challenges and such instability, systems that are based on this staged design model need to automatically adjust the whole system at run time in order to achieve target performance levels.

Existing approaches are based on managing the resources depending on an off-line knowledge of the system characteristics to achieve high performance. Configuring such systems to generate the desired performance levels requires experienced administrators to correctly set multiple relevant control parameters for multiple stages or to determine these parameters experimentally using benchmarks. This management technique is a tedious manual operation which is difficult, time-consuming, error-prone and non-QoS-guaranteed, especially when it is to be implemented in such highly dynamic computing systems, like Internet services. In addition, this configuration is not based on any mathematical relationships between the controlled parameters and the target performance. As an optimal configuration usually depends on an administrator's good guess, therefore, parameter configuration can easily result in over utilizing or under utilizing the available resources.

4 Related Work

Before representing the proposed approach, in this Chapter, we review related work on the Staged design and other similar programming models. In addition, we present a review of Cache Conscious Scheduling, performance control mechanisms for Internet servers and related researches that have been introduced for staged design-based services.

4.1 Staged Design and Similar Design Approaches

Many design approaches have appeared recently in order to cope with the increasing complexity of today programs and with the different requirements of the applications that are based on these programs. This section presents a review of those design approaches which are similar to the staged design SEDA on which this thesis is based.

The staged event-driven architecture (SEDA) is the state of the art of the staged-design. In this architecture the service functionality is broken down into multiple tasks; each task is implemented within an event-driven stage and is executed using a pool of execution threads. In [173], Welsh et al. have proposed this staged event-driven architecture, which has also received active research in the recent few years. For example, in [129], Pariag et al. have presented a performance oriented comparison of event-driven, thread-based and staged design-based servers. The comparison has shown a competitive performance of the staged design-based server to the performance of other architectures obtained using the best tuning. Venkatesan et al. [100] proposed usage of SEDA based event driven service oriented architecture for tackling the flexibility, and scalability requirements of today's network management systems. Bharti et al. [141] have suggested a refinement of SEDA by breaking each stage into sub-stages in order to account for stages with high load in the system. The authors of this work argue that this refinement of SEDA architecture, termed as Fine Grained SEDA, can improve certain performance metrics

like scalability of stages, load conditioning etc.

As mentioned in the previous chapter, a recent system design trend is to identify the independent units of work that construct an application and to build this application explicitly from sub-tasks or stages that represent these units. In addition, the different limitations of the traditional design approaches, give the rise to hybrid design approaches. Hybrid approaches combine ideas from both the thread-based concurrency and the event-driven concurrency, in order to benefit from the advantages, avoid the pitfalls and get the best of both worlds. The Staged Event-Driven Architecture (SEDA) belongs to both categories. For this reason, we begin with an overview of approaches that depend on dividing applications into multiple work units, and then we present studies that have introduced hybrid design structures.

Recently, many major system design tools and programming languages have presented tasks as a new design abstraction. For example, the recent OpenMP version 3.0 shifts the focus in system design from threads to tasks [24]. Microsoft has developed Task Parallel Library (TPL) [105, 117], and Intel has presented Threading Building Blocks (TBB) [88, 138], which is a C++ run-time library that allows the user to program in terms of tasks too. A task is an action that can be executed concurrently with other tasks. The intention behind these new design techniques is to simplify the process of adding parallelism and concurrency to applications, and to hide threads (creation, synchronization, termination...) and hardware features by focusing on tasks. The run-time library takes full responsibility for scheduling the tasks for locality and load balancing. Tasks are similar to stages in that they present the application as a graph of independent work units.

In [40], the authors have presented a stage-wise request queuing architecture for web servers, where the stages represent sub-tasks within sessions. Harizopoulos and Ailamaki have proposed to break database systems into modules and to encapsulate them into stages that are connected to each other through queues, in order to remedy the weaknesses of database management systems [77]. This approach is very similar to SEDA, but it focuses more on locality. Qie et al. have proposed dividing server programs into services, where a service is a program component that provide an independent functionality, to im-

prove the robustness of these programs against Denial of Service (DoS) attacks [135]. In [44], a Pipe-lined web server architecture have been proposed that is suitable for Symmetric Multi-Processor (SMP) and System-on-Chip (SoC) architectures and is similar to the staged design. In [101], a framework has been presented for designing event-driven staged Internet applications, which is similar to SEDA, and in [70], Gordon has proposed a framework and a C++ infrastructure for developing Staged design-based servers.

Considering hybrid design models, the programmer would design parts of the application using threads, where threads are the appropriate abstraction, and other parts using events, where they are more suitable. All the previous design architectures can be considered as hybrid approaches as they depend on the occurrence of some event during the execution to take some response or to do some action. In addition, many existing systems implement the hybrid model to various degrees, but most of them have a bias either toward threads or toward events. For example, Flash is an event-driven web server which exploits the creation of a set of helper processes (threads) to avoid blocking operations in the main servicing process [128]. Capriccio is a user-level, cooperative thread library with a thread scheduler that looks very much like an event-driven application [168]. In [107], a hybrid approach have been proposed that is based on tracing the sequence of system calls to simplify the development of massively concurrent services. In [8], the authors have also presented a hybrid design approach that combines the advantages of both programming styles (threads and events). Zeldovich et al. have presented an approach that runs N copies of the web server as a means of enabling an event-driven web server to leverage multiple CPUs in a multi-processor environment [182].

4.2 Resource Allocation and Cache Conscious Scheduling

The expense of repopulating a processor's cache with a newly dispatched task's footprint, the impact of context switches and the impact of cache misses on the performance have been early reported and studied in multi-programmed systems [106, 111, 161, 162, 166], in multi-threaded applications [30, 51, 52, 77, 103] and recently the case of chip-multi-processing (CMP) and multi-cores systems have been considered [48, 62, 63, 95, 156].

Many studies have reported that context switching introduces high overheads directly and indirectly [61, 106, 162]. Direct overheads include saving and restoring processor registers, flushing the processor pipeline, and executing the scheduler. Indirect context switch overheads include the perturbation of the caches.

In [98], the authors have characterized context switch misses of an application for various cache parameters and investigated how to reduce them. Li et al. [106] have measured both direct and indirect context switching overheads through simulation and concluded that indirect context switch overheads due to the cache perturbation effect are much more significant than direct overheads. They have also shown that the working set and data access patterns of an application could significantly affect the context switch overheads. In [111], the authors have also reported about "reordered" cache misses and the effect of prefetching on context switches overheads. In addition, they have introduced an analytical model to characterize how cache parameters and application behavior influences the number of context switch misses the application suffers from.

Vaswani et al. [166] introduced a quantitative study of the effect of processor reallocation on the performance of various parallel applications multiprogrammed on a shared memory multi-processor system and concluded that the benefits of processor reallocation (increased utilization), outweigh the costs imposed by such reallocation. Torrellas et al. [161] showed that affinity-aware scheduling, in a similar system, reduces the number of cache misses resulting in performance improvements of up to 10% (for the workloads they studied). Although these early works accounted for small benefits of exploiting cache affinity, as processor-memory speed gap continues to grow and when viewed in relation to the shortened thread dispatch time in modern multi-threaded applications, especially, in the highly concurrent server applications, these benefits become more significant.

Motivated by that, many approaches have been introduced to improve the memory accesses behavior of these applications in order to reduce cache misses, by grouping the execution of operations or tasks that have a similar locality. Debattista et al. [51, 52], introduced "batching" as a technique for reducing

the negative impact of the poor cache utilization in multi-threaded applications. This technique was implemented by grouping threads that are expected to share the same data structures and scheduling them on the same processor. Bhatia et. al [29, 30] introduced an approach based on grouping the execution of similar tasks in an event-driven server, and could increase the throughput of the server by up to 40%.

Recently, as chip multi-processing architectures (CMP) are emerging as the dominant architectures for wide range of platforms, issues like thread migration, load balancing and managing shared caches have been studied in a finer granularity levels.

Fedorova et al.[63] determined that the shared L2 cache[1] is a critical resource on CMT (chip-multi-threading) and they introduced a L2-Cache conscious balance-set scheduling approach [54], that separate all runnable threads into groups such that the combined working set of each group fits in the cache. Their results showed an improvement in processor throughput by 27-45%.

Thread Clustering [156] described the design of a scheme to group threads in "clusters" based on detecting sharing patterns on-line using monitoring, in order to reduce expensive cross-chip cache-accesses. In [38], the authors have used hardware performance counters to build set of processes to run at the same time-slice on a Symmetric Multi-Threading processor in order to improve performance

The staged design [77, 103, 173] takes another way, and presents stages as the application building blocks. Grouping requests processing at each stage can reduce cache misses and improve the performance as long as the working set of the stage fit into the cache. Larus et al. [103] proposed the staged design as an approach to implement such scheduling techniques, called cohort scheduling, to exploit the data sharing nature of multiple requests in a workload. This approach uses a simple wave-front algorithm to supply processors to stages. At each stage, a processor executes all the requests that are pending

[1] In [63] L2 was the last level cache. Recent multi-core processors have a dedicated L1 and L2 caches to each core and a shared L3 cache and future systems are expected to have higher cache hierarchies. However, we expect that the shared cache level will continue to be a critical shared resource in these systems too.

in its queue. After the requests are finished, the processor proceeds to the next stage. Harizopoulos and Ailamaki used the staged design to implement cohort scheduling policies in a database management system [76, 77]. This approach uses an alternative to the requests scheduling policy presented in [103] by enforcing a threshold on the number of requests processed at each stage before moving to the next stage. However, these request scheduling policies are inflexible and not suited to fine-grain resource allocation. In addition, in these approaches issues related to multi-processing were not studied. Original SEDA [173] and Li et al. [109] have introduced two approaches which balance the CPU share of the individual stages by adapting the stages thread pool size based on load and performance, however both approaches did not regard the cohort scheduling as a main issue. Kokku et al. [99] introduced an adaptive processor allocation in pipe-lined packet processing systems, although the system is similar to the staged design however they depend on manual setting of control parameters and on assumptions that are feasible in packet processing systems but not in generic Internet applications.

The contribution of this thesis is to introduce a resource allocation policy to allocate processing resources to the individual stages, which uses cohort scheduling in order to benefit from stage's locality in addition to balancing the allocation to the individual stages. Our scheduling policy is comparable to the Weighted Round Robin, an alternative of Fair Queuing [53], which is a technique used in network devices to have a fair share of network resources between data flows. Rather than allocating resources fairly between concurrent data flows, our approach introduces a technique to allocate resources to cooperative tasks to avoid the negative effects of a bottleneck task on the overall system performance. The authors of [149] introduced a similar approach to allocate portions of CPU time periods to threads in applications such as web servers depending on their estimated progress. However, since the staged architecture introduces more explicit mechanism to estimate stages progress, we believe that such an approach is more sophisticated to be implemented in staged architectures.

4.3 Performance Management

As the need for performance guarantees increases in today Internet services, performance management in these services has become a research thrust area in the recent years. Many approaches have been introduced to cope with performance management requirements in such services, like overload control, quality of service guarantee, service differentiation, etc..

In the highly concurrent services of the open Internet, when the system is overloaded maintaining the desired service quality is a difficult problem because both the processing costs and requests arrival rate are highly unpredictable. In overload situations the service quality is strongly related to the number of requests that arrive at the system and those which are waiting to be served [3]. By controlling these numbers many performance metrics can be controlled, e.g. the response time of each accepted request can be guaranteed. Such approaches are often referred to as admission control approaches. They are commonly exploited in providing quality of service guarantees for software systems, and usually make use of a threshold to decide the acceptance or rejection of a request. In what follows, we present some examples of these admission control based approaches.

In [184], Zhou et al. have proposed an approach called selective early request termination to prevent busy Internet services from being harmed by long requests. Similarly, Blanquer et al. [31] have exploited sliding window and selective dropping to ensure the throughput and response time for Internet cluster-based servers. In [41], the authors have proposed a dynamic weighted fair sharing scheduler to control overload in web servers. The weights are adjusted to maximize the throughput objective function, partially based on session transition probabilities from one state to another, in order to avoid that the requests overwhelm the state capacity. Based on empirical parameter configuration, Urgaonkar et al. [165] have developed a system for handling the case of extreme overloads in web application servers. They study an admission control system that runs on a “sentry” tier and decides in real-time which requests to admit to ensure that the contracted performance guarantees are met. In [60], the authors have also presented an admission control mechanism for e-commerce sites that externally observes execution costs of requests, distinguishing different requests types.

Another area of research deals with performance control of Internet services using classical feedback control theory. Abdelzaher et al. [2, 3, 4] have used classical feedback control to adjust control parameters in the presence of load unpredictability based on various performance metrics, including utilization of a bottleneck resources, resource sharing, system loading, etc.. Diao et al. [55] have presented auto-tune approaches for performance and resource control by a combination of automatic system modeling mechanism and various feedback control techniques. Zhou et al. [185] have also presented a similar approach using a proportional-integral-derivative (PID) controller for queuing control, however, the controller parameters configuration in this approach is based on an empirical guess.

Compared with other control approaches, control theoretic-based approaches demonstrate advantages in their flexibility, stability, accuracy and rate of convergence. Especially for software systems that need fast reaction with good robustness, such approaches are easy to use in practice. However, to apply classical feedback control systems the mathematical model of the control system is needed. Otherwise, if the configuration of the control parameters depends on the administrator's experience, the control quality would be unreliable and non-guaranteed.

Considering the staged event-driven architecture (SEDA), Welsh et al. have presented a per stage approach to overload control, based on adaptive admission control mechanisms in each stage [171, 172]. In this approach, the overload controller observes the staged service time and tunes request rate on each stage to attempt to meet the stage's percentile response time target. Whenever the percentile response time of the stage is over the desired target, new requests are rejected. This approach is also extended to perform class-based service differentiation. However, this design has weaknesses in that it cannot efficiently achieve the performance target of the whole system in terms of the stage-based heuristic control algorithm. In addition, requests may be rejected late in the processing pipeline after it has consumed a great deal of resources in upstream stages. Li et al. [108, 110] have introduced a feedback-based controller to guarantee fair service in SEDA based applications. A limiting factor in this approach is that the controller needs a detailed knowledge about the

topology of the network of stages and the behavior of requests in the system to set a per stage target performance. In [93], the authors have also presented a feedback control-based framework that adjusts the SEDA system behavior based on the current system status in order to guide load shedding with the target of maintaining requests processing delays. The framework uses control theory and leverages mathematical methods to establish dynamic models of SEDA systems. However, a weakness in this paper is that the selection of the system model and its parameter does not depend on real relations in the staged service.

5 A Control Architecture for SEDA-Based Applications

In this thesis, the Staged Event-Driven Architecture (SEDA) is deployed as the ground-work to support highly concurrent demands. By implementing the advantages of this architecture and automatic feedback control, we propose a new approach for resource management and performance control of SEDA-based Internet services. This chapter presents a general three-layers control architecture for SEDA-based applications, explains the layers of this control architecture and introduces an overview of a proposed approach for resource management and performance control that follows this three-layers control architecture.

5.1 A Three-Layers Control Architecture

As discussed earlier, challenges that are presented by today Internet services, such as variations in workload and resource requirements, cannot be accommodated with traditional capacity planning and allocation practices. For this reason, self-managing techniques are required, which dynamically adapt the system in response to changes on the basis of short-term demands observation.

A compelling advantage of the Staged design model is that the performance of the system is relatively easy to be observed and understood. Each client request in this design is processed along a staged pipeline. Each stage in this pipeline is similar to a node in a queuing system. Parameters, such as average and maximum queue length, average and maximum wait time, and average and maximum processing time, are easily measured and displayed for each stage in the system. These measurements can be used to provide a good overview of system performance and help identifying bottlenecks [103].

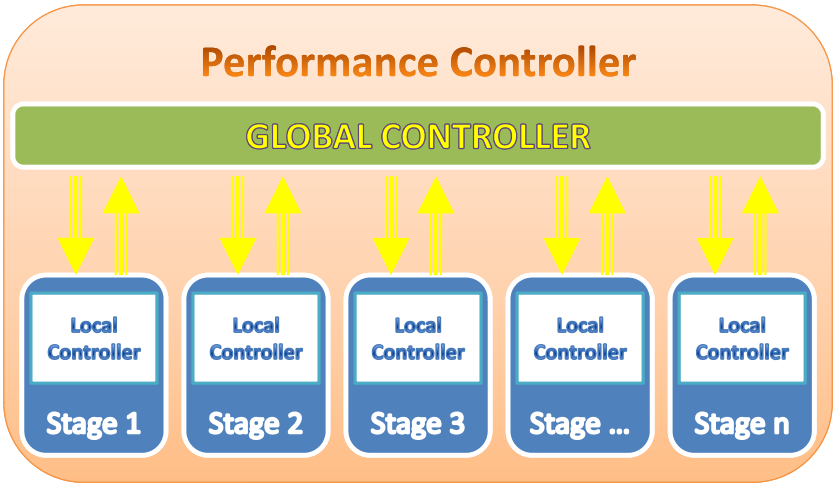


Figure 5.1: Three-Layers Control Architecture.

Based on these properties of the staged design, we propose a three-layers control architecture as depicted in Figure 5.1, for resource allocation and performance control in SEDA-based applications. This architecture consists of a local controller within each stage which is responsible for optimizing the usage of resources that are allocated to the stage, a global controller which is responsible for allocating the resources to the individual stages, and a performance controller which is responsible for controlling the overall system performance and tuning the system parameters to achieve performance targets. Next, we describe these controllers in details.

5.1.1 Local Controller

In the Staged Event-Driven Architecture (SEDA) each stage embodies an independent software component. The explicit event queues between stages separate these stages and act as execution boundaries between them. Therefore, Stages are isolated from each other and each stage is responsible only for processing the subset of requests existing in its queue. Moreover, a stage has scheduling autonomy, which enables it to control the order and concurrency with which its operations are executed. Each stage can be independently managed and stages can be run in sequence or in parallel, or a combination of the two.

When resources are allocated to a stage to be executed, this stage has to optimize the utilization of its resources. Fortunately, considering the previous stage's properties, each stage has the ability of managing these resources independently of other stages in the system. For this reason we suggest a controller within each stage that is called *Local Controller*. The task of this local controller is to adjust the stage's thread pool size in order to optimize the usage of resources that are allocated to the stage and to achieve the optimal stage performance using these resources. Each stage has its own characteristics concerning concurrency level and resource usage, which may differ from other stages in the application. For instance, the thread pool size of a processor-bounded stage should be equal to the number of processing units that are allocated to this stage. Having more threads will increase the overhead of thread management and context switches without adding any improvement to performance. On the other hand, a stage with a mixture of I/O and computation operations could have multiple threads per CPU to overcome I/O delays. For example, a stage that performs file-system accesses can usually handle up to 50 concurrent read/write requests before saturation [71]. In this case, having few threads in this stage may under-utilize the resources, while, there is no benefit to devoting more than this number of threads to this stage.

Stage's autonomy enable this controller to implement a stage-specific policy to manage the resources that are allocated to this stage. One possible approach is to implement a feedback auto-tune controller that provides a method for setting the size of the stage's thread-pool automatically taking in account the stage's dynamics, e.g. to achieve a desired utilization [4, 140], or depending on the stage's event queue size [173]. However, if the stage is programmed strictly following the event-driven approach, it will be processor-bounded as this approach tries to perform I/O operations asynchronously. As a result, few threads or even a single thread per processing unit that is allocated to this stage should be used, as it is the case in many applications that are based on event-driven programming [5, 128]. For example, Zeus Web server [183], which depends on the event-driven programming model, recommends to launch two server processes per node. Beltran et al. [28] have concluded that a simple event-driven Java server that is based on the asynchronous NIO API [169], which is used in SEDA, can scale as well as the best commercial web servers using only one or two worker threads.

Since the concurrency characteristics of stages differ, as well as the available resources for each stage, such a controller has to be implemented within the stage in a way that incorporates as much local knowledge as possible in order to automatically optimize the stage control parameters at run-time.

Another task of this controller will be to observe, collect, analyze, and report the stage's related parameters, such as event queue size, stage performance metrics, etc., in order to introduce these parameters to the other layers in the control architecture. That will increase the modularity of the system and will decrease the management costs, as the control parameters of the stage are observed and managed locally.

5.1.2 Global Controller

Different stages in a SEDA-based application may have different resource requirements and different workloads. Considering this fact and the resource allocation problem in the staged design which is presented in Section 3.2.1, system's resources have to be allocated to the individual stages taking in account their different requirements and characteristics.

To perform this task, we propose a second layer in the control architecture that is called *Global Controller*. This controller is responsible for allocating resources to stages depending on their different needs and performance. This controller must be global in respect to stages in order to decide on the resource allocation so that it avoids bottlenecks in the system by giving more resources to a stage when it becomes a potential performance bottleneck, and it also avoids wasting resources by allocating them to a stage that cannot utilize these resources. Accordingly, the task of this controller is to allocate resources so that it balances the performance of the stages and solves the resource allocation problem.

This controller interacts with the individual local controllers of the stages in a two-way fashion. The global controller gets information about local controllers observations on performance metrics and queues sizes of the individual stages as input. Depending on these inputs, the output of this controller is the allocation of resources to stages, which then are managed by the local

controllers. The global controller just provides specifications that constrain the resources that are allocated to an individual stage without dictating which requests (events) are to be processed at each stage or how to process these requests. The latter is a separate policy that may be provided by the local controllers in a stage-specific fashion.

The global controller can be considered as a feedback-based scheduler that distributes the available system resources among the individual stages in such a way that the whole system performance is maximized by balancing the load and performance of these stages.

5.1.3 Performance Controller

The task of the global controller is to maximize the performance of the staged application using the available system resources. However, Internet services need also to control the performance metrics to guarantee performance levels and to achieve a variety of performance targets. In our control architecture a third layer which is called *Performance Controller* is responsible for this task.

Depending on the targets of the system and by implementing performance control techniques, the performance controller interacts with the global controller to achieve the desired system performance. This controller uses measurements of the system outputs, such as response times, throughputs, and resource utilizations, to adjust the parameters used by the global controller to assign resources to the individual stages. These assigned resources are then managed by the local controllers. Based on observations of changes in these measurements the performance controller can adapt the system automatically at run time to achieve the specified goals.

As the specified system goals are usually at the application/service level, not at per stage level, having the performance controller in a higher layer in the control architecture gives it the ability to control the system performance using simple control algorithms. Using our control architecture the performance controller can automatically and dynamically tune the system to achieve desired performance targets, without dealing with complications or being lost in the details of the system.

5.2 An Overview of The Proposed Approach

This section presents an overview of the proposed resource management and performance control approaches that will be presented in the following chapters of this thesis, and maps these approaches to the three-layers control architecture. As discussed in Section 2.2, processing resources are increasingly the critical resources in today Internet services. For this reason the proposed approach is mainly dedicated to these resources. However, our approach is capable to take other resources, e.g. I/O bandwidth, into account.

The idea of the proposed approach is based on processing requests at each stage as batches, which means to consider multiple requests that are queued at a stage as a unique job unit and processing them consecutively without interleaving their execution with the processing of other requests. The novelty of our approach is that it depends on the allocation of processing resources in the system as time slots to create these batches. Within each time slot portions of the processing time are assigned to the individual stages, depending on the estimated needs of each stage. At each time slot a feedback-based controller adjusts these portions dynamically and automatically as the requirements of the stages change. The controller allocates processing resources to stages in a way that balances the performance of the stages, as it depends on the stages needs, and as a result it maximizes the system end performance. This controller represents the *Global Controller* of the three-layers control architecture in our approach.

Considering that our approach allocates a processing unit to process a batch of requests at one stage before moving to process another batch, which may be from another stage, that means that this processing unit is dedicated to a single stage during the period of batch processing. That helps isolating stages from each other, and as a result a *Local Controller* can be implemented within each stage that uses more sophisticated approaches to adjust the stage's thread pool sizes. For example, the local controller can use an approach that sets an upper bound on the utilization of the processing unit that is allocated to the stage (to process the batch) and dynamically adjusts the thread pool size based on the measured utilization. Such an approach gives the opportunity to a fine-grained thread-pool size control per stage that corresponds to the concurrency level of the stage, which will be intuitively more efficient than other considera-

tions. In contrast, the thread pool size manager of the original SEDA allocates resources based on the load of the stage rather than the concurrency characteristics of this stage. As a result, the number of threads in each stage is not related to the concurrency level of the stage which leads to decreased performance due to increased overhead of threads management within a stage, context switches and resource contentions – this problem is tackled in SEDA by setting a maximum threshold on the number of threads in a stage. As discussed earlier, for scenarios, where the event handler of a stage uses asynchronous I/O primitives, we believe that one or two threads per processing unit are most suitable for SEDA stages.

A key parameter in our approach is the duration of the time slot that we use to allocate the processing resources. The value of this parameter determines batch sizes, determines the time between two assignments of processing units to a stage, and determines also the utilization of other system resources. As a result, this value affects the throughput of the system as well as the response times of the individual requests. Therefore, controlling the size of the time slot can be used to achieve a variety of performance targets. A *Performance Controller* can be implemented that monitors the end to end system performance (the performance metric that is to be controlled) and adjusts the time slot size to achieve externally specified goals. Based on observations of the relation between the time slot size and the controlled performance metric the performance controller can dynamically adjust the time slot size at run time to response to any changes in the system.

Although the proposed approach will delay requests that have to wait for other requests at the same stage or at other stages, this approach has many benefits that will be demonstrated throughout this thesis. However, the feasibility of this approach comes from the fact that the tolerated response time for Internet services is considerably larger than the needed service time for a single request. For this reason, techniques that may cause delaying responses in order to optimize other performance metrics are applicable in Internet services.

6 Adaptive Resource Allocation for Staged Services

As presented in Chapter 3, resource allocation in staged design-based services presents many challenges. This chapter presents a strategy to allocate processing resources in these services, which adapts the resources assigned to each stage based on observations of the changes in the system. The chapter presents also a validation and evaluation of the strategy through a simulation study. Considering the three-layers control architecture which is presented in Chapter 5, the resource allocation strategy which is introduced in this chapter represents a global controller.

6.1 Problem Statement

6.1.1 The System Model

We consider a server with N processing chips and a staged application that consists of M stages. Each processor may be a multi-core chip that has C cores. In this case, a chip will have a last level cache that is shared among cores and each core will have its private caches.

A client request arriving into the system is processed by a subset of the stages prior to departure. Each stage is associated with a set of instructions and data. When a processing unit resumes processing a request at a new stage, stage A for example, during the execution of this request the processing unit spends time loading the code and data of stage A into its caches. Since these data and code are shared in average between all requests executing in the stage A, if the processing unit continues processing requests from this stage it avoids this loading time, and the request will use significantly less CPU time if both code and data are small enough to fit into the cache memory. We consider here in our model that a request's private data is very small compared to the

stage code and data, so that we can ignore its load time in comparison to the stage load time. Allocating the processing unit to process a request from another stage will evict the code and data structures of stage A from the cache to replace them with the code and data of the new stage. In our model, this effect is represented by a load time l_i needed for each stage i to load its data structures and instructions into the caches of a processing unit that resumes processing a request at the stage. In real systems, accesses to data structures that do not exist in the cache take place during execution, not only once at the begin of the execution, and cause multiple cache misses that need more time to be handled. This load time is determined by the latency to access other memory levels in addition to the memory bandwidth. Overlapping loading with processing can reduce the wasted CPU time, if the needed memory blocks are known in advance, however, this is not always the case.

In a multi-processor system, processing multiple requests in parallel on different processing units may cause additional CPU time overhead related to cache coherency effects and the contention of the concurrently processed requests for shared resources (shared data structures, shared locks, bus bandwidth, etc.). In our model, we categorize these effects as the overhead caused by processing requests from the same stage in parallel on other processing units, and the overhead caused by processing requests from other stages in parallel on other processing units.

Within each category, we also differentiate between two types of overheads: the overhead of running requests in parallel on other chips and the overhead of running requests in parallel on other cores at the same chip. This differentiation is motivated by the fact that when requests are processed on different cores at the same chip they share data that exist in caches of this chip. While, in the case of running requests on other chips, requests will need to access data structures that reside on other chips, which is significantly more expensive.

- Processing requests from the same stage in parallel:
  When we allocate more than one processing unit to a stage i to process requests from its incoming queue, multiple requests from this stage are processed in parallel. These requests share many data structures and variables since they are processed at the same stage –see the character-

istics of the staged design Section 3.1. For this reason, shared writable data and critical sections must be protected by mechanisms to prevent race conditions and other effects. These mechanisms, locking for example, may cause degrading concurrency and result in an increase of per request processing time. As the number of requests that are processed in parallel increases, contention for shared locks and data structures increases and this processing time overhead increases even more.

In addition, as multiple copies of these shared data structures exist in different places in the cache-memory hierarchy additional overhead may result in when accessing writable data structures. This overhead results from the used cache coherency protocol, which is responsible for the integrity of data stored in local caches of each processing unit. As the number of requests that are processed in parallel from the same stage increases, the number of processing units that have the shared data in their local caches increases and as a result the overhead of the cache coherency increases.

One of the main issues that affect these overheads is the available data rate of the different levels of cache in the processing units hierarchy. Considering this and the fact that data rate decreases as we move to the bottom of the memory/cache hierarchy (from the private core cache to the main memory), we can see that processing units that share data structures that are accessed through a lower level in the hierarchy have to share a smaller available data rate, which increase the overheads of locking mechanisms and coherency protocols.

In our model, the accumulated overhead of these effects is represented by two parameters OHL_{ii} (OverHead results from requests that are processed from the same stage i Locally on other cores at the same chip –e.g. requests from Stage A that are processed on the two cores at Socket 1 in Figure 6.1) and OHR_{ii} (OverHead results from requests that are processed from the same stage i on Remote or other chips –e.g. requests from Stage A that are processed on cores at Socket 1 and Socket 3 in the Figure), which are additional processing time needed per request. These overheads increase as requests share more writable data structures and

as requests spend more time in their critical sections. However, processing the requests of a stage on the same chip, or more generally as close as possible in the cache-memory hierarchy, rather than on other chips can reduce the effect of these overheads, because of the higher speed of communications between processing units. In addition, this can better utilize the available data rate and memory bandwidth, since the more expensive data accesses use the larger data rates. For example, in Figure 6.1, requests from Stage A are processed on the two cores of the chip at Socket 1. As a result, they communicate using the shared cache level (L3-cache), which has a data rate of more than 150 GB/s in today systems [66]. While these requests communicate with requests from Stage A that are processed on a core at Socket 3 using cross chip communications, which have a maximum data rate of less than 55 GB/s [177].

- Processing requests from different stages in parallel:
  Since different stages may share global variables or some other data structures and need to use shared resources, the same overheads may appear when processing requests from other stages in parallel on other processing units. These effects represented in our model as OHL_{ij} which is the additional processing time needed by a request processing from stage i as a result of processing requests from stage j in parallel on other cores at the same chip –e.g. requests that are processed from Stage B and Stage C on the cores of the chip at Socket 0 in Figure 6.1, and OHR_{ij} which is the additional processing time needed by a request processing from stage i as a result of processing requests from stage j in parallel on other chips –e.g. requests that are processed from Stage A at Socket 1 and that are processed from Stage B at the sockets Socket 0 and Socket 2 in the Figure.

  As requests from different stages share less data structures than requests from the same stage, hence, processing requests from different stages in parallel will cause less overhead of coherency protocols and lock contention than processing requests from the same stage. However, if we process requests from multiple stages in parallel so that they content for a shared cache-memory level and each stage evicts the data sets of other stages this will reduce the performance even further. In this case, memory bandwidth can be a performance bottleneck too that increases the

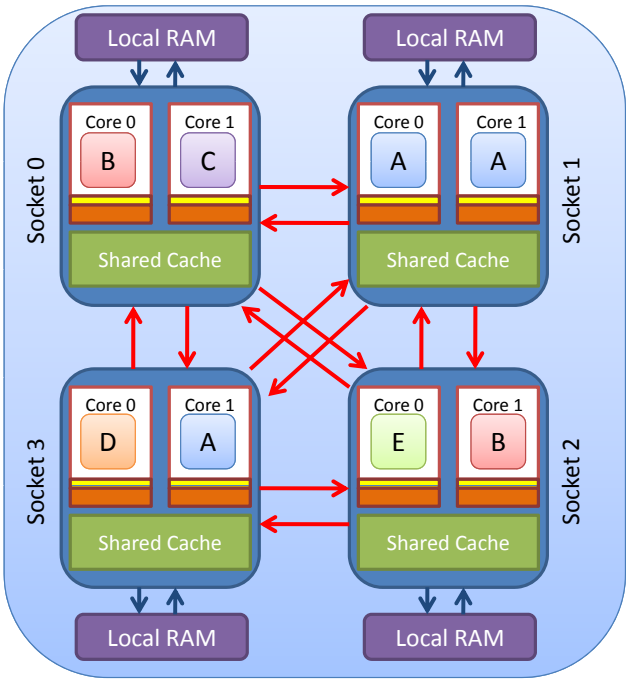


Figure 6.1: Different Communication Overheads

processing overhead.

Values of the considered overhead parameters are mainly related to the number of requests that are running in parallel in the system. As this number increases, those values increase; and as this number decreases the values decrease too. The values of overhead parameters also vary according to the several system related aspects that we have already discussed, and depend on the type of resources on which contentions among the concurrent requests exist. A detailed study is needed to determine these relations, which is out of the scope of the thesis and considered as a very important area for future work in the staged design.

However, shared resources can be modeled as waiting queues [68, 74, 115]. In our system, each of the shared resources, e.g. a lock or a shared data structure, can be represented by a queueing model, which has a finite population size that is the number of requests (or stages) that content for this shared resource [79]. Motivated by this fact and for simplification in our study, we approximate the overheads by assuming a linear relationship between these overheads' values and the number of requests that are processed in parallel,

while considering a constant effect of the other system related aspects, i.e.:

$$\forall i,j; OHL_{ij} = OHL_{ij}(X_{ij}) = OHLV_{ij}.X_{ij}$$

and

$$\forall i,j; OHR_{ij} = OHR_{ij}(Y_{ij}) = OHRV_{ij}.Y_{ij}$$

where X_{ij} is the number of requests from stage j that are processed with the request from stage i in parallel on cores at the same chip, and Y_{ij} is the number of requests from stage j that are processed with the request from stage i in parallel but on other chips. $OHLV_{ij}$ and $OHRV_{ij}$ are system specific constants that represent the discussed system related aspects. Hence, the total processing time needed by a request at stage i is:

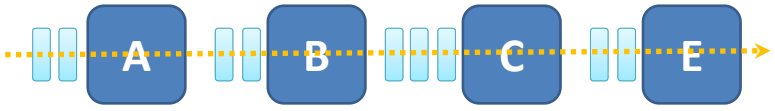
$$\Upsilon_i = e_i + \sum_j OHLV_{ij}.X_{ij} + \sum_j OHRV_{ij}.Y_{ij} \tag{6.1}$$

considering e_i the mean processing time that is needed by a request at stage i if no requests are processed in parallel.

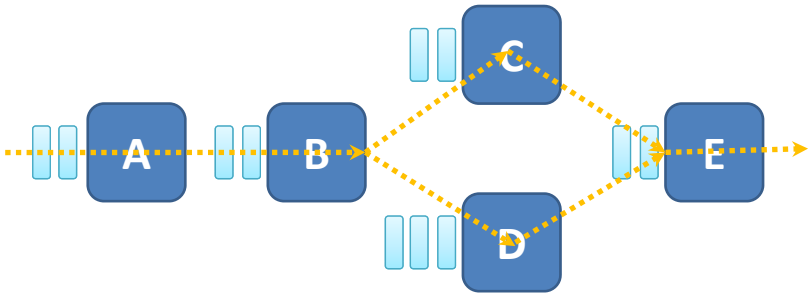
6.1.2 Resource Allocation Problem

Considering the presented system model, the objective in this chapter is to introduce an adaptive resource allocation algorithm to allocate the available processing units in the system to the individual stages at run time. Our target is an algorithm that balances the performance of the individual stages to improve the performance of the system, provides cohort scheduling in order to benefit from locality within stages, avoids the overheads related to parallel processing taking into account the hierarchic parallelism of the server machine, copes with the unpredictable changes in the characteristics of the system and the workload, and avoids oscillations in performance.

As stated earlier, a staged application is specified as a network (or graph) of stages, see Figure 6.2. A client request is processed along a pipeline that consists of a subset of these stages. The overall performance of the system is limited by the performance of a bottleneck stage in the pipeline. Allocating less resources to a stage than it needs may result in having a bottleneck and decreasing the system's performance. To increase the system's perfor-



(a) Simple Pipeline



(b) Network of Stages

Figure 6.2: Staged Application

mance, we must allocate more resources to this bottleneck stage. However, allocating more resources to a stage than it needs will waste these resources. Moreover, since all stages are competing for the same resources and given the dynamic nature of both system workload and resource requirements of individual stages, this increases the complexity of the problem even further.

In addition, as we allocate these resources we need to make a trade-off between running multiple requests from the same stage and running multiple requests from different stages in parallel in order to benefit from locality and at the same time not to be over-committed by parallel processing overheads. Running requests from the same stage in parallel will increase locality as requests reuse data that exist in the cache, but it will increase the overhead of locking and coherency protocols, too. On the other hand, running requests from different stages in parallel will decrease contention for shared data structures and decrease the effect of coherency, as less data is shared, but this will cause more contention for shared cache-memory levels and memory bandwidth.

6.1.3 Resource Management Goals

Before presenting the proposed resource allocation approach, we list our targets and show how this approach is related to the three-layers control architecture, which is presented in Chapter 5.

Regarding the problem under consideration and the studied system model, our approach to allocate processing resources to the individual stages of a SEDA-based Internet service has the following targets:

1. Cohort scheduling: Staged design introduces a programming abstraction to implement cohort scheduling policies [103], and our approach must provide cohort scheduling in order to increase the system's performance through the benefit from instruction and data locality within a stage.

2. Avoid parallel processing overheads: As mentioned earlier, in the staged design requests that are processed from the same stage share more data than requests that are processed from different stages [77, 103, 173]. For this reason, when processing requests from the same stage in parallel, we allocate processing units to these requests so that they communicate through a path with a large data rate, which means to process them as close as possible in the processing units hierarchy (on the cores of the same chip for example). This will reduce the overheads that result from parallel processing of these requests, like coherency protocols and locking mechanisms overheads, and decrease their impact on performance, because of the higher communication speed and the larger bandwidth. At the same time, requests from different stages are processed on different processing units that may have a lower data rate communication among them, but never share a crucial cache-memory level, in order to avoid the contention for available cache/memory space.

3. Balancing resource allocation to the individual stages in order to avoid having bottlenecks in the system which can impact the whole system's performance, and to avoid wasting resources by giving a stage more resources than it actually needs. From another point of view, this will also balance the load on the individual processing units so that we do not leave a processing unit idle while other units are over-loaded.

4. Adapt the allocation dynamically and automatically in order to cope with changes in system and workload characteristics. Loads on the different stages in the system may change at run time as will as the resource requirements of a specific stage, as a result of changes in the behavior of clients for example. For this reason, the allocation policy has to respond to these changes automatically at run-time.

Chapter 5 presents a three-layers control architecture for resource management and performance control of SEDA-based applications. This architecture consists of a local controller within each stage to optimize the usage of resources that are allocated to the stage, a global controller that allocates the available resources to the individual stages, and a performance controller that changes the system parameters to achieve a variety of performance targets. Regarding this control architecture, the resource allocation approach presented in this chapter is a global controller that balances the allocation of the available processing units in the system to the individual stages.

6.2 The Proposed Approach

6.2.1 Resource Allocation Policy

To maximize the overall throughput of the system for given available resources, we must balance the performance of the different stages. In the case of a simple pipeline (Figure 6.2a), the throughput of each stage must be equal to the throughput of other stages in the pipeline. That can be achieved by a simple policy that gives CPU time to each stage proportionally to its estimated performance. In the general case having a network of stages (Figure 6.2b), different stages would have different loads depending on the flow of events in the system, so load has to be considered when assigning resources.

Additionally, in order to benefit from the staged design and to exploit the locality within stages, we have to batch the execution of multiple requests at each stage on a processing unit before moving this processing unit to execute requests at another stage. The size of these batches must be controlled in order to achieve two targets: First, the size of the batches must be adjusted in order to balance the throughput of different stages and as a result optimizing the overall system throughput. Second, the size of these batches must be controlled to avoid oscillations in the overall throughput and to satisfy other performance goals (response time target, resources utilization, etc.).

The idea of the proposed approach is based on this batch processing at each stage. We consider multiple requests that are queued at a stage as one job unit and processing them consecutively without interleaving their execution with the processing of other requests. Our approach depends on the alloca-

tion of processing resources in the system as time slots to create these batches and to determine their sizes. Within each time slot, portions of the processing time are assigned to the individual stages, based on observations of load and performance of the stages in the system, which help in estimating the needs of each stage. At each time slot a controller adjusts these portions dynamically and automatically, so that they are changed as the requirements of the stages change. The controller allocates processing resources to stages in a way that balances the performance of the stages, as it depends on the stages needs, and as a result it maximizes the end to end system performance.

6.2.1.1 Simple Pipeline

In this simple case, the system is a simple pipeline of stages. A request therefore passes sequentially through the M stages as depicted in Figure 6.2a. As already mentioned, in this case we distribute, the CPU time proportionally to the performance of the stages.

Let $th_i(k)$ be the average throughput of stage i estimated in the last sample period k and let T be the period of CPU time which we want to allocate to the stages in the next sample (T is the size of the time slot). To achieve the maximum throughput the CPU has to be allocated in the next period so that the following relation is satisfied:

$$\forall i, j \in \{1, \ldots, M\}; th_i(k) \cdot t_i(k+1) = th_j(k) \cdot t_j(k+1) \tag{6.2}$$

where $t_i(k+1)$ is the CPU time that would be allocated to the stage i during the next time slot T.

Together with the requirement that

$$\sum_{1 \leq i \leq M} t_i(k+1) = T \tag{6.3}$$

we get a linear equations system from which we can get the values of $t_i(k+1)$ which are the allocations in the next period T:

$$t_i(k+1) = \frac{T}{th_i(k)} \cdot \frac{1}{\sum_j th_j(k)^{-1}} \tag{6.4}$$

Our controller uses this formula at each time slot to calculate the CPU time assignments to the stages during the next period. At each time slot the controller has to read the estimated performance of each stage and then calculate the new allocations.

6.2.1.2 Network of Stages

In this generic case, a request has an execution path through a subset of stages depending on the type of this request. For example, in Figure 6.2b requests enter the system at stage A and after being processed at stage B they go to one of the two stages C or D depending on the request's type, then move to stage E and leave the system.

Let $th_i(k)$ be the estimated throughput of stage i in the last sample period, and $S_i(k+1)$ the number of requests in the queue of the stage i. From these two values we can calculate the estimated processing time needed by this stage to process all the requests in its incoming queue:

$$w_i(k+1) = \frac{S_i(k+1)}{th_i(k)}; i \in \{1, .., M\} \tag{6.5}$$

Depending on these values, we can now calculate the portions of processing time that are to be given to the stages in the next time slot T:

$$p_i(k+1) = \frac{w_i(k+1)}{\sum_j w_j(k+1)} \tag{6.6}$$

The CPU time that is to be allocated to each stage during the next time slot is

$$t_i(k+1) = p_i(k+1) \cdot T \tag{6.7}$$

At this point, we have two choices: Either, we allocate these times to stages by setting a timer which will interrupt the execution of a stage when its allocated time elapsed, or we calculate a batch size which determines the number of requests that are processed from the stage's queue before moving to another stage, based on the time that we want to allocate to the stage. Since the stages are based on the event-driven programming model, the latter approach seems more efficient as it does not interrupt the processing of a request, therefore using cache locality better. For this reason, we consider this approach from now on and derive the batch sizes by:

$$B_i(k+1) = t_i(k+1) \cdot th_i(k) \tag{6.8}$$

Allocating CPU time this way ensures that each stage gets its required processing time in the next period and as a result the throughput of the stages will be balanced which is potential to increase the overall performance of the system.

6.2.2 Overload Protection

Since the proposed allocation policy depends on the number of requests in the event queues, it is sensible to overload which is not avoidable in today's servers even with careful capacity planning – see for example what happened to the "Microsoft Photosynth" at its first launch [23]. In overload situations, the number of requests in the system will over-commit the available resources. In our case the requests enter the system at the first stage. Since the CPU time allocated to a stage is proportional to the number of requests in its queue that means that the first stage will get the majority of CPU time, in the case of overload. Therefore, the portions of CPU time of other stages decrease, which will decrease their throughput and as a result decreases the overall system throughput. To avoid this effect, we have to modify the resource allocation algorithm so that it avoids this behavior.

At each period we calculate the estimated CPU time $w_i(k+1)$ needed for stage i as previously given in equation 6.5 except for the entrance stage (stage A) and allocate for each stage i:

$$t_i(k+1) = w_i(k+1);\ 2 \leq i \leq M \tag{6.9}$$

If the sum of these values is less than the size of the used time slot T, we give the remaining time in the next time slot to the first stage to process requests from its queue. If not, the first stage will wait until the following time slot to be executed. Allocating CPU time this way ensures that we only accept requests at the entrance stage that can be processed in the next time slot, therefore avoiding over-commitment of the processing resources.

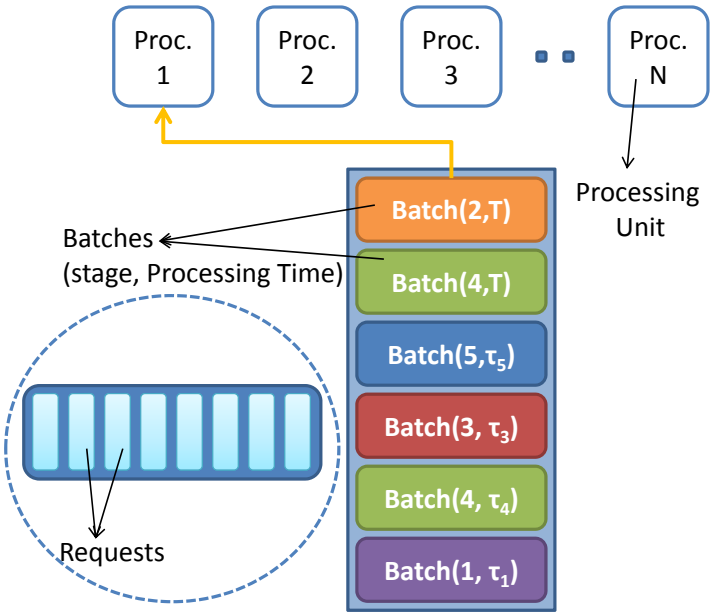


Figure 6.3: Batches Processing

6.2.3 The Case of Multiple Processing Units

Up to this point the proposed policy does not consider the existence of multiple processing units and a parallelism hierarchy in the system. As stated earlier, the proposed approach is based on allocating the processing resources as time slots of the size T to the staged service. At each time slot we calculate the processing time that is to be allocated to each stage within this time slot. After that, we create batches of stages requests (events) and then consider each batch as a job unit to be processed by a processing unit. In the case of running the staged server on a machine consisting of N processing units (multi-core system , SMP or a cluster), the available processing time within a time slot of the size T will be $T.N$.

As previously, let $th_i(k)$ be the estimated throughput of stage i in the last sampling period. As the number of allocated processing units to a stage may change at each time slot, for this reason, the stage's throughput considered here is given by the number of requests processed per time unit per processing unit. This throughput can be calculated by dividing the number of finished requests at this stage within the last sampling period by the sum of processing times allocated to the stage within this period. In addition, let $S_i(k+1)$ be the

number of requests that exist in the queue of the stage i. The values $w_i(k+1)$ and $p_i(k+1)$ can also be calculated from the equations 6.5 and 6.6.

However, the processing time to be allocated to each stage during the next time slot is:

$$t_i(k+1) = p_i(k+1).T.N;\ 1 \leq i \leq M \tag{6.10}$$

and to avoid the impact on performance in the case of overload, allocated processor times have to be calculated like in Section 6.2.2.

Depending on these values, we now create batches of stages requests as follows:

Firstly, for each stage i, we can write:

$$t_i(k+1) = c_i T + \tau_i\,;\ \tau_i < T,\ c_i \in \mathbb{N}$$

Then, from the requests in the queue of stage i we create c_i batches of the size ($T.th_i(k)$); each one of these batches will need approximately one processing unit for the whole period of the next time slot T to be processed. We create also one batch of the size ($\tau_i.th_i(k)$) which will need approximately τ_i time units to be processed. After creating these batches from each stage, we put them in a run queue that is shared among the processing units, to be processed in the next time slot –see Figure 6.3. Each job in this queue is a batch of requests from a unique stage, and we sort these jobs in descending order depending on the time each job needs to be processed, so that we can easily select the next batch that needs the most processing time. Each batch is allocated to one processing unit and the processing unit processes all the requests in the batch before taking the next batch. Figure 6.4 shows how the processing units are allocated to the stages at each time slot according to this scheduling policy. If the batches' queue (jobs' queue) is empty, an idle processing unit tries to process requests from the batches that are processed by other processing units, to avoid wasting a very long idle time.

Now, considering the hierarchic parallelism in our model how to allocate this hierarchy of processing units to the individual stages?

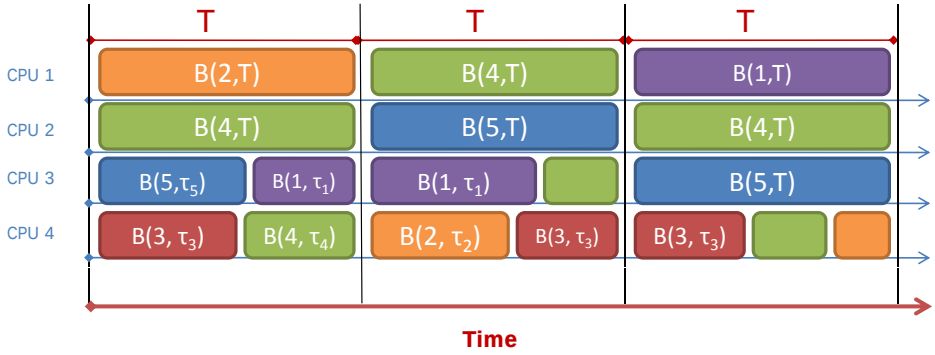


Figure 6.4: Allocating Processing Units to Stages Based on Batches Processing.

As the benefit from locality within a stage is a main target of the staged design, scheduling decisions are mainly related to the stage's working set size. In this context stage's working set means the code and data that are accessed by a request that is processed at this stage[1]. If the stage's working set fits into the cache of the processing unit, we can avoid the overhead of cache misses by processing multiple requests from a stage that has its working set in the cache.

In general, the relation between the stage working set size and the size of the different levels of memory-cache hierarchy may vary. However, since applications can use CPUs, cores and threads topology information exported by the operating system [146], a more sophisticated approach is to combine a Processing Units Tree (PUT) with our resource allocation approach. The Processing Units Tree represents the hardware hierarchy of the processing units and the scheduling domains in the system [59], and it can be used to guide the controller during the allocation of batches to the individual processing units so that we process batches from the same stage as close in the tree as possible to reduce the overhead resulting from parallel processing. Figure 6.5 shows a PUT that represents a system which has four chips and each chip has two cores. More complex tree structures (more levels and more nodes) are possible that represent more complex hardware architectures, since future system are expected to have higher memory hierarchy and more parallelism [46, 80, 90]. Allocation policies that consider the stage's working set size when co-allocating batches from different stages on a node in the tree are also possible.

[1]Stage's working set in this sense is comparable with the term "Task Region" in the OpenMP terminology [24].

However, here we consider two cases regarding the relation between stage's working set (SWS) and the cache hierarchy:

1. SWS $\leq$ Per-Core-Cache:
   In this case the stage's working set fits into the core's private cache (usually L2-cache) and we can use the proposed approach without any modification only by considering each core as an independent processing unit and taking scheduling decisions at core's level with replacing N in Equation 6.10 by $N.C$, where C is the number of cores per chip. Although in this case applying the scheduling policy at the chip's level is possible, applying it at the core's level will increase the performance further more. In this case, the number of requests from the same stage that are processed in parallel on other cores on the same chip will decrease and as a result decreases the overhead of accesses to the next cache level (L3) which are more expensive.

2. Per-Core-Cache $<$ SWS $\leq$ On-Chip Shared Cache:
   In this case the stage's working set fits into the shared on-chip cache (usually L3-Cache), and we take the scheduling decision at the chip's level (Process a batch by a chip) to avoid the contention among multiple different stages running on the same chip for the shared cache.

Many other cases are also existed that depend on the hardware architecture and the different sizes of caches and stage's working sets. A detailed study of these different cases and their effects is an area for future work, especially the effects of the parallelism hierarchy when the stage working set does not fit into both cache levels.

6.3 Evaluation and Analysis

We validate our proposed scheduling policy by comparing it with other scheduling policies under various load conditions through a simulation study. The experiments build on the general case of a network of stages (see Figure 6.2) since this is the generic case that can easily be reduced to simpler cases such as a pipeline of stages.

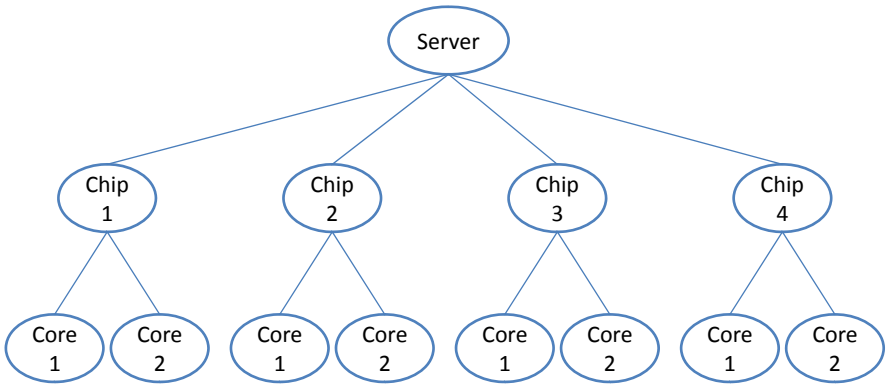


Figure 6.5: Processing Units Tree

6.3.1 Experiments Environment

In order to validate our proposed approach and to evaluate its performance, we apply a simulation study on a prototype of a staged design-based service that we have implemented using the Möbius environment [1]. In our prototype, the system consists of five stages as depicted in Figure 6.2b[2]. Requests enter the system at the first stage (stage A), with exponentially distributed inter-arrival times. Then, these requests sequentially pass through other stages until they leave the system at stage (E). After being processed at stage (B), a request passes to one of the two stages (C or D) depending on the request type. Requests arriving at each stage are put in a dedicated waiting queue. When a processing unit is given to a stage to process requests from its queue, the stage must first load its common data structures and code, which takes a time l_i. Next, requests from the queue are processed without further need to load data. In our implemented model, a single request will have a service time at stage i which is derived from an exponential distributed value with a mean e_i, and the requests in a queue are processed in a first in first out (FIFO) order. We distinguish between the two stages (C and D) by giving the stage (C) a larger mean service time to make it a bottleneck stage in the work-flow.

Here, we consider that a stage's working set fits into the shared on-chip cache level but not in the per core private caches, for this reason our approach makes

[2]We have conducted many of the experiments that are presented in what follows on other prototypes too (a simple pipeline and a more complex network of stages). The results for these prototypes was very similar to the results for the prototype that is presented here.

scheduling decisions at chip's level. In addition, we ignore the additional cache space that is needed as a result of processing multiple requests from the same stage on a chip, since in the staged design it is very small in comparison to the stage's working set size.

6.3.2 Experimental Results

The presented experiments evaluate the ability of the proposed approach to benefit from stage's locality, and evaluate the effect of a variety of system's characteristics on the performance of our approach. However, we begin with a simple evaluation of the performance benefit of cohort scheduling policies.

6.3.2.1 Evaluation of Cohort Scheduling Effect

To evaluate the effect of cohort scheduling on performance we have implemented a prototype of an application that consists of two stages. The evaluation is done by executing the first stage once then executing the second stage, and after that executing the first stage again many times without interleaving executions of the second stage. In each execution we measure the needed time to finish the stage execution. At the beginning we execute the first stage then the second stage once then the execution delayed for one second before taking measurements. The reason of this action is to avoid unexpected overheads that can cause the first execution to take a very long time such as the memory reservations and the initial creation of data structures. Figure 6.6 shows the code used for our evaluation.

Several experiments with different codes and functionalities of the stages "$Stage1$" and "$Stage2$" have been carried out. The results of these experiments have shown values of a stage load time which vary from less that 10% up to more than 200% of the stage execution time. This load time is calculated as the difference between the time of the first execution of $Stage1$ after the one second delay and the average time of the following executions. As can be seen from Figure 6.6, we have used the Time Stamp Counter [179] to measure how long it takes for an execution of a stage. Table 6.1 shows examples of our experiments results[3]. In the first experiment (Exp.1 in the table) $Stage1$ reads an array of

[3]The presented results are the average of three runs, and they are given by the number of the system clock ticks

```
#include <stdio.h>
#include <unistd.h>
#include "tsc-test.h"
#include <time.h>

int i;

unsigned long long start, stop;

unsigned lo1,lo2,hi1,hi2;

unsigned long long time_Val[25];

int main()
{
  i=0;

  Stage1();

  Stage2();

  usleep(1000000);

  printf("Starting...\n");

  while(i<25)
    {
      asm volatile ("rdtsc" : "=a" (lo1), "=d" (hi1));

      Stage1();

      asm volatile ("rdtsc" : "=a" (lo2), "=d" (hi2));

      start = (unsigned long long) hi1 << 32 | lo1;
      stop = (unsigned long long) hi2 << 32 | lo2;

      time_Val[i]= stop-start;

      i++;
    }

   i=0;


   while(i<25)
    {
      printf("%d\n",time_Val[i]);

      i++;
    }
}
```

Figure 6.6: Cohort Scheduling Evaluation Code

	Exp.1	Exp.2	Exp.3
SWS Size (KB)	10	374	748
First Run (ticks)	9061397	187597	361531
Next Runs (ticks)	8710881	89037	184247

Table 6.1: Performance Benefit of Cohort Scheduling

characters from the memory and writes them in a file. This experiment shows a small difference between the time it takes for the first run of the stage and the next runs. In the other experiments (Exp.2 and Exp.3 in the table), $Stage1$ also reads an array of characters from the memory and writes them to another array in the memory. We can see here a large difference between the first and the next runs of the stage.

A detailed study of the factors that affect the benefit from cohort scheduling, especially the effect of stage's code size and stage's data size combinations, deserves more investigations, which is out the scope of this thesis. However, even with small values of cohort scheduling performance benefit, such scheduling policies are valuable because of the highly frequent context switches in SEDA-based server applications, which may happen whenever events pass from one stage to another. For example, experiments have shown that a web server which is based on SEDA can cause more than $30,000$ context switches per second [129, 167].

6.3.2.2 Unique Processing Unit

Now we want to evaluate the efficiency of the proposed approach as a cohort scheduling policy. Since existing cohort scheduling policies have been evaluated for uni-processor systems [76], and to avoid the effects of other factors in the system, we consider a uni-processor system in this section.

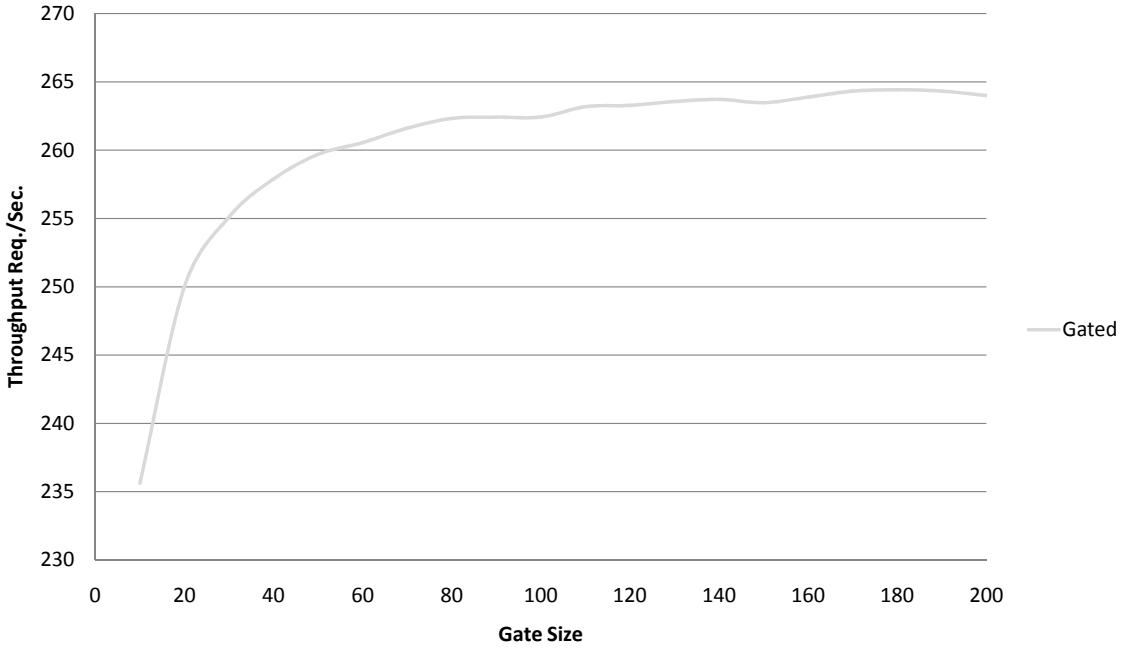
The presented experiments compare our approach with three basic approaches called "Random", "All" and "Gated". In the "Random" approach, the CPU is given to one stage selected randomly to execute exactly one request from its queue. This approach ignores the cache effect and we use it as a base to see the ability of the compared approaches to exploit the locality within a stage. This approach can also be considered as a representation of the two traditional concurrency approaches, event-driven and thread-based concurrency –

see Section 2.3. In contrast, "All" is the optimal approach regarding the benefit from locality within stages. In this approach, the processor is supplied to the stages in a simple front-wave algorithm. At each stage, the processor executes all the requests pending in its queue as a batch before continuing to process requests at the next stages. The third approach is the "Gated" approach, which have been implemented for staged database systems [76] and staged web servers [103]. This approach is similar to the "All" approach but it puts a threshold, called the gate size (g), on the number of requests processed as a batch at each stage. If there are less than g requests in the queue of a stage, all of them will be processed as one batch, while otherwise the CPU is given to the next stage after processing only g requests from the stage's queue.

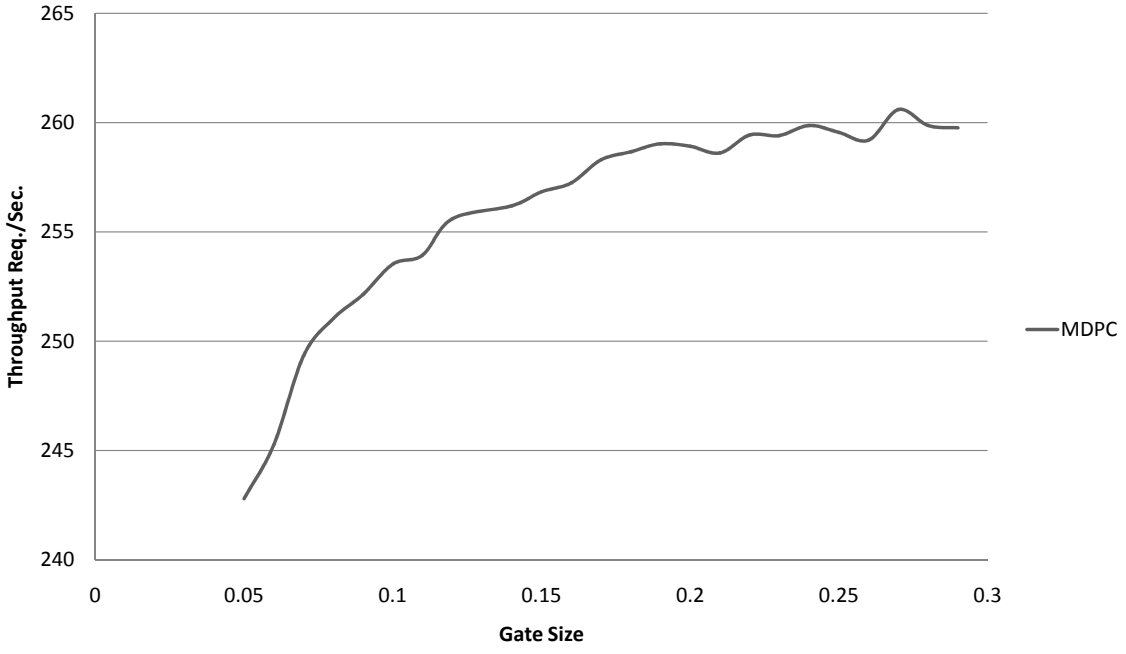
It is clear that the performance of our approach is affected by the value of T which is subject to the performance controller, while the performance of the "Gated" approach is affected by the gate size g. In both cases, these values determine the size of batches which affects the benefit from locality within stages. Figure 6.7a and Figure 6.7b show the throughput of the "Gated" approach and the throughput of our approach using different values of g and T. In our experiments, both parameters were independently tuned, and we have chosen suitable values for T and g which give a high throughput and avoid having a very large batch size which will penalize response times.

In the presented results of this section, the mean processing time that is needed by a request at stage i (e_i) takes the values: 1 ms for stages A, B and D; 2 ms for stage C and 0.5 ms for stage E. For each stage the stage load time l_i is equal to 1 ms. T is equal to 200 ms and g is equal to 150. Both values have been determined experimentally, so that the approaches show high and comparable performance. Other values of these parameters have shown similar results – see our previous evaluation results in [11]. The presented results are the average of multiple simulation runs with relative confidence intervals of 10% and a confidence level of more than 90%.

1. Throughput and Response Time:
   In this experiment we compare the approaches under different load conditions. Figure 6.8 shows how the average system throughput (within a time of 100 seconds) changes under different request arrival rates as well as under overload.



(a) Gate Size's Effect



(b) Time-Slot Size's Effect

Figure 6.7: Gate & Time Slot Size Effect

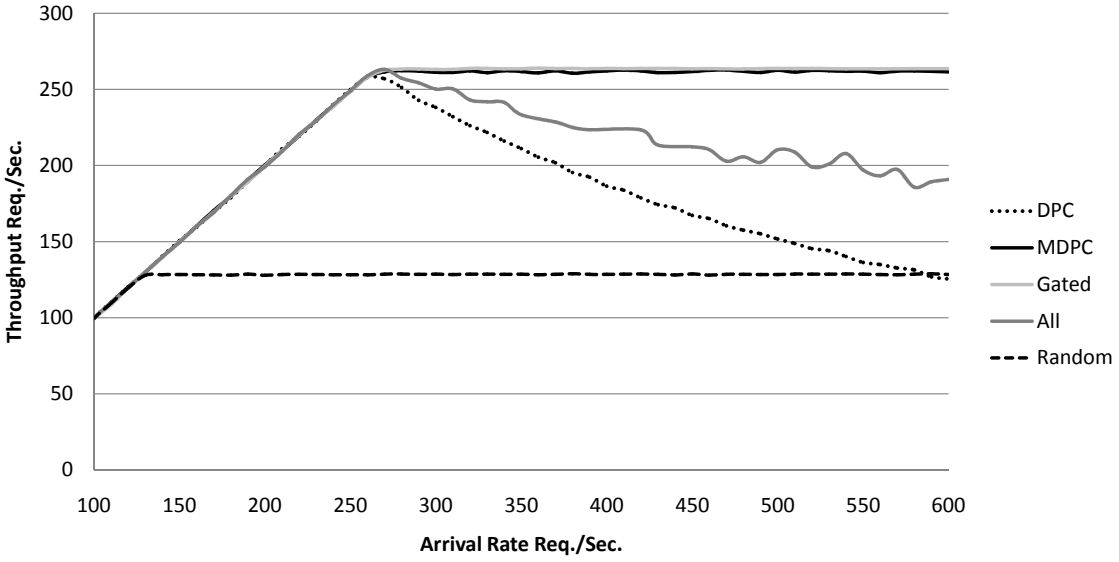


Figure 6.8: System Throughput vs. Requests Arrival Rate

We can see that our proposed allocation policy "MDPC"[4] can benefit from cache locality and can achieve almost the same peak throughput as the optimal approach "All" while avoiding performance degradation in cases of overload, because it limits the number of requests that are processed at the first stage to avoid over-committing the system. The "All" approach suffers from performance degradation when the requests arrival rate increases over the saturation point of the system, because the batch size increases dramatically leading to increasing end-to-end times for requests. The "Gated" approach behaves similar to our approach, but a limitation of this approach is the fixed batch size (the gate size g), while in our approach the batch size changes automatically, according to the system characteristics. The proposed approach without the modification for overload protection, called "DPC" in the Figure, suffers as expected from performance degradation under overload.

Figure 6.9 shows that the average response time using our approach is

[4]MDPC stands for Modified Demand/Performance Control. We give our approach this name since it depends on both the demand on the stage and its performance to allocate resources. The word "Modified" is added here because of the modification in the algorithm that we made for overload protection – see Section 6.2.2.

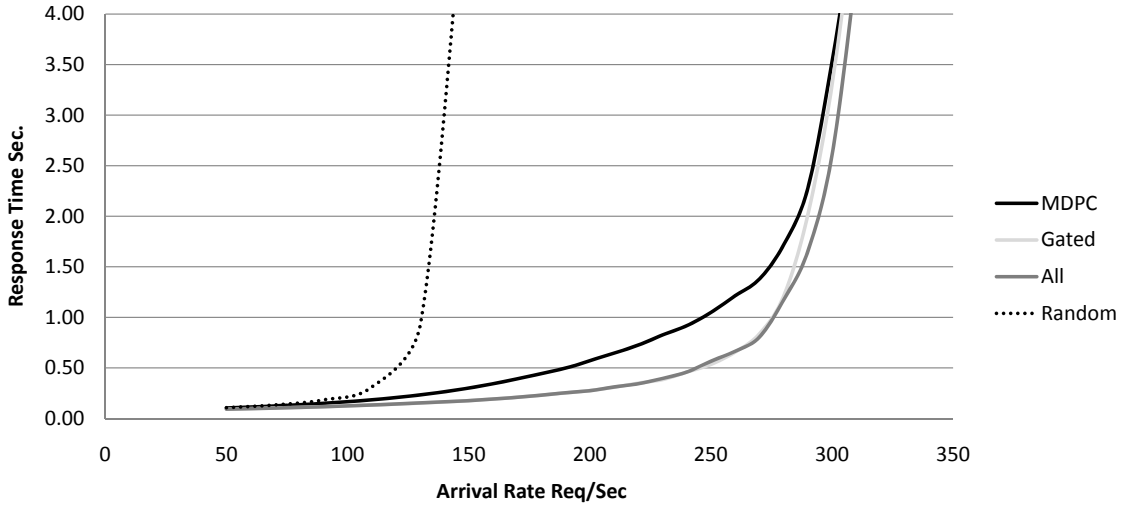


Figure 6.9: Average Response Time

slightly larger compared to "All" and "Gated". This increase is an expected result of accepting additional requests at each time slot. However, that results in having requests to be served at each stage rather than only a front-wave processing which is used in "All" and "Gated" and could cause negative effects on the utilization of other resources, like I/O system. Furthermore, we are able to control this increase in response time by adjusting the value of the used time slot size T within the performance controller.

2. Dynamic Changes Effect:
   Figure 6.10 shows the system throughput under dynamic load. In this experiment the system runs for 20 seconds with a fixed arrival rate, then this arrival rate is doubled for 10 seconds before returning to its original value. We can see that the throughput of the "DPC" approach first decreases as a result to giving the majority of CPU time to the first stage (as expected), but after returning the request arrival rate to its original value the throughput increases because the last stages have more requests in their queue and consequently get more CPU time. In contrast, the performance of "MDPC" increases because the system can process more requests in each time slot. The "Gated" approach is similar to our approach but as previously mentioned the improvement of the system throughput under this approach is a subject of the gate size.

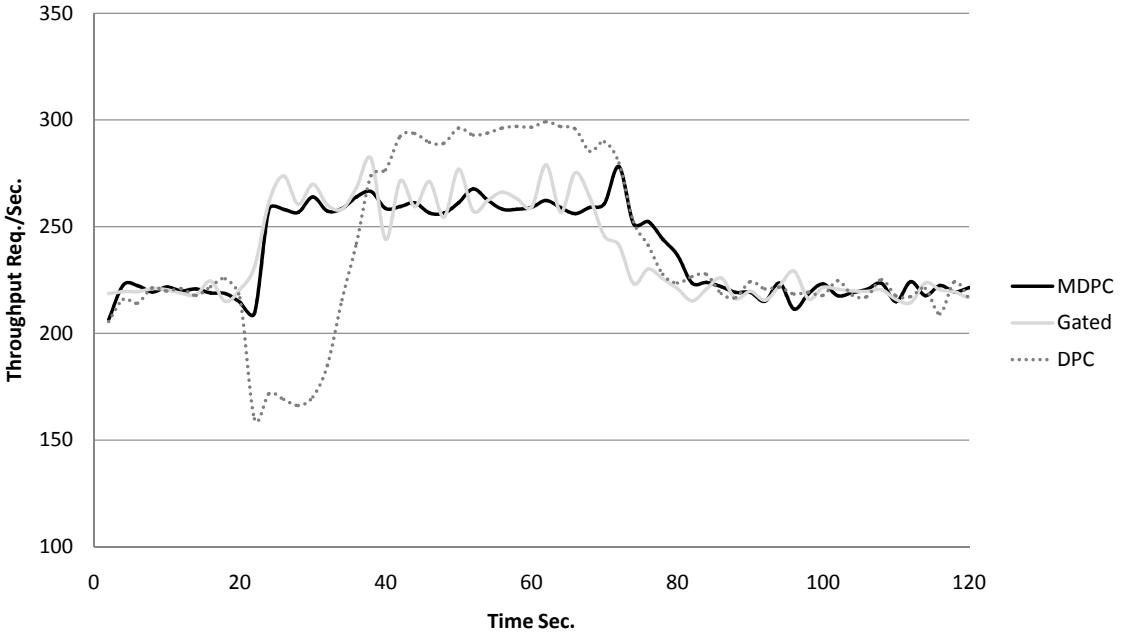


Figure 6.10: System Throughput under Dynamic Load

In this experiment the "All" approach shows large oscillations in throughput as can be seen in Figure 6.11. As the requests arrival rate increases, the batch size increases gradually and when the arrival rate returns to its original value, the batch size decreases also gradually. As a result, the observed system throughput depends on the elapsed time between observations (in our experiment, we measured the throughput in 2 seconds intervals).

Figure 6.12 shows the system performance under another form of dynamic load. In this experiment, we show the effect of load spikes on the performance of the different approaches. After running the system for 20 seconds, a large number of requests arrive at the system simultaneously. We can see that the throughput of "All" sharply decreases and then sharply increases because of the very big batch size. "DPC" shows a similar behavior but the decrease in throughput is less and more controlled because of using the size of the time slot T to limit the batch size. "MDPC" and "Gated" show robust throughput as previously. Their throughput increases to the maximum and then decreases after finishing processing

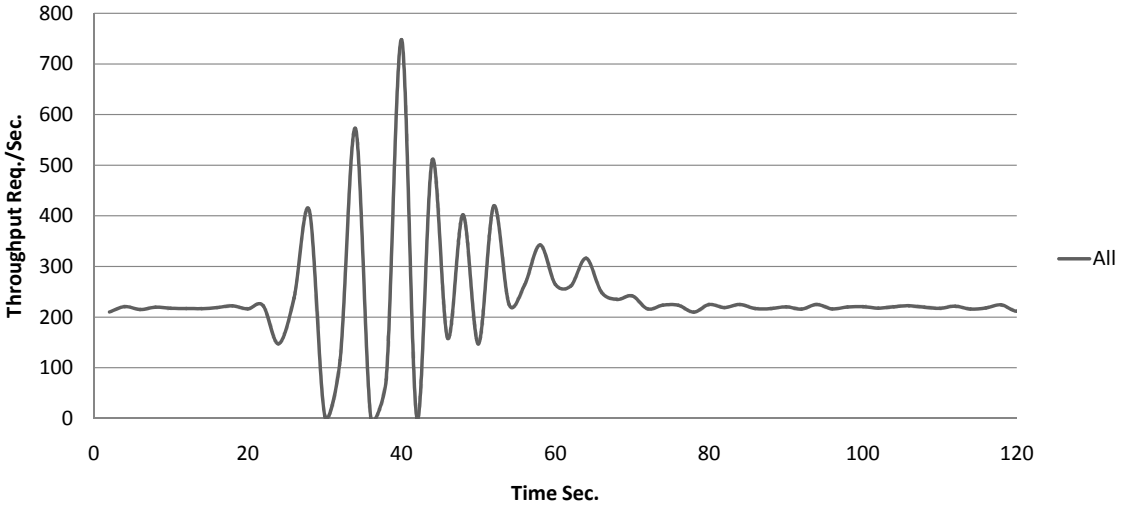


Figure 6.11: "All" Throughput under Dynamic Load

the requests which are still in the queue as a result of the load spike.

The previous experiments show the effect of changes of requests arrival rate in the system. We continue to investigate the performance of "MDPC" and "Gated" approaches considering another dynamic effect as depicted in Figure 6.13. Here the ratio of requests that go to the stages (C) and (D) changes dynamically and therefore influences the demand on the bottleneck stage (C). After running the system for 20 seconds, the ratio of requests that go to the stage (C) decreases from 50% to 5% (that means, 5% of all requests enter the system go to (C) while 95% go to (D)) for 30 seconds and then this ratio goes back to its original value of 50%. After running the system for another 50 seconds the ratio increases to 95% again for 30 seconds before it returns to 50%. We can see that the system throughput for both approaches increases with decreasing number of bottleneck requests. However, we can also see that our approach "MDPC" can benefit from this change slightly better than the "Gated" approach because of the ability to adjust batch sizes dynamically which results in processing more requests. The same effect is visible for increasing the number of bottleneck request. While both approaches show decreasing throughput, the impact on "MDPC" is less than on "Gated".

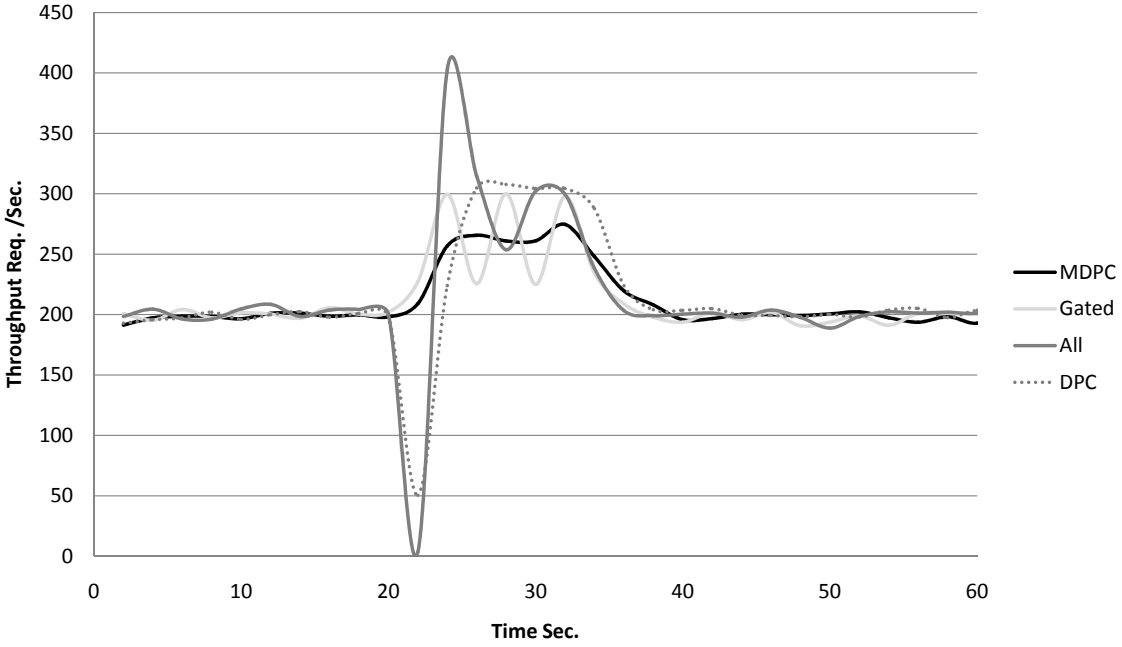


Figure 6.12: Load Spike Effect

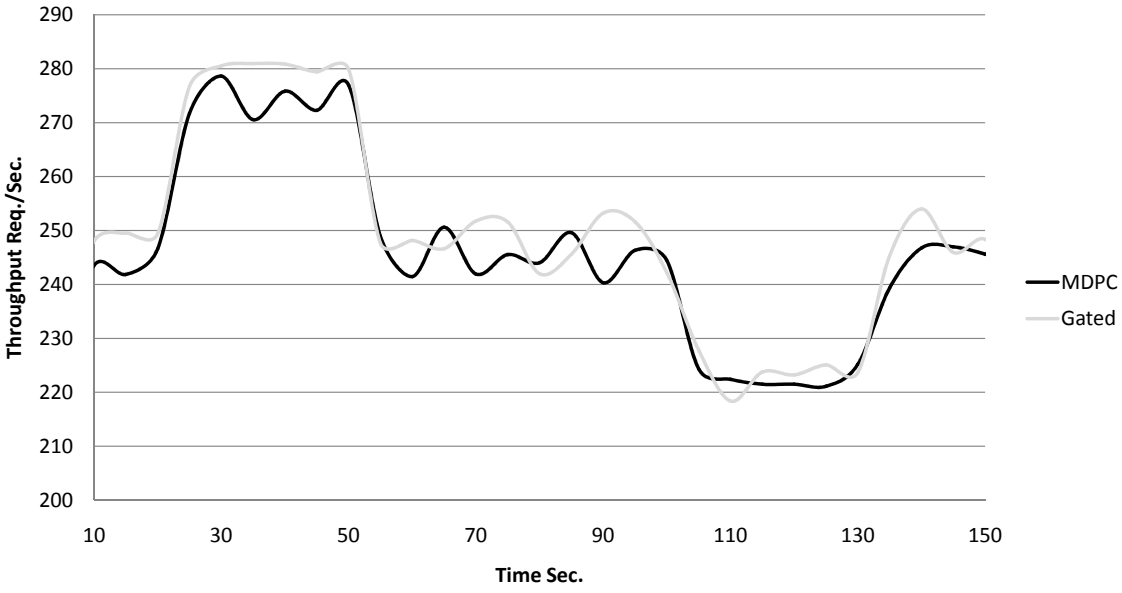


Figure 6.13: Bottleneck Requests Effect

6.3.2.3 Model Parameters Effects

Returning to the case of multiple processing units, in the following experiments, we use the model to compare our approach to a centralized run queue approach, a static allocation approach that depends on off-line optimization and to the wave-front cohort scheduling approach "Gated". Here we are interested in the effect of the model parameters on the performance of the proposed scheduling approach and on the other compared approaches.

The centralized run queue approach imitates the case of standard event driven model and ignores cache affinity. An idle processing unit processes the first request in a shared run queue regardless of to which stage it belongs. In the "Gated" approach, the processing units in the system are allocated to a stage, process a maximum of g (gate size) requests from its queue before continuing to process requests from other stages in a wave-front fashion. The off-line optimization based approach, allocates the processing units statically to the stages depending on the expected requirements of the stages during execution.

In the presented results, the simulated server machine has 16 processing units organized as eight dual core chips. The mean processing times, which are needed by a request at the different stages, are like in the previous experiments: 1 ms for stages A, B and D; 2 ms for stage C and 0.5 ms for stage E. $\forall i;\ l_i = e_i$, except the first experiment in which we vary this value. T is equal to 100 ms and g is equal to 150.

As we allocate processing units in these experiments at chip level, we set ($\forall i, j;\ OHLV_{ij} = 0$), except in the last experiment in which ($\forall i;\ OHLV_{ii} = 1\%.e_i$) and ($\forall i, j,\ i \neq j;\ OHLV_{ij} = 0$). The values of ($OHRV_{ij}$) will be specified for each experiment in what follows.

Like in the previous section, the presented results here are also the average of multiple simulation runs with relative confidence intervals of 10% and a confidence level of more than 90%.

1. Stage's Working-Set Size Effect:
   In this experiment, we only consider the overhead of loading the stage's working set into the cache when it is allocated a new processing unit to be executed on it, and we ignore the overheads of parallel processing. (i.e.,

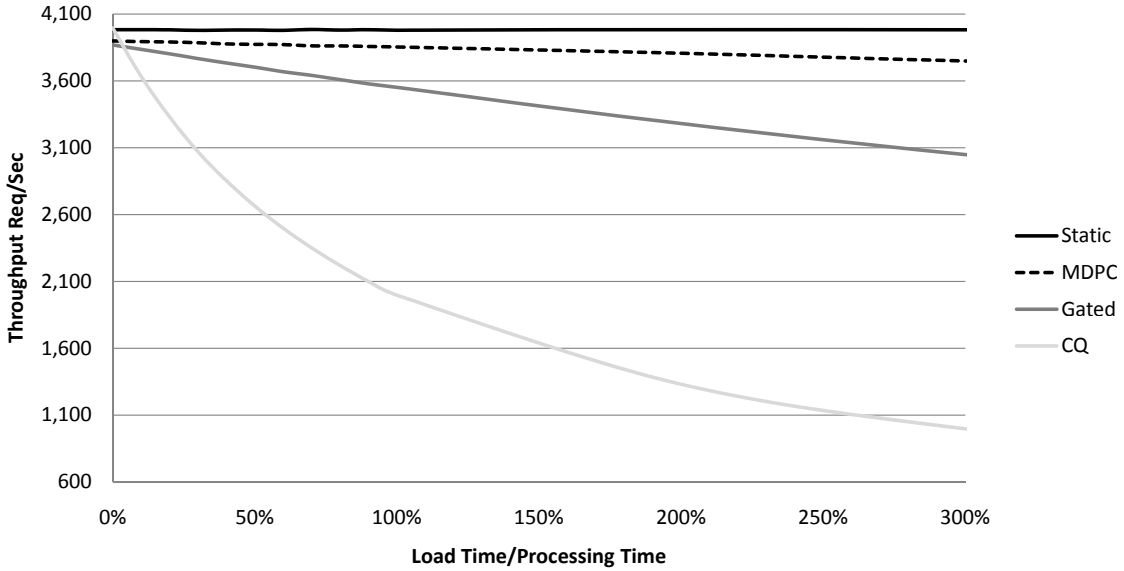


Figure 6.14: Stage Load Time Effect

$l_i > 0$, $\forall i, j; OHLV_{ij} = 0$ and $\forall i, j; OHRV_{ij} = 0$).

Figure 6.14 shows that as the load time of the stage working set increases in comparison to the average request processing time, the performance of the centralized queue approach (CQ in the Figure) which ignore locality (while favoring load balancing) decreases dramatically. In contrast, the other three approaches that take data locality in account can benefit from this locality to increase the system throughput. The figure shows that the performance of the static allocation approach is almost not affected as the load time increases, because the stage's working set is loaded into the cache of the processing units allocated to this stage only once at the beginning. When we compare our approach (MDPC in the Figure) to the Gated approach, we can see that since our approach causes less loads of a stage's working set per batch its performance degrades less than the performance of the Gated approach as the load time increases.

A similar effect of the stage's working-set size can be seen if we fix the size of the working set and change the number of processing units in the system. In Figure 6.15 we set $\forall i;\ l_i = e_i$. The figure shows that as the number of processing units increases both the Static approach and the

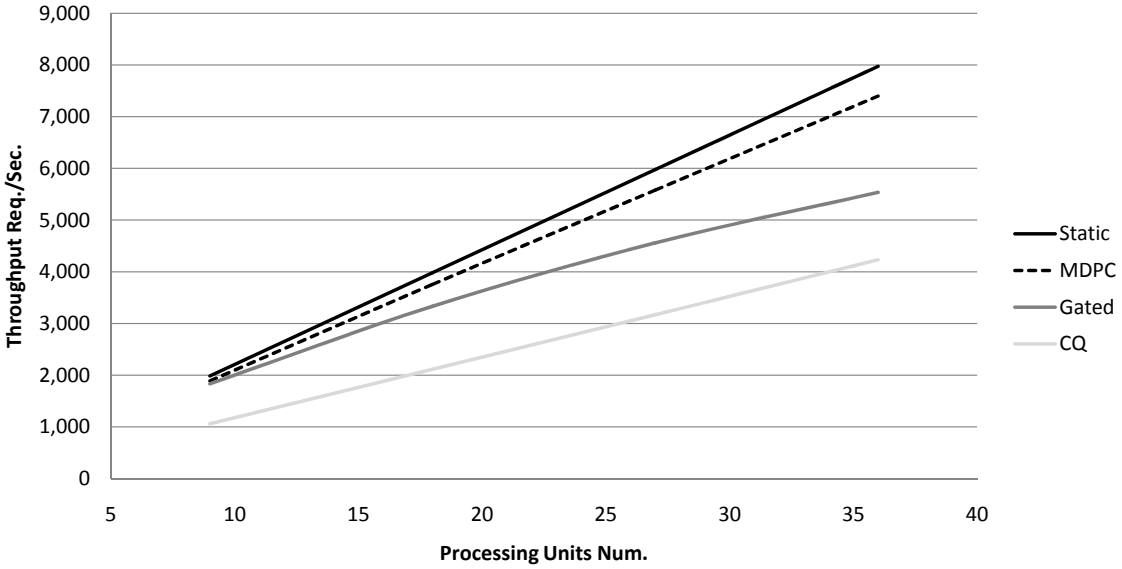


Figure 6.15: Processing Units Number Effect

MDPC approach can benefit more from this increase as they cause less increase in the number of cache loads.

2. Overhead of Concurrent Requests From the Same Stage:
   In this experiment, in addition to the stage's working set load time, exists an overhead resulting in from processing multiple requests from the same stage in parallel on different chips, (i.e., $l_i > 0$, $OHRV_{ii} > 0$ and $\forall i, j,\ i \neq j; OHRV_{ij} = 0$).

   Figure 6.16 shows that the performance of the centralized queue approach, which ignores data locality, outperforms the performance of the Gated approach as the overhead increases in comparison to the average request processing time. In the Gated approach, the processing units in the system are allocated to a single stage. This increases the number of requests processed from the same stage in parallel, which increases the total processing time needed per request. MDPC and Static favor allocating less processing units for a longer time to a stage, and for this reason the benefit from locality still not over-committed by parallel processing overhead.

3. Overhead of Concurrent Requests From Other Stages:
   In this experiment, one more overhead is considered, that is the overhead

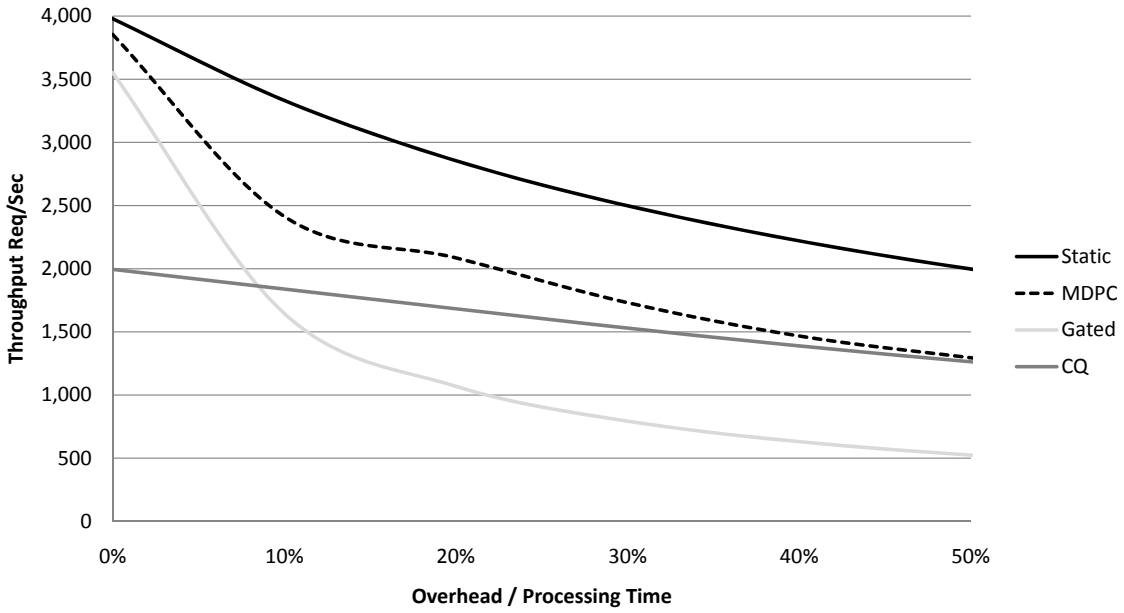


Figure 6.16: Parallel Processing Overhead Effect of Same Stage Requests

resulting in from processing requests from other stages simultaneously on the other chips. Although this overhead should be small in the staged architecture, as stages are designed to have their private code and data structures, it is possible that this overhead still exists because of shared global variables across stages and contention for other resources.

In this experiment $l_i > 0$, $\forall i, j$; $OHRV_{ij} > 0$, so that $\forall i; OHRV_{ii} = 10\% \times e_i$, and we change the value of $OHRV_{ij}$; $i \neq j$. The experiment shows how the performance of the different policies affected as the relation between the two values changed – the value of the overhead of concurrent requests from the same stage and the value of the overhead of concurrent requests from other stages.

Figure 6.17 shows the result of this experiment. We can see that the performance of the three approaches, Static, Gated and MDPC, degrades as ($OHRV_{ij}$; $i \neq j$) increases in comparison to ($OHRV_{ii}$). However, the performance of the Gated approach degrades less than the other two approaches and the performance of the Static approach degrades even more than our approach. When the overhead of concurrent requests from

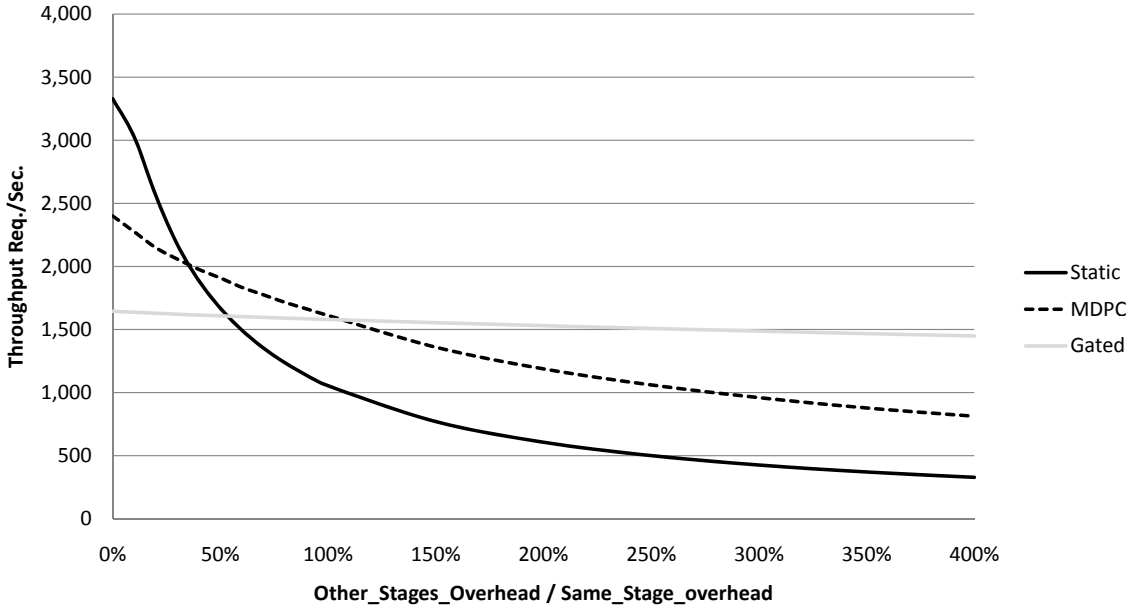


Figure 6.17: Parallel Processing Overhead Effect of Requests from Other Stages

other stages is almost equal to the overhead of concurrent requests from the same stage (i.e. $OHRV_{ij} = OHRV_{ii}$), our approach and the Gated approach show almost the same performance, and after this point the Gated approach show the best performance.

The reason of this behavior is that the Gated approach allocates all the processing units to process the requests from a single stage and a processing unit is moved to process requests from other stages only when processing the batch of the current stage has been finished. As a result the overhead of concurrent requests from other stages is the minimum. In contrast, this overhead is the maximum in the case of the Static approach.

4. Dynamic Changes Effect:
   In this experiment, the characteristics of the workload on the system change dynamically during the execution, so that the proportion of bottleneck requests (requests that go to the bottleneck stage (C) in Figure 6.2b) changes (increases in Figure 6.18a and decreases in Figure 6.18b) after running the system for thirty seconds and returns to its original value after thirty more seconds. The parallel processing overheads are ignored,

i.e., $\forall i, j;\ OHRV_{ij} = 0$ and $OHLV_{ij} = 0$.

In Figure 6.18a, we begin with 50% bottleneck requests that increase to 95%. The figure shows that the performance of all three approaches decreases as the number of bottleneck requests increases. However, this causes more decrease in the performance of the Static approach. The reason of this behavior, in addition to the extra processing time needed to serve more bottleneck requests, is that this approach does not change the resource allocation to the stages dynamically which wastes the idle time of the processing units that are allocated to stage (D) while stage (C) needs more processing power. That is not the case in the other two approaches which balance the allocation at run time.

Similarly, in figure 6.18b we begin with 50% bottleneck requests that decrease to only 5%. The MDPC approach and the Gated approach can benefit from the surplus processing power to process more requests and as a result increase the system throughput. The Static approach, like in the previous experiment, do not re-allocate idle time of processing resources that are allocated to stage (C) to stage (D) which have more requests now to be processed.

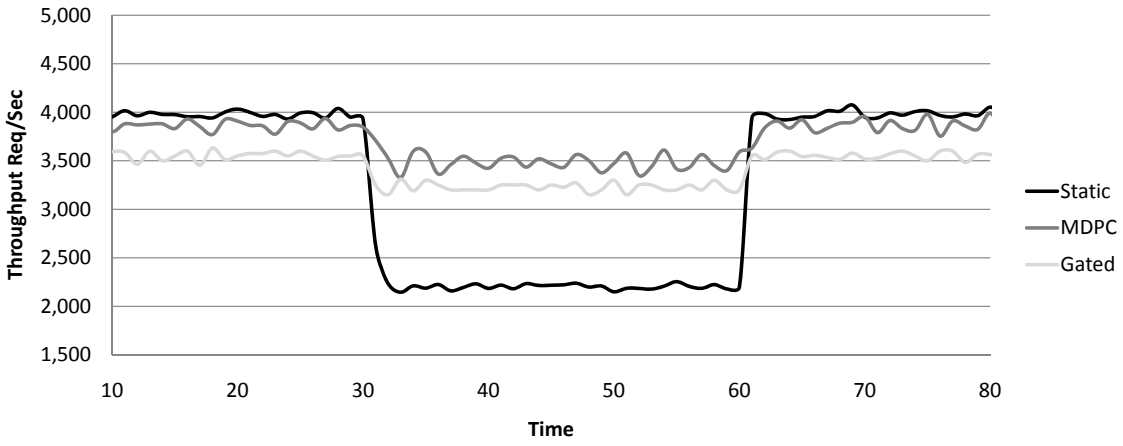
6.3.2.4 Parallelism hierarchy effect

In this experiment, we investigate how the performance of our approach is affected by the processing units hierarchy. We consider here that

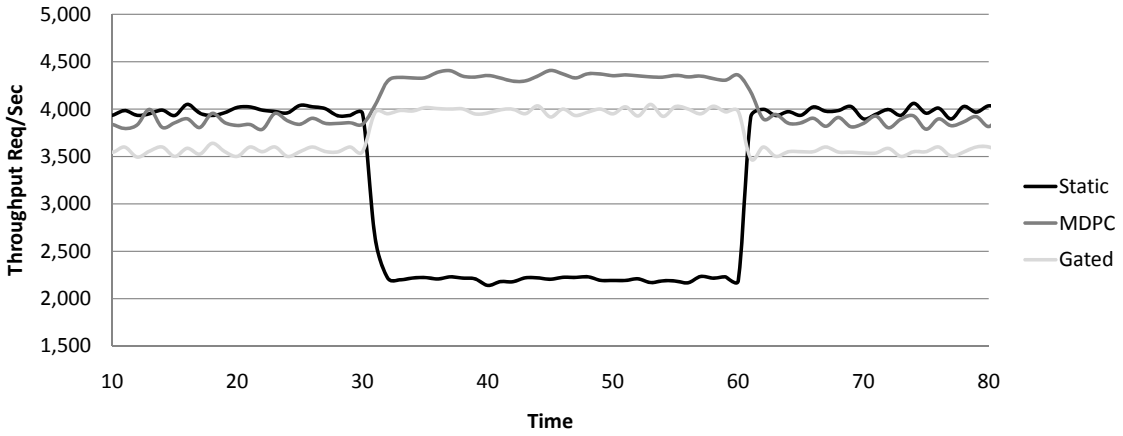
$$\forall i;\ l_i = e_i,\ OHRV_{ii} = 10\%.e_i,\ OHLV_{ii} = 1\%.e_i$$

and all other overheads are equal to zero.

The number of simulated processing units is always 16 processing units, but we change the organization of these processing units as 16 single-core chips, 8 dual-core chips, 4 quad-core chips, 2 chips with eight cores, and a single chip with 16 cores, and see how the performance of the system is affected by these changes. In all organizations caches are as previously – a private cache per core and a shared cache per chip.



(a) Increasing Bottleneck Requests



(b) Decreasing Bottleneck Requests

Figure 6.18: Dynamic Behavior

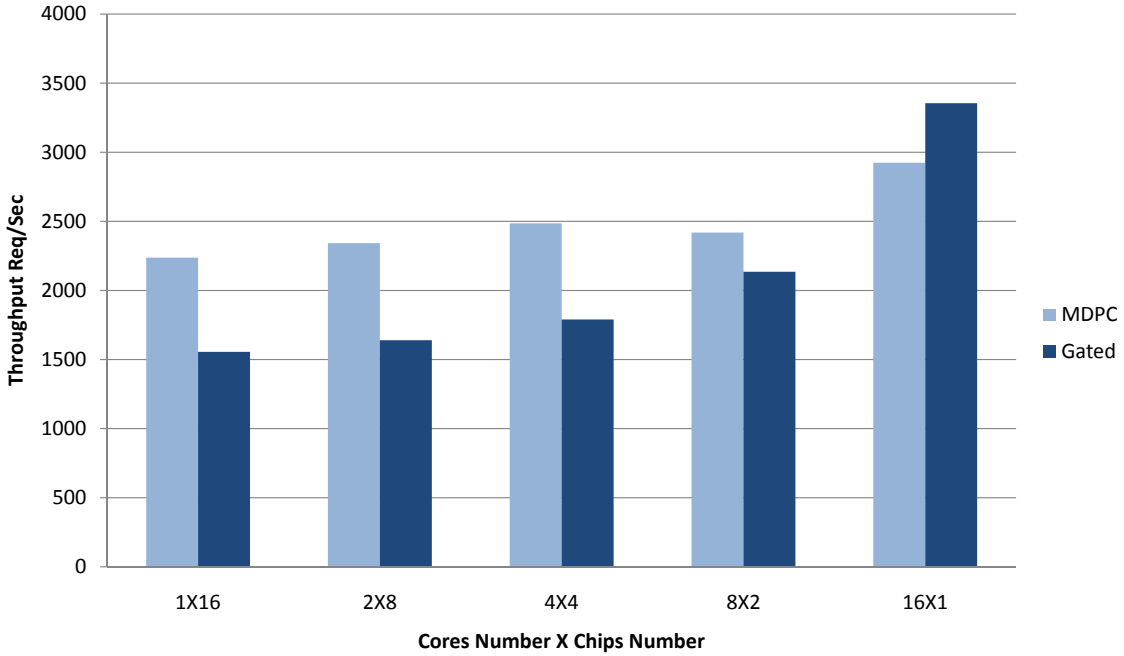


Figure 6.19: Parallelism Hierarchy Effect

From Figure 6.19, we can see that the performance of our approach increases as the number of cores per chip increases. This behavior is a result of avoiding the more expensive overheads resulting from "remote" processing of requests from the same stage on other chips. In the case of a single chip with 16 cores, our approach and the Gated approach are almost similar as they both allocate all the processing units to a single stage before moving to process other stages, however, in the experiment we can see that the performance of the Gated approach outperforms our approach; the reason is that in our approach the cores of the chip wait until all the requests of the batch from the stage are processed before being allocated to other stages. We do that to avoid the contention of the stages for the shared cache level, which can impact the performance even more, depending on the needed processing time per request and the stage's working sets sizes. The Gated approach ignores shared cache contention, and allocates idle cores immediately to the next stage. As the overhead of this behavior is not simulated in our model, for this reason, the Gated approach shows better performance.

The impact of our trade off between allocating idle cores to other stages and

avoiding shared cache contention can be seen if we compare the performance of the case of 4 cores per chip with the case of 8 cores per chip; here we can see that the performance of our approach in the 4 cores case is slightly better than the performance of the 8 cores case since in the later case more cores can stay idle before being allocated to other stages; however even with the impact of this idle time the performance is still better than the performance of the Gated approach.

6.4 Discussion

This chapter has introduced an adaptive scheduling policy to allocate processing resources in Internet services that are based on the staged event-driven architecture (SEDA). Experimental results have demonstrated that the proposed policy leads to a better performance under the investigated system characteristics while avoiding performance degradation under overload and dynamic changes in the system, by adapting the resources allocated to each stage based on observations of the changes in the stages load and performance.

It has been shown that our approach can benefit from locality within stages to increase system throughput and it can decrease the effect of parallel processing overheads that degrade the performance of existing cohort scheduling approaches dramatically. In contrast to the approaches that depend on offline optimization, our approach can automatically adapt resource allocation and sustain high performance even in the case of dynamic changes in the system, as it adjusts resource allocation among the different stages on the basis of short-term demand estimates. The experiments have also shown that our approach can benefit from the chip multi-processing trend in today's microprocessor technology, which tends to increase the on chip parallelism (e.g. the number of cores per chip).

The proposed approach allocates the processing units to the stages by scheduling the requests in the stage's queue to be processed as batches, which is a policy that provides many benefits. Firstly, batches present a compromise between a shared or centralized run queue (among multiple processing units), which promotes load balancing aims but decreases cache efficiency, and between traditional cache affinity schedulers which utilize per-processor run

queue. Synchronization overheads can also become an issue in large scale multi-processors systems. A second benefit of scheduling requests as batches rather than individual requests is that it can reduce these overheads since it reduces the contention for the shared run queue [51]. Moreover, in addition to increase the benefit from locality within a stage, batching the execution of multiple requests from a stage on a processing unit decreases the number of requests from the same stage that are processed simultaneously on other processing units and as a result decreases the memory accesses across the processing units which cause additional processing overheads [156].

In our experiments we have considered that the stages are based on the event-driven programming model, and they use a single execution thread per processing unit. However, even in the case of multi-threaded stages, processing a batch of requests from the same stage at a processing unit can benefit performance as it causes locating sharing threads onto the same processing unit. As a result these threads incidentally perform prefetching of shared regions for each other, and they help to obtain and maintain frequently used shared regions in the processing unit local caches [156].

As stated earlier, the presented resource allocation policy is a part of a three-layers control architecture that represent the global controller – see Chapter 5. This global controller depends on a local controller that exists in each of the different stages. The local controllers collect, analyze, and report a variety of metrics at a per stage level. The global controller interacts with these local controllers to get these information, which are required to provide the global resource management. In the case of our global controller these information contain an estimation of the stage's throughput and the size of the stage's queue. As a result of this controllers interaction, the accuracy of the local controllers affects the efficiency of the global controller. For example, in our experiments local controller have used exponential averaging to estimate the the throughput of the individual stage. The accuracy of these estimation affect the portions of processing time that are allocated to each stage. However, a benefit of our approach is that it depends also on the number of requests in the stages' queues, and as a result it can improve the allocations in the subsequent time slots even if the local controllers fail to provide accurate estimation of the stages' performance.

7 Adaptive Performance Control for Staged Services

Although the Staged design emerges as a programming paradigm to implement high performance Internet services that support massively concurrent demands, it presents many challenges when the system is to be tuned to achieve a desired performance target or to guarantee a determined service level. This Chapter demonstrates an adaptive control approach that automatically manages system resources in order to control the performance for staged applications. This approach builds on the three-layers control architecture (Chapter 5) and is based on a combination of the adaptive resource allocation policy presented in the previous chapter (Chapter 6) and a feedback based performance controller.

7.1 Performance Control Challenges

As discussed throughout previous chapters, the staged design has emerged as a programming paradigm to implement high performance Internet services that support massively concurrent demands. This design avoids many pitfalls related to the conventional concurrency models, and presents system design advantages at hardware- and software engineering level. However, the staged design introduces also many challenges when the system is to be tuned to achieve a desired performance target or to guarantee a determined service level.

In staged design-based Internet services quality guarantees for requests processing are difficult to maintain for many reasons. First, requests are continually generated in large volumes by external and internal sources such as clients, stages and I/O operations. Second, the processing costs of requests vary dynamically along the time and are difficult to predict. The last, fluctuations of loads and resources usage of the individual stages may cause over-

loading that interferes with the robustness of the system or may under-utilize system resources. To deal with these challenges and such instability, systems that are based on this design model need to be automatically adjusted at run time in order to be able to achieve target performance levels.

Existing approaches are based on managing the available resources depending on an off-line knowledge of the system characteristics to achieve high performance [108, 110, 172]. Configuring such systems to generate the desired performance requires experienced administrators to correctly set multiple relevant control parameters for multiple stages, or to determine these parameters experimentally using benchmarks. These management techniques require tedious manual operations which are difficult, time-consuming, error-prone and non-QoS-guaranteed, especially when they are to be implemented in such highly dynamic computing systems, like Internet services. In addition, these configurations are not based on any mathematical relationships between the controlled parameters and the target performance. As an optimal configuration usually depends on an administrator's good guess, therefore, parameters configuration can easily result in over utilization or under utilization of the available system resources.

Request processing delay, i.e, the time it takes on the server to serve a client's request, is probably the most visible performance metric and the most critical quality from the perspective of most users. Thus, in this thesis, we consider guarantees on this request processing delay or what we can call "Sever-side response time". With respect to a request, processing delay is defined as the time elapsed since the request arrives at the network queue of the service until it leaves the system. The delay seen by a client includes also the time a request spends in the network in addition to the time it spends on the server. However, as research in the networking community addressed the problem of bounding network delays, our approach addresses the complementary problem of bounding the delay on the server-side.

The consideration here is that the system administrator is allowed to specify a target sever response time (TRT), and the goal is to maintain the average processing delay of requests to be under this target. This specification can be considered as a part of a service level agreement (SLA) which is a contract

between a service provider and its customers. Such an agreement consists of one or more service-level objectives (SLOs). An SLO has three parts: the metric (e.g., average response time), the bound (TRT seconds), and a relational operator (less than) [81]. Service providers intuitively want to have sufficient resources to meet their SLOs, but they do not want to have more resources than required since doing so imposes unnecessary costs. However, a fact that is to be addressed too is that these objectives must be achieved in the presence of time-varying loads and changes in hardware and software configurations.

7.2 The Proposed Performance Management Approach

Our contribution is to present an adaptive control approach that automatically manages resources and adapts the system in dynamic working environments in order to control the performance for staged design-based Internet services. Our approach builds on the three-layers control architecture (Chapter 5) and is based on a combination of the dynamic resource allocation policy presented in the previous chapter (Chapter 6) and a feedback based performance controller.

7.2.1 The Proposed Approach

In order to guarantee that the average response time is less than a given performance target, both the rate of accepted requests and the departure rate of the responses need to be balanced, so that the queuing time would not be so long as to degrade the performance. We build on the proposed three-layers control architecture (Chapter 5) and on the presented scheduling policy (Chapter 6) to adjust both the system throughput and the rate of accepted requests, in order to control the performance for staged design-based Internet services. A property of our scheduling policy is that it presents the value of the time slot size T as a key parameter to control a variety of performance metrics. The value of T affects the system throughput, the individual requests response time, resources utilization, etc. As we here focus on the server response time as the performance metric we depend on this property to introduce an approach which guarantees that the average response time of the individual requests is under the given target, while sustain a high system throughput.

Consider that we have a target response time TRT. Requests that arrive at the server are accepted to be serviced if the server is able to guarantee that

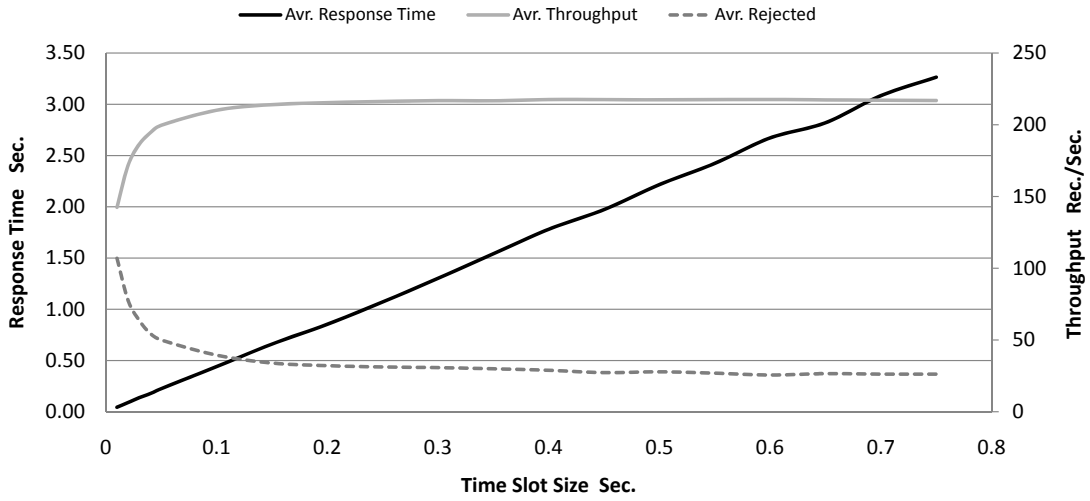


Figure 7.1: Time Slot Size vs. Performance Metrics

the average response time of the accepted requests is under TRT, else they will be rejected [1]. The idea of our control approach is to use the advantages of the adaptive scheduling policy presented in the previous chapter to optimize the allocation of system resources at run-time, and to reject these additional requests. At each time slot, only the requests that are not possible to be processed within the time slot are rejected, so that the system can provide the required quality of service to the maximum number of requests.

As explained in Section 6.2.2, the Equation 6.9 gives the processing times that are to be allocated to the individual stages at each time slot, except the entrance stage. The rest of the available processing time that remains after allocating these times to the stages is to be allocated to the entrance stage in order to accept requests in the system and to process requests from the arrival queue. This remaining time can be calculated as[2]:

$$t_1(k+1) = T - \sum_{2 \leq i \leq M} t_i(k+1) \tag{7.1}$$

Based on this value ($t_1(k+1)$) the number of accepted and rejected requests at this time slot can be determined. If ($t_1(k+1) \leq 0$) then all the requests in the queue of the entrance stage (the requests that are waiting to be accepted in

[1] Other policies are also possible like service degradation or forwarding to other servers.
[2] In the case of multiple processing units, the value T in this relation must be multiplied by the number of processing units – See Section 6.2.3.

the system) have to be rejected in order to guarantee the desired performance level for the requests that are currently existed in the system. Elsewhere, if ($t_1(k+1) > 0$) we only accept as much requests at the first stage as this stage can process in this processing time ($t_1(k+1)$). That means we accept ($t_1(k+1).th_1(k)$) requests and reject the remaining requests in the queue.

Using this technique makes the average response time of the accepted requests proportional to the value of the time slot size. Increasing the value of T will increase the number of accepted requests that will be processed as a batch, as a result this will increase the response time of the accepted requests since they will wait for other requests in the batch to be processed before being processed at the following stages. At the same time increasing T is potential to increase the system throughput as a result of increasing the benefit from locality within the stages as the batch size increases. Figure 7.1 shows the linear relation between the average response time and the time slot size T – how the average response time increases as T increases and vice verse. It shows also how the system throughput increases as T increases. Based on this relation we can implement a feedback-based controller that adjusts the size of the time slot size to keep the average response time for the accepted requests under the desired target TRT.

A simple heuristic controller that manipulates the time slot size in order to trace the average response time target, based on observations of the system behavior, can easily achieve the control targets [10]. However, a control theoretic-based approach can adapt the system in a more reliable manner. The following section explains the implementation of such a feedback controller.

7.2.2 Feedback Control

Feedback control is a technique that has been widely implemented in dynamic environments to solve the performance control problem under unpredictable behavior. Since the early 1990s, there has been broad interest in the application of this control technique to computing systems [81]. The main idea of feedback control is to use measurements of a system's outputs, such as response time, throughput, and resource utilization, to achieve externally specified goals. This is done by adjusting the system control inputs, such as parameters that affect scheduling policies, concurrency levels or other system

parameters. Since the measured outputs are used to determine the control inputs, and the control inputs then affect the outputs, this control architecture is called feedback or closed loop control.

Today, feedback control techniques offer a robust solution to the server performance control problem and a promising way of achieving desired performance in emerging critical Internet applications. However, there are several areas of challenges in applying feedback control to computing systems, like constructing models of the target systems and controllers, designing feedback controllers and developing evaluation criteria for these controllers.

7.2.2.1 System Model

Considering the discussion in the previous section – Section 7.2.1, we can see that implementing the proposed scheduling policy in staged design-based services presents new parameters in the system that simplify the reasoning about its performance and make the system amenable to feedback control. Using this policy, the different performance metrics of the staged service are related to the value of the time slot size and affected by this value. However, designing a feedback control based system requires an ability to quantify the effect of control inputs on measured outputs, both of which may vary with time. As previously mentioned, here we are interested in the average end-to-end response time, from entry into the first queue in the staged service until exiting the system, as the performance metric to be controlled. For this reason, and in order to implement a feedback controller that achieves timing guarantees, we need to quantify the effect of the time slot size (T) (the control input in our system) on the average response time (the measured output that we want to control).

In general, constructing first principles models that quantify such effects for complex computing systems may be extremely difficult, as considerable sophistication, a detailed knowledge how the system operates and a detailed knowledge of the relationships between the different system parameters is required to do so. The complex structure of staged design based services, the effects of server-hardware architecture and the characteristics of system's workloads increase this difficulty. For this reason, we depend on statistical system

identification techniques to construct a black-box model of the target system[3]. In control theory, the term black-box model is used since only the inputs and outputs of the target system are needed to be known. We employ a first order ARX model to describe the relationship between the time slot size (T) and the measured average response time (RT). ARX models are difference equations that relate linear functions of outputs history (current and past outputs) to linear functions of inputs history (current and past inputs). Equation 7.2 shows the general equation of the deployed ARX model:

$$RT(k) = a\,RT(k-1) + b\,\triangle T(k) \tag{7.2}$$

In this equation $RT(k)$ and $\triangle T(k)$ denote the average response time and the change in the time slot size in the kth sample period respectively; a and b are system specific scalars which are obtained by system identification and are referred to as the model parameters [112]. In this model the output RT depends only on the input $\triangle T$ values and the output RT values from one time unit in the past, for this reason it is called a first-order model. The equation is also an example of a single-input single-output model (SISO Model), since there is only one input $\triangle T$ and one output RT.

The equation indicates that whenever the time slot size is varied, the average response time would be changed immediately in the same sample time without any delays, and this change in average response time is linear related to the changing of the time slot size in this sampling period.

Selecting this model is motivated by the results of our experiments which show a linear relationship between the average response time and the time slot size even when the system throughput saturates – see Figure 7.1. It is also motivated by the fact that such linear time varying system models work surprisingly well for many control applications, as demonstrated by many researchers [81]. For example, a similar model has been implemented to control the performance of the Apache web server [4], and to control the performance of a SEDA based HTTP server called Haboob [109]. In [4], the model has been used to represent the relationship between the utilization of server resources and the requests arrival rate, and in [109] this model has been used to represent the relationship between the number of threads in a stage's thread pool

[3]We use the term target system to refer to the staged service that is to be controlled.

and the throughput of this stage. The novelty of our approach is that we depend on a new parameter in the system, the time slot size (T), which enables us to present a model of the whole staged service, not only of one stage in the system. A common challenge in this case is the long time it takes for changes in control input to affect the system output, because of the multiple stages in the service, which is known in control theory as dead time. However, considering that the control sampling period is significantly larger than the time slot size, and considering that, according to the scheduling policy, any changes in the time slot size will immediately affect the response times of all the requests in the system at all stages, not only the requests that are accepted at the entrance stage, this will eliminate the dead time effect in our controller. We argue that, as such simple models suffice to achieve the desired control targets, there is no need to develop more complex models. In addition, using more complex models, e.g. models with more parameters, is not always better as it can result in the modeling of nonexistent dynamics.

The model parameters a and b in the Equation 7.2 are needed to be estimated in order to develop the system model that is to be deployed in our feedback controller. Least-squares techniques are usually used for such estimations and here we choose the least mean square (LMS) which is one of these techniques to estimate the values of these parameters. Based on a series of measured data, the least squares method attempts to find a function which closely approximates the data, and tries to minimize the sum of the squares of the differences between the points generated by the function and the corresponding points in the data. In our system model, LMS estimates the values of a and b in Equation 7.2 for minimizing the difference between the estimated and actual average end-to-end response time.

If the estimated average response time is denoted by $\widetilde{RT}$, then

$$\widetilde{RT}(k) = a\,RT(k-1) + b\,\triangle T(k) \tag{7.3}$$

and the kth estimation error is:

$$e(k) = RT(k) - \widetilde{RT}(k) = RT(k) - a\,RT(k-1) - b\,\triangle T(k) \tag{7.4}$$

Least mean squares technique aims to choose a and b values so as to minimize the sum of the squared errors. In other words, it is to minimize the function:

$$J(a,b) = \sum_{k=2}^{\Omega} e^2(k) \tag{7.5}$$

where Ω is the total number of elements in the measured data series.

The values of a and b that minimize $J(a,b)$ can be found by taking partial derivatives and setting them to zero –Equation 7.6 and Equation 7.7 [81].

$$\frac{\partial J(a,b)}{\partial a} = -2\sum_{k=2}^{\Omega} RT(k-1)[RT(k) - a\,RT(k-1) - b\,\triangle T(k)] = 0 \tag{7.6}$$

$$\frac{\partial J(a,b)}{\partial b} = -2\sum_{k=2}^{\Omega} \triangle T(k)[RT(k) - a\,RT(k-1) - b\,\triangle T(k)] = 0 \tag{7.7}$$

For convenience of notation, define the following quantities:

$$S_1 = \sum_{k=2}^{\Omega} RT(k-1)\,RT(k-1) \tag{7.8}$$

$$S_2 = \sum_{k=2}^{\Omega} RT(k-1)\,\triangle T(k) \tag{7.9}$$

$$S_3 = \sum_{k=2}^{\Omega} \triangle T(k)\,\triangle T(k) \tag{7.10}$$

$$S_4 = \sum_{k=2}^{\Omega} RT(k)\,RT(k-1) \tag{7.11}$$

$$S_5 = \sum_{k=2}^{\Omega} RT(k)\,\triangle T(k) \tag{7.12}$$

So we can achieve the optimal values of a and b for the system model, by manipulating the equations 7.6 and 7.7.

$$a = \frac{S_2S_5 - S_3S_4}{S_2^2 - S_1S_3} \tag{7.13}$$

Time Slot Size T (ms)	Average Response Time RT (Sec.)
10	0.044
50	0.225
100	0.440
150	0.661
200	0.853
250	1.071
300	1.303
350	1.541
400	1.781
450	1.973
500	2.217
550	2.424
600	2.672
650	2.820
700	3.085
750	3.265

Table 7.1: Data Series Elements

$$b = \frac{S_2 S_4 - S_1 S_5}{S_2^2 - S_1 S_3} \tag{7.14}$$

Table 7.1 shows the data elements that we have used to estimate the system parameters, Equation 7.15 shows the system estimation model, and Figure 7.2 shows a comparison of the actual and the estimated average response time of the implemented staged service prototype (see Section 6.3) and this system model. The figure confirms that this system model is valid and demonstrates that it is reliable to be used in the automatic control system designs. Note that the above system identification algorithm can be used in off-line or on-line modeling of the controlled system.

$$RT(k) = 0.996\, RT(k-1) + 4.464 \triangle T(k) \tag{7.15}$$

7.2.2.2 Feedback Controller Design

Based on the developed system model (Equation 7.2), here we demonstrate how to implement this model into a feedback-based controller in order to achieve the performance target and to control the system behavior.

The main components of a feedback control system form a feedback control

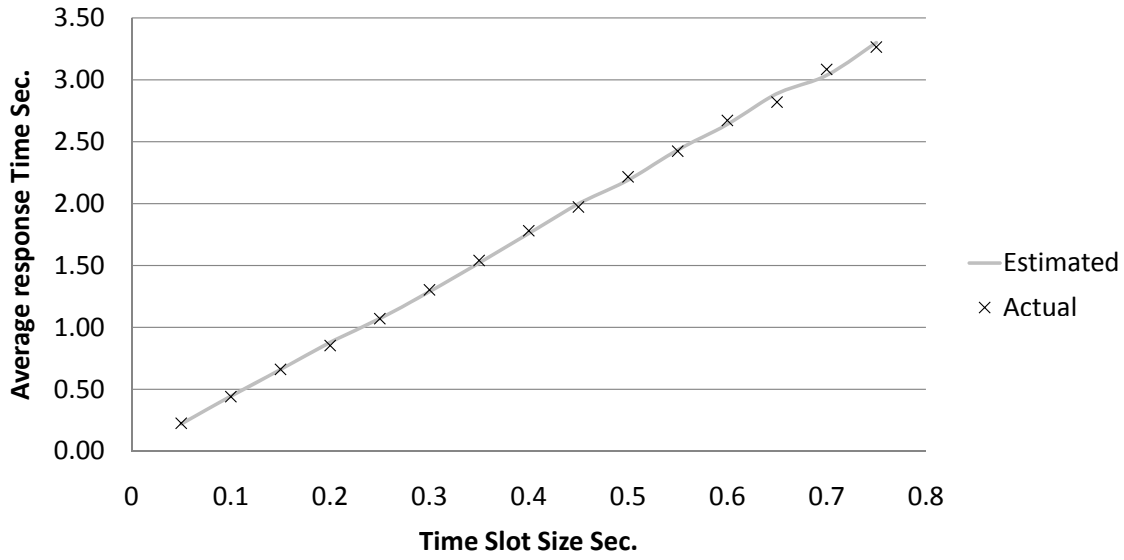


Figure 7.2: Comparison of the Estimated and the Actual Average Response Time.

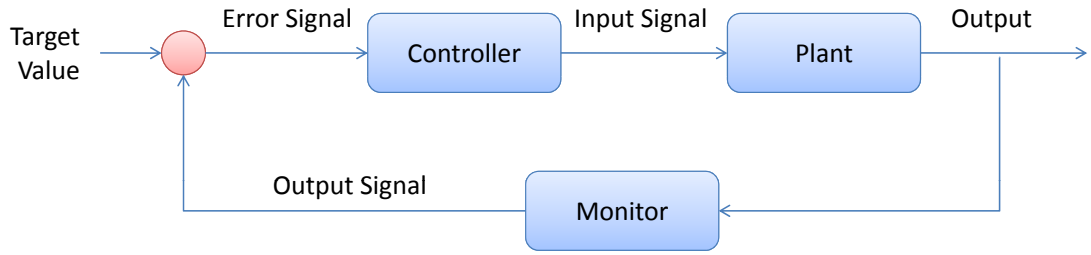


Figure 7.3: The Feedback Control Loop

loop, as shown in Figure 7.3. This system consists of: The Plant, which is the system to be controlled; Monitor, which periodically measures the status of the plant; and a Controller, which compares the current status of the plant measured by the monitor versus the desired status that is set by the administrator. The difference between the output signal and the desired value is mapped into a control input signal that adjusts the behavior of the plant accordingly. In such control systems, the goal of the control operations is to achieve the desired target, despite the effects of system and environment uncertain disturbances.

Considering our system, this general control model can be translated into a concrete model as follows. The plant to be controlled is the staged service.

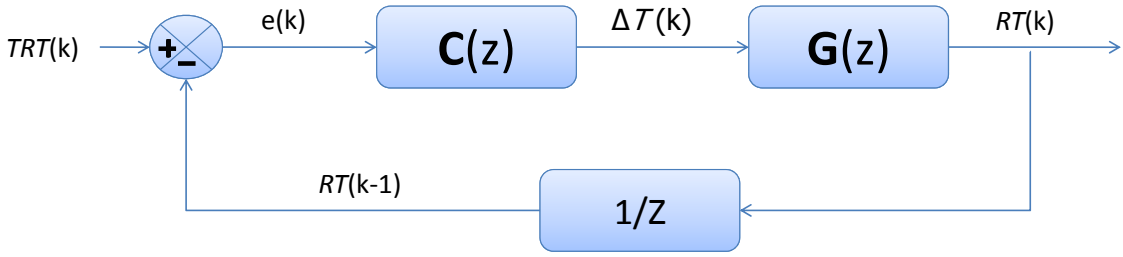


Figure 7.4: Block Diagram of The Feedback Control System

Plant status is measured by the monitor periodically and output signals are sent to the controller. We use the average response time (RT) as the output signal. The controller compares the output signal with the target average response time (TRT) and depending on the resulted error signal it adjusts the input signal, which is the change in the time slot size ($\triangle T$) in our system. The fluctuation of arrival patterns, individual stages workload and processing costs are all disturbances in the system.

Equation 7.2 presents the model of the target system (the Plant). Since our control is time-discrete, we apply the standard z-transformation on this equation in order to get its Z-domain representation [81]:

$$\mathbf{RT}(z) = \frac{bz}{z-a} \triangle \mathbf{T}(z) \tag{7.16}$$

The system transfer function in the Z-domain, which describes how the input $\triangle T$ is transformed into the output RT, can also be given as:

$$\mathbf{G}(z) = \frac{\mathbf{RT}(z)}{\triangle \mathbf{T}(z)} = \frac{bz}{z-a} \tag{7.17}$$

Considering these equations, the feedback control system in Figure 7.3 can be simplified to the block diagram of our control system which is shown in Figure 7.4. In this figure $\boldsymbol{C}(z)$ and $\boldsymbol{G}(z)$ represent the controller and the system model respectively. The change in the time slot size $\triangle T(k)$ is the control input to the plant, which is generated by the controller through computing the error lying between the reference value $TRT(k)$ and the system output $RT(k-1)$ in the last sampling period. When $\triangle T(k) > 0$, it means that the time slot size will be increased; otherwise, when $\triangle T(k) < 0$, the time slot size will be decreased.

In this control system, it is the responsibility of the controller to determine the value of the control input $\triangle T(k)$ based on the value of $e(k)$. This is done by specifying a control law that quantifies how to set the control input to the target system. Here we use the proportional control law, which is formally given by the Equation 7.18.

$$\triangle T(k) = K_p\, e(k) \tag{7.18}$$

In this equation K_p is a constant that is chosen when designing the proportional controller, and is often referred to as the controller gain. The transfer function for this controller is:

$$\mathbf{C}(z) = \frac{\triangle \mathbf{T}(z)}{\mathbf{e}(z)} = K_p \tag{7.19}$$

Feedback control theory provides a series of mathematics tools which allow us to choose and tune the controller parameters (K_p for our controller) efficiently, in order to achieve many desirable properties of this controller, e.g. stability, accuracy, etc.. The closed-loop system transfer function, which relates the reference input (the target average response time $\mathbf{TRT}(z)$[4]) to the output $\mathbf{RT}(z)$, is needed to apply these tools. Equation 7.20 gives the closed-loop transfer function for our feedback control loop. The derivation of this equation requires a little algebra and it is presented in [81].

$$\mathbf{F}(z) = \frac{\mathbf{RT}(z)}{\mathbf{TRT}(z)} = \frac{\mathbf{C}(z)\,\mathbf{G}(z)}{1 + \mathbf{C}(z)\,\boldsymbol{G}(z)\,z^{-1}} \tag{7.20}$$

In this equation, the numerator of $\mathbf{F}(z)$ is the feed-forward transfer function and the denominator is one plus the loop transfer function. In general, this relationship holds as long as the feedback signal is subtracted from the reference input, regardless of the specific blocks in the feedback control system.

By manipulating and substituting Equations 7.17 and 7.19 in Equation 7.20 we get:

$$\mathbf{F}(z) = \frac{K_p \frac{bz}{z-a}}{1 + K_p \frac{bz}{z-a}\, z^{-1}} = \frac{K_p\, b}{1 - (a - K_p\, b)z^{-1}} \tag{7.21}$$

[4]Changes in the reference input reflect changes in policy, such as changing a service-level objective.

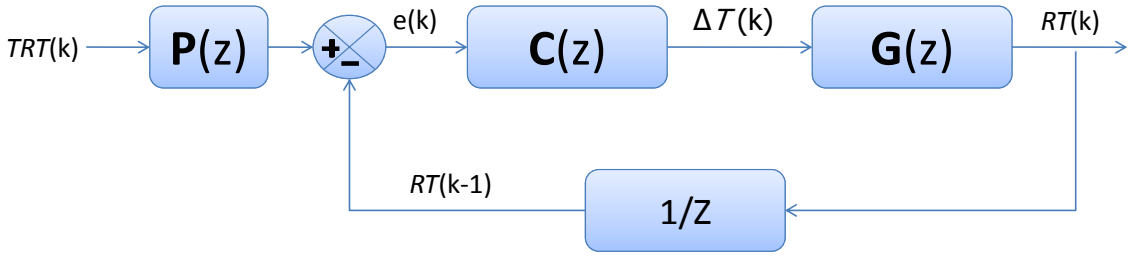


Figure 7.5: Block Diagram of The Feedback Control System With Precompensation.

The most basic property of a controller is stability; that is, a bounded input always produces a bounded output. The stability of the closed-loop system is determined by the poles of the transfer function in Equation 7.21. The poles are the values of z for which the denominator is equal to zero, and the system is stable in closed-loop if all of these poles have a magnitude of less than 1, i.e.:

$$1 - (a - K_p\, b)z^{-1} = 0$$

The transfer function has only one pole which is:

$$z = a - K_p\, b \tag{7.22}$$

And the feedback control system is stable when:

$$\mid a - K_p\, b \mid < 1 \tag{7.23}$$

or:

$$\frac{a-1}{b} < K_p < \frac{a+1}{b} \tag{7.24}$$

Therefore, this relation presents a constraint on the choice of the controller parameter (K_p) values in Equation 7.21. These values must be determined so that all poles of the system transfer function are placed in the unit circle.

Considering that our feedback control system, given by the transfer function in Equation 7.20, is stable, the system final output will converge to a final value RT_{ss}, which is called the steady state output of the system. The difference between this steady state value and the target value TRT, which is called the steady state control error, determines the accuracy of the closed-loop control system. Thus, if the steady state output is the desired output value and the

steady state control error (e_{ss}) is equal to zero, then the system can perform as expected and can achieve the performance control target.

It is known from control theory that $e_{ss} = 0$ if, and only if $\mathbf{F}(1) = 1$. $\mathbf{F}(1)$ is called the steady state gain of the system, which quantifies the steady-state effect of the input on the output. In Equation 7.20 the input is the reference value $\mathbf{TRT}(z)$, which is the target average response time, and the output is the measured average response time in the system $\mathbf{RT}(z)$. Therefore, the steady state gain for our feedback control system can be given as:

$$\mathbf{F}(1) = \frac{RT_{ss}}{TRT} \tag{7.25}$$

As a proportional controller is used in our control system, we can see from Equation 7.21 and considering the previous discussion that steady state error can be reduced by using a larger K_p. However, the proportional controller will result in $\mid e_{ss} \mid > 0$, as we need a value of K_p so that:

$$\frac{K_p\, b}{1 - (a - K_p\, b)} = 1 \tag{7.26}$$

In order to eliminate this steady state error precompensation is usually used with proportional controllers. In Figure 7.5 the feedback system includes a block labeled $\mathbf{P}(z)$, the precompensator, which provides a way to adjust the reference signal $TRT(k)$ to correct for the steady-state gain of the closed-loop system. Using this precompensator, the transfer function from the reference input to the output is:

$$\acute{\mathbf{F}}(z) = \frac{\mathbf{RT}(z)}{\mathbf{TRT}(z)} = \frac{K_p\, b\, \mathbf{P}(z)}{1 - (a - K_p\, b)z^{-1}} \tag{7.27}$$

We want to make $\acute{\mathbf{F}}(1) = 1$ by selecting a constant precompensator and a value of K_p based on the constraint 7.23 and the relation:

$$\frac{K_p\, b\, \mathbf{P}(z)}{1 - (a - K_p\, b)} = 1 \tag{7.28}$$

So we let:

$$1 - (a - K_p\, b) = 1 \tag{7.29}$$

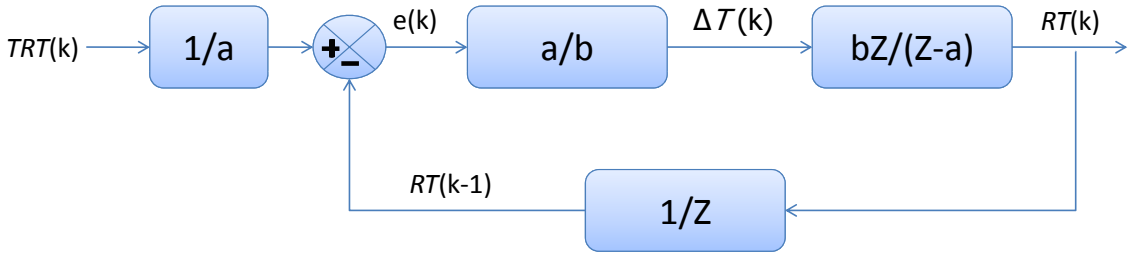


Figure 7.6: Block Diagram of The Control System Model

Then we can get:

$$K_p = \frac{a}{b} \tag{7.30}$$

Putting the value of K_p in Equation 7.28 we get:

$$\mathbf{P}(z) = \frac{1}{a} \tag{7.31}$$

And the control system block diagram is as shown in Figure 7.6.

This controller will be activated automatically to adjust the time slot size in two cases. First, when the measured average response time of the accepted requests exceeds the target average response time. Second, when the number of rejected requests during the last sample period is not equal to zero.

7.3 Experiments

In order to validate the proposed controller and to evaluate its performance, we use the staged design-based service simulation prototype which we have implemented using the Möbius environment – See Section 6.3.

In our experiments e_i (the mean request's service time at stage i) takes the values: 1 ms for stages A, B, D and E; and 2 ms for stage C. For all stages l_i is equal to e_i. The system starts running with a given value of T (100 ms in our experiments) and the control sample period is one second.

Like in the experiments of the previous chapter, the presented results here are the average of multiple simulation runs with relative confidence intervals of 10% and a confidence level of more than 90%.

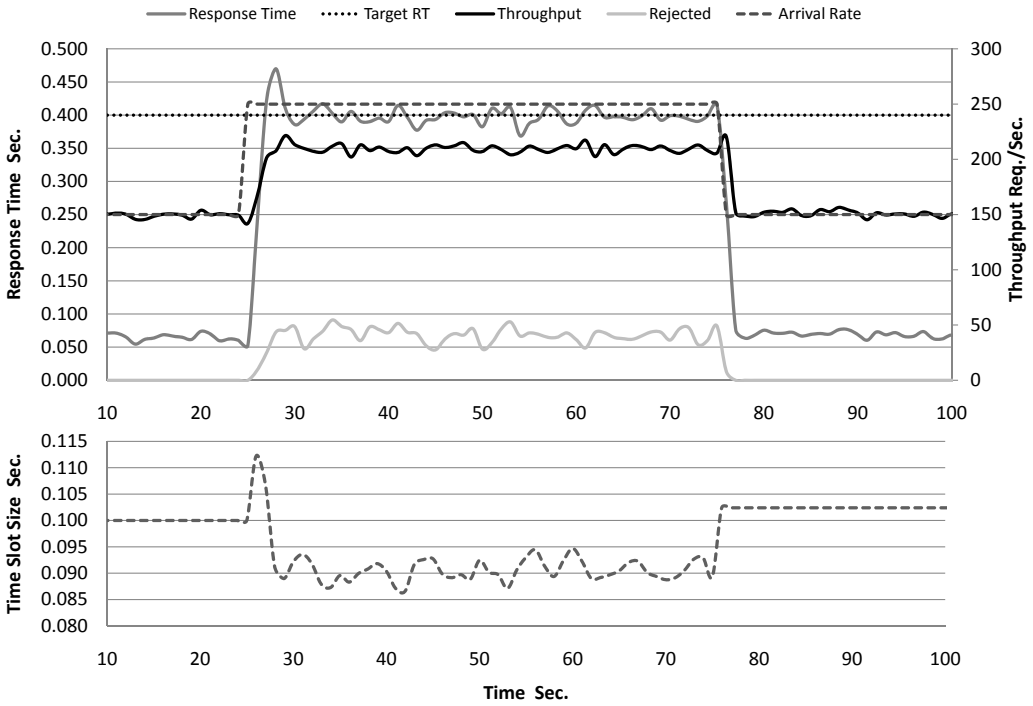


Figure 7.7: Dynamic Changes in System Workload

7.3.1 Maintaining Stable Response Time

These experiments test the ability of our performance controller to keep the average response time under the desired target TRT during unpredictable dynamic changes in the system.

In Figure 7.7 after running the system for 25 seconds the requests arrival-rate increases from 150 to 250 requests per second, which brings the system in an overload situation. The figure shows that as the load increases the controller automatically adjusts the time slot size to accept as much requests into the system as the system is able to process without violating the service level target. Additional requests that can not be processed are rejected until the load returns to 150 requests per seconds after 50 seconds.

In Figure 7.8 the requests arrival-rate does not change but the proportion of bottleneck requests (requests that go to the bottleneck stage (C) in Figure 6.2b) changes. After running the system for 20 seconds the proportion of bottleneck requests decreases from 50% to 5% and after more 20 seconds it increases to

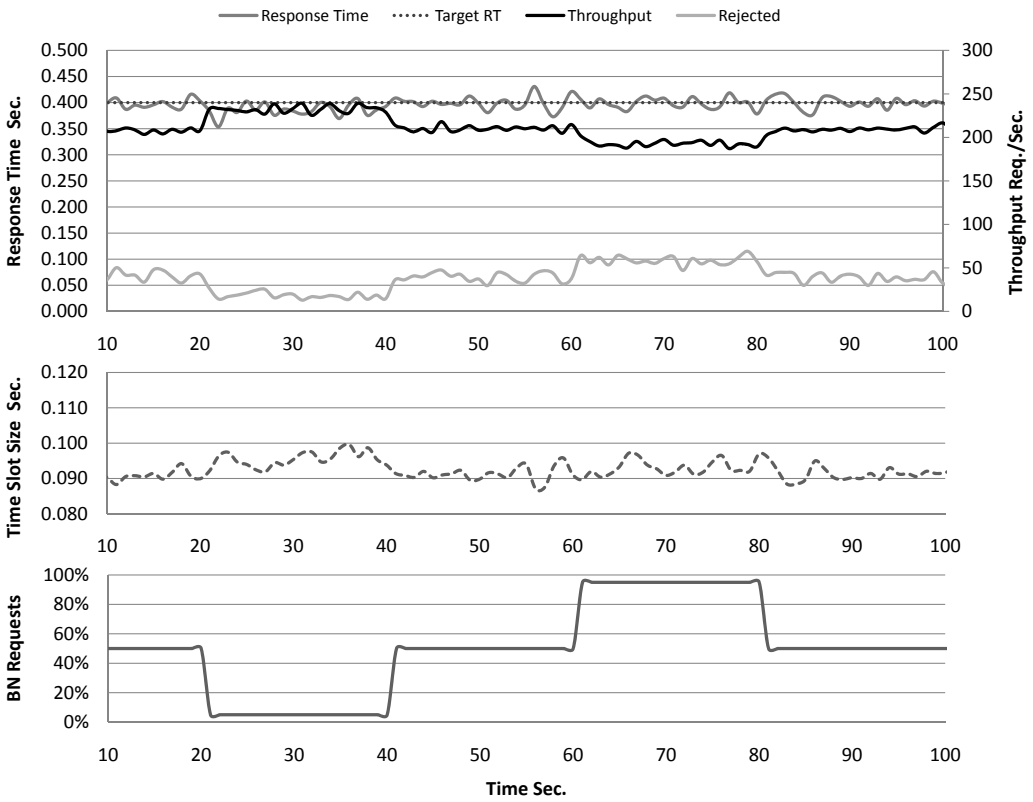


Figure 7.8: Dynamic Changes in The proportion of Bottleneck Requests

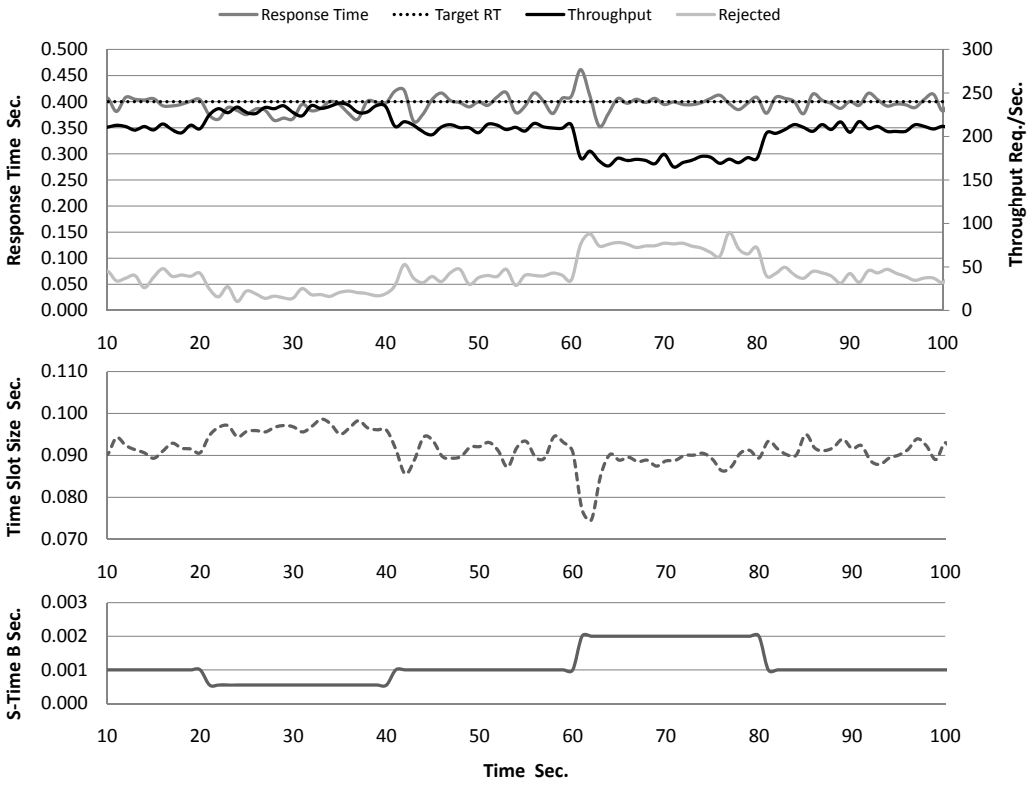


Figure 7.9: Dynamic Changes in Stage B Service Time

50% again. Then after more 20 seconds it increases to 95% for 20 seconds and then returns again to 50%. The experiment shows a smaller effect of the performance controller on the time slot size in this case. We argue that in such dynamic changes which happen within the system (changes in work-flow) the main responsibility to adjust the system is of the global controller which adapts the allocation of system resources according to the changes in stages needs and adjusts the number of accepted requests at each time slot. In this case, the performance controller adjusts the time slot size only when there is still an opportunity to process more requests by increasing the time slot size or there is a need to decrease it in order to avoid the performance target violation.

Figure 7.9 shows the behavior of our performance controller under another form of dynamic changes in the system. In this experiment the service time that a request needs to be processed at stage B in our staged service (Figure 6.2b) decreases to 0.5 ms after running the system for 20 seconds and then after more 20 seconds it returns to its original value (1 ms), after that it increases

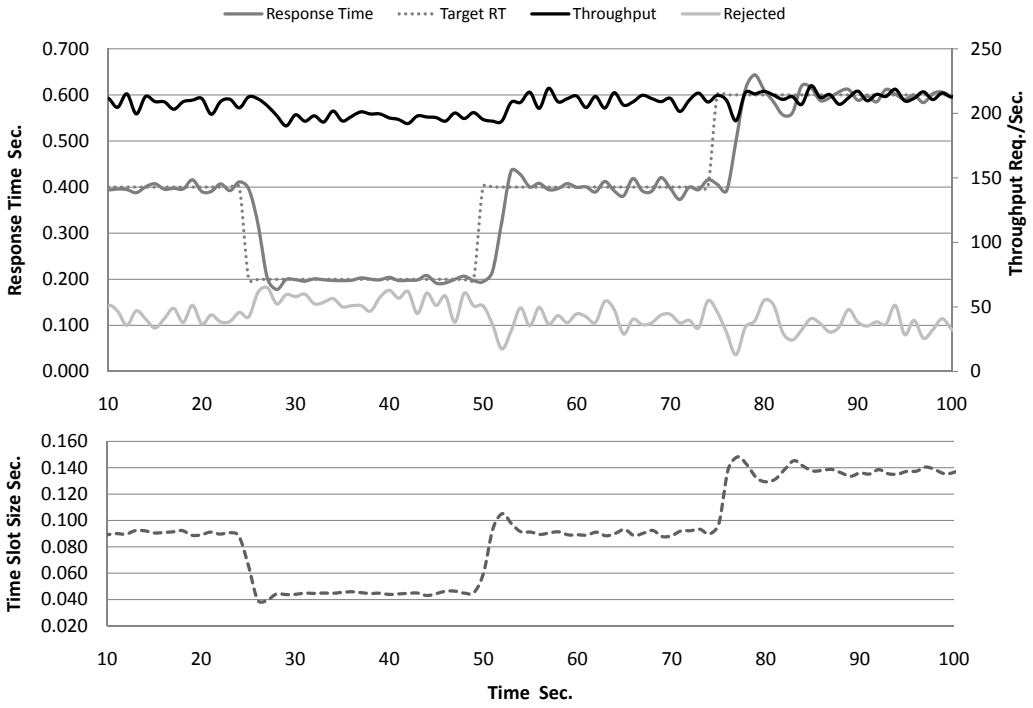


Figure 7.10: Tracing a Dynamic Target Response Time.

to 2 ms again for 20 seconds. Such changes in a stage service time can represent many scenarios like changing the algorithm that is implemented within a stage, rising or degrading the quality of the service at run time, a software or hardware failure in the system, etc. From the Figure we can see that our controller can respond to these dynamic changes automatically in order to keep the average response time of the accepted requests under the desired target. However, like in the previous experiment, the global controller play a main role here to adjust the number of requests that are accepted at each time slot.

7.3.2 Trace a Dynamic Target Response Time

The goal of this experiment is to test the ability of the controller to guarantee that the average response time of the accepted requests is below TRT, and its ability to respond to changes in this target response time TRT at run-time. We begin with a target response time of 400 ms. After running the system for 25 seconds, we decrease the target to 200 ms for 25 seconds and then again after 25 more seconds we increase it to 600 ms.

Figure 7.10 shows that as the target response time decreases the controller responds automatically by decreasing the time slot size which results in increasing the number of rejected requests and decreasing the system throughput. Conversely, as the target response time increases the controller automatically increases the time slot size to accept more requests per time slot and to decrease the number of rejected requests.

7.4 Discussion

In this section we bring it together, and show how our feedback-based performance controller works within the three-layers control architectures, which is presented in Chapter 5.

The system's administrator determines the target performance of the system, which is the target average response time. At run-time the performance controller periodically compares this value with the average response time of the accepted requests that is measured in the system, and adjusts the time slot size accordingly. The new time slot size will be used by the global controller as the time slot duration to allocate the available processing resources to the individual stages within the following sample period. At the entrance stage only the requests that can be processed within the time slot will be accepted, while other requests will be rejected, forwarded to other servers or a kind of service degradation can be done. When the resources are allocated to a stage to process the waiting requests in its incoming queue, the local controller of the stage allocates the execution threads to the requests according to the policy that is used within this stage.

Considering this control architecture, disturbances in the system that result in from changes in the processing costs of requests at the different stages or from changes in the requests flow in the staged service are almost hidden for the performance controller by the local controller which observe these changes and the global controller which changes the number of accepted requests dynamically to fill up the time slot period. For this reason, the system model on which the performance controller depends to adjust the time slot size will be robust to such disturbances in the system.

As a result, and according to the experiments, our performance control approach is able to adapt the system to the dynamic changes in order to achieve desired performance targets with simple control algorithm and automatic tuning of the system parameters. Even while the system works with unpredictable loadings with high and different variances, it can still perform as expected, and can meet the target performance levels.

8 Conclusions and Future Work

In this thesis we have proposed a three-layers control architecture for resource management and performance control of staged design-based Internet services. In addition, we have presented an adaptive resource allocation policy and a feedback based performance controller that follow this control architecture. This Chapter summarizes our results, presents many directions for future work and a variety of potential implementation areas.

8.1 Conclusions

The staged architecture has emerged as a programming paradigm to implement high performance Internet services that support massively concurrent demands. This design avoids many pitfalls related to the conventional concurrency models, and presents system design advantages at hardware- and software engineering level. However, this architecture introduces many challenges related to resource allocation to the individual stages which have different requirements, and other challenges that appear when the system is to be tuned to achieve a desired performance target or to guarantee a determined service level. These challenges are also magnified by the complexity which is presented by the structure of modern server's hardware platforms.

To address these challenges, this thesis has proposed a three-layers control architecture for resource management and performance control of staged design-based services. The control architecture consists of three layers of controllers, local controller within each stage, global controller, and a performance controller.

Based on this control architecture, the thesis has also presented an adaptive resource allocation policy to allocate processing resources to the stages of the staged service, and a feedback-based performance controller for performance control and service level guarantees of staged Internet services. The

resource allocation policy represents the global controller in the three-layers control architecture, that control allocating the system resources according to the stages needs, and the feedback-based performance controller represents the performance control layer in the control architecture.

Experimental results of our simulation-based study have demonstrated that the proposed resource allocation policy leads to a better performance under different characteristics of the system and under dynamically changing load conditions. It has been also shown that this policy can benefit from locality within the individual stages, better utilize the available processing resources, and avoid the overheads that appear in modern parallel processing systems.

In addition, our experiments have demonstrated the ability of the proposed performance controller to adjust the system at run-time dynamically and automatically to maintain the desired performance targets under a variety of dynamic changes in the system; and have also demonstrated that this controller can adjust the system to trace dynamic changes in performance targets at run-time.

Compared with other approaches, our control approach exhibits many desired property in automatic control for highly dynamic systems. It provides an effective way to guarantee service quality under high concurrency. By employing feedback control theory our performance controller can significantly reduce the cost of manual system configurations and provide reliable parameter settings, which enhance the system performance. Additionally, depending on the control architecture and communications among the three layers of controllers, we could reduce the needed prior knowledge of the system behavior, which is a problem that characterizes other approaches in the staged design.

8.2 Future Work

In this thesis, the Staged Event-Driven Architecture (SEDA) has been deployed as the ground-work to support highly concurrent demands. By combining the advantages of this architecture with the advantages of automatic feedback control and control theory, we have proposed a new approach for resource management and performance control of SEDA-based Internet services. How-

ever, the thesis has presented also many issues that have to be revisited in order to understand and optimize systems that are based on the staged design.

For example, relations between stage's structure characteristics and the characteristics of the system hardware platform have to be studied in more details in order to increase the benefit from performance improvements in modern computing systems. A more detailed study of the effects of different combinations of the hierarchic parallelism that appear in today's server platforms is also very important. To some extent, our resource allocation approach has considered these issues. However, allocation policies that consider the stage's working set size when co-allocating batches from different stages on a shared node in the processing units tree are also possible, and deserve more investigation.

On the other hand, our global controller depends on investigating the current state of the staged service in order to take decisions for resource allocation to the individual stages. Enriching this control layer with a control-theoretic based controller can increase the accuracy of the approach for long-term allocations. Such an approach gives the opportunity to implement our control policy in higher scales system, like clusters or distributed systems, in a more effective and reliable manner.

Considering the performance control layers, the proposed approach depends on adjusting the size of the time slot size at run-time to achieve performance targets. In this thesis, we have presented a proportional based pre-compensator controller. We believe that testing and comparing the performance of other control-theoretic based approaches can be fruitful. In addition, to cope with cost efficiency and "Green IT" requirements, our approach can be extended to control the number of used processing units in the system in addition to the time slot size. That can be achieved depending on many techniques that exist in today's processors and enable switching off processing units to save energy [144], or depending on "pay as you use" computer leasing to save costs.

The approach which is presented in this thesis depends on combining the resource allocation policy with the feedback-based performance controller to reject excessive requests that arrive at the system, this approach can be ex-

tended to achieve a variety of targets. Today, it is desired to provide individual performance guarantees to individual client categories; our approach can be extended to achieve such requirements by using different queues for different clients categories and applying rejection at each queue depending on performance targets. Approaches that are based on service degradation can be also implemented, depending on a policy that combines the value of the time slot size with different levels of the service.

Besides this, in the thesis, we have concentrated on the global controller and the performance control depending on event-driven stages and local controllers which use as much execution threads as the number of processing units that are allocated to the stage. The presented approach is more generic, and the effect of the different implementations of the local controller deserves studying.

Finally, our experimental results have based on a simulation prototype which has provided the needed flexibility to study the effect of a variety of system related parameters. However, implementing a case study representing a realistic scenario may raise other research questions.

List of Symbols

Ω	Number of elements in the data series that is used for system identification.
Υ_i	Total processing time needed by a request at stage i in the existence of parallelity.
a, b	System Model Parameters.
$B_i(k)$	Size of the batch of requests from stage i which is to be processed in the sample period k.
$\boldsymbol{C}(z)$	Controller Transfer Function in the Z-Domain.
C	Number of Cores per Chip.
$e(k)$	Estimation error in the sample period k.
e_i	Mean processing time that is needed by a request at stage i.
e_{ss}	Steady State Error.
$\mathbf{F}(z)$	Closed-Loop System Transfer Function.
$\acute{\mathbf{F}}(z)$	Transfer Function of the Closed-Loop System with Precompensation.
$\boldsymbol{G}(z)$	System Transfer Function in the Z-Domain.
g	Gate Size in the Gated approach.
$J(a, b)$	Sum of the squared errors.
K_p	Proportional Controller Gain.
l_i	Load Time for Stage i working set.
M	Number of Stages.
N	Number of Processing Chips.

OHL_{ij} Additional processing time needed by a request processing from stage i as a result of processing requests from stage j in parallel on other cores at the same chip.

$OHLV_{ij}$ System specific constant that represents the additional processing time needed by a request processing from stage i as a result of processing a request from stage j in parallel on another core at the same chip.

OHR_{ij} Additional processing time needed by a request processing from stage i as a result of processing requests from stage j in parallel on other chips.

$OHRV_{ij}$ System specific constant that represents the additional processing time needed by a request processing from stage i as a result of processing a request from stage j in parallel on another chip.

$\mathbf{P}(z)$ Precompensator's Function.

$p_i(k)$ Portion of processing time that is to be given to stage i in the sample period k.

$\widetilde{RT}(k)$ Estimated average response time in the sample period k.

$RT(k)$ Measured average response time in the sample period k.

RT_{ss} Steady State Average Response Time.

$S_i(k)$ Number of requests in the queue of stage i in the sample period k.

$\triangle T(k)$ Change in the time slot size in the sample period k.

T Time Slot Size

$t_i(k)$ Allocated processing time to stage i in the sample period k.

$th_i(k)$ Average throughput of stage i in the sample period k.

TRT Target Response Time.

$w_i(k)$ Estimated processing time that is needed by stage i to process all the requests in its incoming queue in the sample period k.

X_{ij} Number of requests from stage j that are processed with a request from stage i in parallel on cores at the same chip.

Y_{ij} Number of requests from stage j that are processed with a request from stage i in parallel on other chips.

Previously Published Work

- M. S. Al-Hakeem and H.-U. Heiß, "Adaptive scheduling for staged applications: The case of multiple processing units," in Fifth International Conference on Internet and Web Applications and Services (ICIW 2010). Los Alamitos, CA, USA: IEEE Computer Society, May 2010, pp. 51–60.
- M. S. Al-Hakeem, J. Richling, and H.-U. Heiß, "Performance Guarantees for Staged Design Based Services," in Workshop on System Communication and Engineering in Computer Science, October 2010, invited Paper.
- M. S. Al-Hakeem, J. Richling, G. Mühl, and H.-U. Heiß, "An adaptive scheduling policy for staged applications," in Fourth International Conference on Internet and Web Applications and Services (ICIW 2009). IEEE, May 2009.

Bibliography

[1] "The Möbius Tool," http://www.mobius.uiuc.edu/, 2008, last accessed.

[2] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 80–96, 2002.

[3] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu, "Feedback Performance Control in Software Services," *IEEE Control Systems Magazine*, vol. 23, p. 2003, 2003.

[4] T. Abdelzaher and C. Lu, "Modeling and performance control of internet servers," in *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*, vol. 3, 2000, pp. 2234–2239 vol.3.

[5] Accoria Networks, Inc., "Rock Web Server," http://www.accoria.com/, 2008.

[6] Advanced Micro Devices, Inc., "Six-Core AMD Opteron™ processor features," www.amd.com, 2009.

[7] ——, "AMD Opteron™ 6000 series platform," www.amd.com, 2010.

[8] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management," in *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 289–302.

[9] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" in *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1999, pp. 266–277.

[10] M. S. Al-Hakeem, J. Richling, and H.-U. Heiß, "Performance Guarantees for Staged Design Based Services," in *Workshop on System Communication and Engineering in Computer Science*, October 2010, invited Paper.

[11] M. S. Al-Hakeem, J. Richling, G. Mühl, and H.-U. Heiß, "An adaptive scheduling policy for staged applications," in *Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*. IEEE, May 2009.

[12] Alexa Internet, Inc., "Traffic History Graph for aljazeera.net," http://www.alexa.com/, 2009.

[13] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian, "Resource management in the autonomic service-oriented architecture," in *IEEE International Conference on Autonomic Computing, 2006. ICAC '06.*, June 2006, pp. 84–92.

[14] V. Almeida, D. Menascé, R. Riedi, R. Fonseca, and W. M. Jr., "Characterizing and Modeling Robot Workload on E-Business Sites," in *Proceedings of 2001 ACM Sigmetrics Conf.* ACM, June 2001.

[15] V. Almeida, D. Menascé, R. Riedi, F. Peligrinelli, R. Fonseca, and W. M. Jr., "Analyzing Web Robots and Their Impact on Caching," in *Proceedings of the 6th Web Caching and Content Delivery Workshop*, 2001.

[16] J. Alpert and N. Hajaj, "We knew the web was big..." http://googleblog.blogspot.com/, July 2008.

[17] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite, "Specification and implementation of dynamic web site benchmarks," *2002 IEEE International Workshop on Workload Characterization, WWC-5*, pp. 3–13, Nov 2002.

[18] AnalogX, LLC., "Internet traffic report," www.internettrafficreport.com, 2009.

[19] Apache Software Foundation, "The Apache Web Server," http://www.apache.org/, Last accessed, 2008.

[20] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: the search for invariants," *SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 126–137, 1996.

[21] M. Arrington, "Facebook now nearly twice the size of MySpace worldwide," http://www.techcrunch.com/, January 2009.

[22] AT&T Inc., "AT&T to Invest More Than $17 Billion in 2009 to Drive Economic Growth," www.att.com, March 2009.

[23] J. S. Auckland, "Fryup: Microsoft photosynthed," http://computerworld.co.nz, Aug 2008.

[24] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The Design of OpenMP Tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[25] G. Bangs, P. Druschel, and J. C. Mogul, "Better operating system features for faster network servers," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 3, pp. 23–30, 1998.

[26] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[27] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," *SIGARCH Comput. Archit. News*, vol. 26, no. 3, pp. 3–14, 1998.

[28] V. Beltran, D. Carrera, J. Torres, and E. Ayguade, "Evaluating the Scalability of Java Event-Driven Web Servers," in *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 134–142.

[29] S. Bhatia, C. Consel, and J. Lawall, "Memory-manager/scheduler codesign: optimizing event-driven servers to improve cache behavior," in *ISMM '06: Proceedings of the 5th international symposium on Memory management*. New York, NY, USA: ACM, 2006, pp. 104–114.

[30] S. Bhatia, J. Lawall, and C. Consel, "Minimizing cache misses in an event-driven network server: A case study of TUX," in *The 31st IEEE Conference on Local Computer Networks (LCN)*. Tampa, Florida: IEEE Computer Society, Nov. 2006, pp. 47–54.

[31] J. M. Blanquer, A. Batchelli, K. Schauser, and R. Wolski, "Quorum: flexible quality of service for internet services," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 159–174.

[32] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*. New York, NY, USA: ACM, 2000, pp. 195–203.

[33] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel, "A multithreaded PowerPC processor for commercial servers," *IBM Journal of Research and Development*, vol. 44, no. 6, pp. 885–898, 2000.

[34] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media, Inc., November 2005.

[35] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: evidence and implications," in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, Mar 1999, pp. 126–134 vol.1.

[36] G. Brown, "The internet is as vital as water and gas," http://www.timesonline.co.uk, June 2009.

[37] R. E. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[38] J. R. Bulpin and I. A. Pratt, "Hyper-threading aware process scheduling heuristics," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 27–27.

[39] J. Burt, "AMD fires back at Facebook," www.eweek.com, July 2009.

[40] J. Carlström and R. Rom, "Application-aware admission control and scheduling in web servers," in *INFOCOM*, 2002.

[41] H. Chen and P. Mohapatra, "Session-based overload control in QoS-aware Web servers," *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 516–524 vol.2, 2002.

[42] Y. Chen, X. Fan, W. Yang, K. Chen, and G. Xu, "Stage based parallel programming model for high concurrency, stateful network services: internals and design principles," *Int. J. High Perform. Comput. Netw.*, vol. 3, no. 1, pp. 33–44, 2005.

[43] L. Cherkasova and M. Karlsson, "Dynamics and evolution of web sites: Analysis, metrics and design issues," in *Proceedings of the Sixth International Symposium on Computers and Communications*, 2001, pp. 3–5.

[44] G. S. Choi, J.-H. Kim, D. Ersoz, and C. R. Das, "A multi-threaded PIPELINED Web server architecture for SMP/SoC machines," in *WWW '05: Proceedings of the 14th international conference on World Wide Web*. New York, NY, USA: ACM, 2005, pp. 730–739.

[45] Cisco Systems, Inc., www.cisco.com, 2009.

[46] S. Clara, "Intel Previews Intel Xeon® 'Nehalem-EX' Processor," http://www.intel.com, May 2009, press Release.

[47] ComScore, Inc., "In australia, online retail sites see traffic surge in december," www.comscore.com, January 2010, press Release.

[48] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, "Performance implications of single thread migration on a chip multi-core," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 80–91, 2005.

[49] DailyTech LLC., "Massive DDOS Attacks on U.S., South Korea, Came From the UK, Researcher Says," http://www.reuters.com/, July 2009.

[50] V. Daniel, P. Prashant, and R. Dan, "Provisioning servers in the application tier for e-commerce systems," *ACM Trans. Interet Technol.*, vol. 7, no. 1, p. 7, 2007.

[51] K. Debattista, K. Vella, and J. Cordina, "Wait-free cache-affinity thread scheduling," *Software, IEE Proceedings* -, vol. 150, no. 2, pp. 137–146, April 2003.

[52] ——, "Cache-Affinity Scheduling for Fine Grain Multithreading," in *Communicating Process Architectures 2002*, sep 2002, pp. 135–146.

[53] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *SIGCOMM '89: Symposium proceedings on*

Communications architectures & protocols. New York, NY, USA: ACM, 1989, pp. 1–12.

[54] P. J. Denning, "Thrashing: its causes and prevention," in *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. New York, NY, USA: ACM, 1968, pp. 915–922.

[55] Y. Diao, N. G, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server," in *Proceedings of the Network Operations and Management Symposium 2002*, 2002, pp. 219–234.

[56] L. Dignan, "E-commerce sites: You have 2 seconds to load your Web Pages," http://www.zdnet.com/, September 2009.

[57] DomainTools, http://www.domaintools.com/internet-statistics/, March 2009, last Accessed.

[58] F. Douglis and M. F. Kaashoek, "Guest editors' introduction: Scalable internet services," *IEEE Internet Computing*, vol. 5, pp. 36–37, 2001.

[59] I. Eklektix, "Scheduling domains," http://lwn.net/Articles/80911/, April 2004.

[60] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM, 2004, pp. 276–286.

[61] J. C. F. M. David and R. H. Campbell, "Context Switch Overheads for Linux on ARM Platforms," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[62] A. Fedorova, M. Seltzer, D. Nussbaum, and C. Small, "Throughput-oriented scheduling on chip multithreading systems," Harvard University, Technical Report TR-17-04, August 2004.

[63] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 26–26.

[64] K. Fiveash, "Microsoft's Photosynth falls out of cloud," http://www.theregister.co.uk/, August 2008.

[65] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[66] J. D. Gelas, "High-End x86: The Nehalem EX Xeon 7500 and Dell R810," www.anandtech.com, April 2010.

[67] P. Gelsinger, P. Gargini, G. Parker, and A. Yu, "Microprocessors circa 2000," *Spectrum, IEEE*, vol. 26, no. 10, pp. 43–47, Oct 1989.

[68] D. C. Gilbert, "Modeling spin locks with queuing networks," *SIGOPS Oper. Syst. Rev.*, vol. 12, no. 1, pp. 29–42, 1978.

[69] J. Gorbold, "AMD Opteron 6174 vs Intel Xeon X5650 Review," www.bit-tech.net, March 2010.

[70] M. Gordon, "Staged design for highly concurrent web servers," Master's thesis, Technische Universität Berlin, 2005.

[71] S. D. Gribble, "A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction," Ph.D. dissertation, UC Berkeley, September 2000.

[72] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguade, "Characterizing Secure Dynamic Web Applications Scalability," *19th IEEE International Parallel and Distributed Processing Symposium, 2005. Proceedings.*, p. 108a, April 2005.

[73] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé, "Session-based adaptive overload control for secure dynamic web applications," in *34th International Conference on Parallel Processing (ICPP 05)*, 2005, pp. 14–17.

[74] A. Hac, "On the modeling of shared resources with various lock granularities using queueing networks," *Performance Evaluation*, vol. 6, no. 2, pp. 103 – 115, 1986.

[75] I. Haddad and G. Butler, "Experimental studies of scalability in clustered web systems," *Parallel and Distributed Processing Symposium, International*, vol. 9, p. 185b, 2004.

[76] S. Harizopoulos and A. Ailamaki, "Affinity scheduling in staged server architectures," Carnegie Mellon University, Tech. Rep., 2002.

[77] ——, "A case for staged database systems," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, 2003.

[78] ——, "Improving instruction cache performance in OLTP," *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 887–920, 2006.

[79] H.-U. Heiß, "Operating system design," http://www.kbs.tu-berlin.de, 2010, lecture.

[80] J. Held, J. Bautista, and S. Koehl, "From a few cores to many: A tera-scale computing research review," www.intel.com, 2006, white Paper, Intel Corporation.

[81] J. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, Inc., 2004.

[82] J. L. Hellerstein, F. Zhang, and P. Shahabuddin, "A statistical approach to predictive detection," *Computer Networks*, vol. 35, no. 1, pp. 77–95, 2001.

[83] J. Hennessy, "The future of systems research," *Computer*, vol. 32, no. 8, pp. 27–33, Aug 1999.

[84] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Elsevier, Inc., 2007.

[85] IBM, "IBM unleashes world's fastest chip in powerful new computer," www.ibm.com, May 2007, press release.

[86] IBM, "IBM Unveils New POWER7 Systems To Manage Increasingly Data-Intensive Services," www.ibm.com, Feb. 2010, press release.

[87] Intel Corporation, "Intel 4-Processor Server System S7000FC4UR," http://www.intel.com/.

[88] ——, "Intel® threading building blocks," www.intel.com, 2007, tutorial.

[89] ——, "Intel® Microarchitecture, Codenamed Nehalem," www.intel.com, 2009.

[90] ——, "Single-chip cloud computer," www.intel.com, December 2009.

[91] ——, "Intel® Xeon® Processor 5000 Sequence," www.intel.com, 2010, last Accessed.

[92] Internet World Stats, "Internet usage statistics," http://www.internetworldstats.com/, December 2008.

[93] C. Jun, Z. Ming-Tian, and Y. Xiao-Yan, "Feedback control quality adaptation framework for seda system," in *Apperceiving Computing and Intelligence Analysis, 2008. ICACIA 2008. International Conference on*, Dec. 2008, pp. 255–259.

[94] K. Kant, R. Iyer, and P. Mohapatra, "Architectural impact of secure socket layer on internet servers," *International Conference on Computer Design 2000. Proceedings*, pp. 7–14, 2000.

[95] V. Kazempour, A. Fedorova, and P. Alagheband, "Performance implications of cache affinity on multicore processors," in *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 151–161.

[96] D. Kegel, "The C10K Problem," www.kegel.com, September 2006.

[97] G. Key, "Memory Scaling on Core i7 - Is DDR3-1066 Really the Best Choice? ," www.anandtech.com, June 2009.

[98] P. Koka and M. H. Lipasti, "Opportunities for Cache Friendly Process Scheduling," in *Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[99] R. Kokku, U. Shevade, N. Shah, H. M. Vin, and M. Dahlin, "Adaptive processor allocation in packet processing systems," University of Texas at Austin, Tech. Rep., 2004.

[100] V. Krishnamoorthy, N. Unni, and V. Niranjan, "Event-driven service-oriented architecture for an agile and scalable network management system," in *Next Generation Web Services Practices, 2005. NWeSP 2005. International Conference on*, Aug. 2005, pp. 6 pp.–.

[101] R. Kumar, Y. Maharaj, and E. Torlak, "Squnk: Stages, queues 'n kontrollers, a high performance network application toolkit." December 2001.

[102] A. Kvalnes, D. Johansen, R. van Renesse, and A. Arnesen, "Vortex: an event-driven multiprocessor operating system supporting performance isolation." University of Tromso, Technical Report 2003-45, June 2003.

[103] J. R. Larus and M. Parkes, "Using cohort scheduling to enhance server performance (extended abstract)," in *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*. New York, NY, USA: ACM, 2001, pp. 182–187.

[104] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, pp. 3–19, 1979.

[105] D. Leijen and J. Hall, "Optimize managed code for multi-core machines," *MSDN Magazine*, October 2007.

[106] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*. New York, NY, USA: ACM, 2007, p. 2.

[107] P. Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives," *SIGPLAN Not.*, vol. 42, no. 6, pp. 189–199, 2007.

[108] Z. Li, "Fair service for high-concurrent requests," Master's thesis, The University of Sydney, August 2007.

[109] Z. Li, S. Chen, D. Levy, and J. Zic, "Auto-tune design and evaluation on staged event-driven architecture," in *MODDM '06: Proceedings of the 1st workshop on MOdel Driven Development for Middleware (MODDM '06)*. New York, NY, USA: ACM, 2006, pp. 1–6.

[110] Z. Li, D. Levy, S. Chen, and J. Zic, "Explicitly controlling the fair service for busy web servers," in *ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 159–168.

[111] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker, "Characterizing and modeling the behavior of context switch misses," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 91–101.

[112] L. Ljung, *System Identification Theory for the User*, 2nd ed. Prentice-Hall, Inc., 1999.

[113] N. R. Mahapatra and B. Venkatrao, "The processor-memory bottleneck: problems and solutions," *Crossroads*, p. 2, 1999.

[114] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 06, no. 01, pp. 4–15, February 2002.

[115] D. A. Menascé and V. A. F. Almeida, *Capacity Planning for Web Services, Metrics, Models, and Methods*. Prentice Hall, Inc., 2002.

[116] Microsoft Corporation, "Internet Information Services (IIS) 6.0."

[117] ——, "Task Parallel Library," www.microsoft.com, last Accessed.

[118] ——, "ASP.NET," http://www.asp.net/, 2009.

[119] ——, "Estimating Bandwidth Requirements and Connection Speed (IIS 6.0)," http://www.microsoft.com, 2009, last accessed.

[120] G. Moore, "Progress in digital integrated electronics," in *1975 International Electron Devices Meeting*, vol. 21, 1975, pp. 11–13.

[121] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, 1999.

[122] P. Mutton, "National rail website affected by snow," http://news.netcraft.com/, January 2010.

[123] NCSA, "Blue Waters computing system," www.ncsa.illinois.edu, 2010.

[124] Netcraft Ltd., "February 2009 Web Server Survey," 2009.

[125] J. Ng, "DDR3 Will be Cheaper, Faster in 2009," www.dailytech.com, January 2009.

[126] J. K. Ousterhout, "Why threads are a bad idea (for most purposes)," in *Invited Talk at 1996 Usenix Technical Conference*, 1996.

[127] P. Mutton, "Extended Validation SSL Certificates 2 Years Old," http://news.netcraft.com/, February 2009.

[128] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: an efficient and portable web server," in *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1999, pp. 15–15.

[129] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, "Comparing the performance of web server architectures," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 231–243, 2007.

[130] Parks Associates., "Parks associates forecasts over 640 million broadband households worldwide by 2013," www.parksassociates.com, July 2009.

[131] I. Paul, "Jackson's Death a Blow to the Internet," http://www.pcworld.com/, June 2009.

[132] Paul Mutton, "One Million SSL Sites on the Web," http://news.netcraft.com/, February 2009.

[133] Press Association, "Broadband investment to grow digital economy," www.ukinvest.gov.uk, January 2010.

[134] PriMetrica, Inc., "Global Internet Geography," http://www.telegeography.com/, 2009.

[135] X. Qie, R. Pang, and L. Peterson, "Defensive programming: using an annotation toolkit to build DoS-resistant software," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 45–60, 2002.

[136] R. M. Ramanathan and V. Thomas, "Platform 2015: Intel® Processor and Platform Evolution for the Next Decade," 2005, white Paper, Intel Corporation.

[137] J. Rattner, "Polaris points the way to terascale computing," www.zdnet.com, December 2006.

[138] J. Reinders, *Intel Threading Building Blocks*. O'Reilly Media Inc., 2007.

[139] REUTERS, "Hacker attacks silence Twitter, slow Facebook," http://www.reuters.com/, August 2009.

[140] D. Robsman, "Thread optimization," http://www.freepatentsonline.com/, November 2002.

[141] S. Bharti, V. Kaulgud and V. Niranjan, "Fine Grained SEDA Architecture for Service Oriented Network Management Systems," *International Journal of Web Services Practices*, vol. 1, no. 1-2, pp. 158–166, 2005.

[142] A. Savoia, "The science of web site load testing," www.keynote.com, 2000, White Paper.

[143] I. Schmerken, *Can the Market's Systems Keep Up With Electronic Trading?*, Wall Street & Technology, February 2007.

[144] J. H. Schönherr, J. Richling, M. Werner, and G. Mühl, "Event-driven processor power management," in *1st International Conference on Energy-Efficient Computing and Networking (e-energy 2010)*. New York, NY, USA: ACM, apr 2010, pp. 61–70.

[145] R. J. Shapiro, "The Internet's Capacity To Handle Fast-Rising Demand for Bandwidth," US Internet Industry Association, September 2007.

[146] S. Siddha, "Multi-core and linux kernel," http://software.intel.com/, June 2007.

[147] R. Sim, "We heard you loud and clear," http://mailcall.spaces.live.com/, February 2008.

[148] W. Stallings, *Operating Systems: Internals and Design Principles*, 5th ed. Prentice Hall, 2004.

[149] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real-rate scheduling," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 145–158.

[150] Steiner Associates LLC., "Amazon surpasses ebay's unique traffic in december," www.auctionbytes.com, February 2009.

[151] M. Stonebraker, G. Held, E. Wong, and P. Kreps, "The design and implementation of ingres," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 189–222, 1976.

[152] Sun Microsystems, Inc., "Ultrasparc t2 processor," www.sun.com.

[153] ——, "Multithreaded application acceleration with chip multithreading (CMT), multicore-multithreaded UltraSPARC processors," www.sun.com, August 2008, white paper.

[154] ——, "Java EE," http://java.sun.com/, 2009.

[155] R. Swinburne, "Intel Corei7 - Nehalem Architecture Dive," www.bit-tech.net, November 2008.

[156] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 47–58.

[157] X. Tang, J. Xu, and S. T. Chanson, *Web Content Delivery*, ser. Web Information Systems Engineering and Internet Technologies Book Series. Springer Science+Bussiness Media, Inc., 2005, vol. 2.

[158] The Nielsen Company, "Nielsen Online announces december U.S. search share rankings," http://www.nielsen-online.com, January 2009.

[159] The PHP Group, "PHP: Hypertext Preprocessor," http://www.php.net/, 2009.

[160] W. Tianju, B. Koo, L. Zhihong, and H. Ke, "Simsync: A table-based constraint processing language for synchronization control," in *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, Aug. 2009, pp. 691–695.

[161] J. Torrellas, A. Tucker, and A. Gupta, "Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 24, no. 2, pp. 139–151, 1995.

[162] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *ecs'07: Experimental computer science on Experimental computer science*. Berkeley, CA, USA: USENIX Association, 2007, pp. 3–3.

[163] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading pentium 4 processor," in *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2003, p. 26.

[164] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*. New York, NY, USA: ACM, 1998, pp. 533–544.

[165] B. Urgaonkar and P. Shenoy, "Cataclysm: policing extreme overloads in internet applications," in *WWW '05: Proceedings of the 14th international conference on World Wide Web*. New York, NY, USA: ACM, 2005, pp. 740–749.

[166] R. Vaswani and J. Zahorjan, "The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 26–40, 1991.

[167] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2003.

[168] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," in *Proceedings of the Nineteenth Symposium on Operating System Principles (SOSP)*, October 2003.

[169] M. Welsh, "NBIO: Nonblocking I/O for Java," www.eecs.harvard.edu, July 2002.

[170] M. Welsh and D. Culler, "Virtualization considered harmful: Os design directions for well-conditioned services," in *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. Washington, DC, USA: IEEE Computer Society, 2001, p. 139.

[171] ——, "Overload management as a fundamental service design primitive," in *EW 10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*. New York, NY, USA: ACM, 2002, pp. 63–69.

[172] ——, "Adaptive overload control for busy internet servers," in *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 2003.

[173] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," in *SOSP '01: Proceedings of the*

eighteenth ACM symposium on Operating systems principles. New York, NY, USA: ACM, 2001, pp. 230–243.

[174] M. Welsh, "An architecture for highly concurrent, well-conditioned internet services," Ph.D. dissertation, 2002.

[175] Wikimedia Foundation, Inc., "Locality of reference," www.wikipedia.org, Last accessed, Nov 2009.

[176] ——, "Cache coherence," http://en.wikipedia.org/, May 2010, last accessed.

[177] ——, "List of device bit rates," http://en.wikipedia.org/, August 2010, last Accessed.

[178] ——, "Server (computing)," http://en.wikipedia.org/, August 2010, last Accessed.

[179] ——, "Time Stamp Counter," http://en.wikipedia.org/, August 2010, last Accessed.

[180] ——, "Web 2.0," http://www.wikipedia.org/, August 2010, last Accessed.

[181] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.

[182] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and M. F. Kaashoek, "Multiprocessor support for event-driven programs," in *USENIX Annual Technical Conference, General Track*, 2003, pp. 239–252.

[183] Zeus Technology Limited, "Zeus Web Server," http://www.zeus.com/, 2008.

[184] J. Zhou and T. Yang, "Selective early request termination for busy internet services," in *WWW '06: Proceedings of the 15th international conference on World Wide Web*. New York, NY, USA: ACM, 2006, pp. 605–614.

[185] X. Zhou, Y. Cai, J. Wei, and C.-Z. Xu, "A robust application-level approach for responsiveness differentiation," in *ICWS '05: Proceedings of the IEEE International Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 373–380.