ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

"Virtualization is the creation of a virtual version of something, such as an operating system, a server, a storage device or network resources."

La trahison des images

René Magritte (1898-1967).
1928-1929. Oil on canvas. 62.2 x 81cm
"This is not a pipe"

Adaptive Execution Environments for Application Servers

By David Carrera

Advisors:
Jordi Torres
Eduard Ayguadé

A dissertation submitted in partial fulfillment of the requirements for the degree of:

Doctor per la Universitat Politècnica de Catalunya

Barcelona (Spain)
2008

Technical University of Catalunya

Abstract

The growth experienced by the web and by the Internet over the last years has fuelled web application servers to break into most of the existing distributed execution environments. Web application servers take distributed applications one step forward in their accessibility, easiness of development and standardization, by using the most extended communication protocols and by providing rich development frameworks.

Once the initial stages of technology standardization and market penetration were completed, the wide acceptation of application servers as an effective and efficient platform to deploy distributed applications has brought new challenges to the application servers market. Customers come with increasing requirements about performance, robustness, availability and manageability; and at the same time the applications that they want to deploy on the application servers introduce unprecedented levels of complexity and resource demand.

Modern-day companies run a large number of applications, including transactional web applications as well as batch processes. Many organizations rely on such a set of heterogeneous applications to deliver critical services to their customers and partners. For example, in financial institutions, transactional web workloads are used to trade stocks and query indices, while computationally intensive non-interactive workloads are used to analyze portfolios or model stock performance. Due to intrinsic differences among these workloads, today they are typically run on separate dedicated hardware and managed using workload specific management software. Such separation adds to the complexity of data center management and reduces the flexibility of resource allocation. Therefore, organizations demand management solutions that permit these kinds of workloads to run on the same physical hardware and be managed using the same management software. Virtualization technologies can play an important role in this transition; running applications inside of virtual containers simplifies enormously the complexity of management and deployment associated to a datacenter. And even more, opens up a new scenario in which both transactional and long-running workloads can be collocated in the same set of machines with performance isolation guarantees whereas

more of the data center capacity is exploited.

Following the evolution of the application server execution environment, the factors that determine their performance have evolved too, with new ones that have come out with the raising complexity of the environment, while the already known ones that determined the performance in the early stages of the application server technology remain relevant in modern scenarios. In the old times, the performance of an application server was mainly determined by the behavior of its local execution stack, what usually resulted to be the source of most performance bottlenecks. Later, when the middleware became more efficient, more load could be put on each application server instance and thus the management of such a large number of concurrent client connections resulted to be a new hot spot in terms of performance. Finally, when the capacity of any single node was exceeded, the execution environments were massively clusterized to spread the load across a very large number of application server instances, what meant that each instance was allocated a certain amount of resources. The result of this process is that even in the most advanced service management architecture that can be currently found, 1) understanding the performance impact caused by the application server execution stack, 2) efficiently managing client connections, and 3) adequately allocating resources to each application server instance, are three incremental steps of crucial importance in order to optimize the performance of such a complex facility. And given the size and complexity of modern data centers, all of them should operate automatically without need of human interaction.

Following the three items described above, this thesis contributes to the performance management of a complex application server execution environment by 1) proposing an automatic monitoring framework that provides a performance insight in the context of a single machine; 2) proposing and evaluating a new architectural application server design that improves the adaptability to changing workload conditions; and 3) proposing and evaluating an automatic resource allocation technique for clustered and virtualized execution environments. The sum of the three techniques proposed in this thesis opens up a new range of options to improve the performance of the system both off-line (1) and on-line (2 and 3).

This work has been partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01, TIN2004-07739-C02-01 and TIN2007-60625, and by the BSC-IBM collaboration agreement SoW Adaptive Systems. I acknowledge the European Center for Parallelism of Barcelona (CEPBA) and the Barcelona Supercomputing Center (BSC) for supplying the computing resources for some of the experiments presented in this thesis.

Agraïments

El 31 d'agost de 2001 a les 12:44, ara fa uns 6 anys i mig, vaig escriure un correu electrònic al Jordi Torres per tal de mostrar-li el meu interès en un projecte de final de carrera que portava per títol "MILLORA DEL RENDIMENT D'APLICACIONS PARAL.LELES I DISTRIBUIDES EN JAVA". En aquell moment no tenia molt clares les implicacions d'aquell missatge, tot i que ja llavors afegia "*(. . .) ja fa temps que em volta pel cap la possibilitat de, un cop acabada la carrera, quedar-me a la universitat fent recerca i obtenir algun dia el doctorat (. . .)*".

Encara a principis de 2008 veig aquella llunyana data com un punt d'inflexió a la meva vida. El doctorat toca el seu final en aquests moments, i darrera meu queden uns quants anys de formació com a investigador. Aquella decisió significava, en aquell moment, que pretenia passar uns quants anys més en l'entorn universitari per a complementar uns estudis superiors que no m'havien semblat suficientment desafiants abans de tornar al mercat dur de l'empresa privada amb una formació suficientment potent com per a fer coses d'entitat. Però el que realment va significar aquella decisió és que vaig quedar immers en el món acadèmic, des d'on m'ensumava que les grans coses a fer al sector industrial nacional no eren tan importants com imaginava. Aquesta sensació em va anar allunyant cada cop més del pensament de tornar al mercat privat, i em va fer descobrir que la recerca acadèmica també podia ser estimulant així com també m'ho va semblar la docència. Finalment vaig descobrir que fer recerca des de la universitat, però de la mà d'empreses privades punteres, pot combinar el millor dels dos móns exceptuant, per suposat, els salaris alts.

Com és natural, durant aquests anys m'he trobat amb molta gent, alguns m'han ajudat i d'altres m'han dificultat la feina, conscient o inconscientment. Allò més important que he après en aquest temps és que aquells que més m'han ajudat a millorar com a investigador són aquells que més dificultats m'han generat durant el camí: haver de lluitar per les teves idees fa que te les miris per tots els costats i amb ulls més crítics que si totes et venen de cara. Això no treu mèrit als que explicitament m'han ajudat, i per això a ells els dedico aquests agraïments.

Primerament, la meva família. Els valors que em van transmetre en el seu moment és el que més valoro de tota la meva infància, i és l'única raó per la que té sentit esforçar-se continuament per millorar. Sempre han estat al meu costat i m'han ensenyat que donar tot el que tens per allò que creus és el millor camí per trobar la felicitat. Es mereixen aquest reconeixement públic i molt més!

Segonament, la Mireia. Aquesta criatura preciosa al voltant de la qual gira tota la meva vida

i amb qui, tal com sempre m'agrada recordar, el dia que tinguem 32 anys ja haurem compartit el 100% del 50% de la nostra vida. Sense ella res del que he fet en els últims anys tindria massa sentit, i espero que estar amb mi hagi donat sentit a la seva vida igualment, perquè vull seguir amb ella per molt i molt de temps!

Tercerament, als meus directors de tesis: dos caràcters totalment diferenciats amb una bona amistat que els uneix. El Jordi, emprenedor i hiperactiu de mena, sempre pensant en el demà mentres deixa enrera el dia d'avui a tota pressa. I l'Eduard, més reflexiu i pensatiu, mirant el camí que deixem enrera per a entendre el que vindrà. Gràcies a tots dos per la part que us toca.

Tots els *eDragons* també mereixen una menció especial, una raça especial d'investigadors coneguts allà on anem, o no! En qualsevol cas, el millor equip de gent amb qui treballar. Tal com va dir la Gosia un dia, els bons manager són els que es saben envoltar de la gent que millor treballa, i els nostres ho han fet molt bé. Especialment vull mencionar el Vicenç, amb qui més hores hem treballat colze amb colze, compartint dinars de divendres i matinades escribint articles brillants, i amb qui, sens dubte, més cops hem arribat a raonaments absurds per a explicar resultats incomprensibles.

També vull mencionar la Gosia Steinder i l'Ian Whalley, juntament amb la resta de gent d'IBM Watson amb qui he treballat els últims anys. Crec que són els investigadors més brillants que he conegut fins el moment, i són l'estímul professional més gran per a continuar fent recerca.

Finalment, a tota la gent que ha viscut com jo aquests anys. Des dels inicis de la sala CEPBA on ens vam criar junts, fins als que compartim menjador diàriament. El Juanjo i l'Alex, els *bloggers* rivals, el Xavi i el Marc, els cronistes de l'actualitat i "la razón", el Víctor, el Norbert, el Carles, i el Sergio, els DB's del menjador, el Rogeli i el Raúl, amb qui vam compartir orígens, i també tots els administradors de sistemes, juguin o no a badminton.

Especialment vull remarcar la Montse, la "*roommie*", amb qui comparteixo més hores de despatx que ningú, però que m'ha fet la vida molt fàcil durant tot aquest temps, tant a Barcelona com a New York. També la Yolanda, una companyia tan agraïda com agradable, amb qui sempre es pot confiar, capaç de deixar-ho tot per donar ànims a qui sigui i tota una personalitat a conèixer i disfrutar. El Pau també ha tingut un paper important en aquest temps, company de hobbies i musicòleg incorregible, m'ha fet conèixer tota una *amalgama* de móns nous.

Acknowledgments

On August 31st of 2001, 12:44pm, around 6 years and half ago, I wrote an e-mail to Jordi Torres in order to show my interest in a master thesis project titled "IMPROVEMENTS IN THE PERFORMANCE OF PARALLEL AND DISTRIBUTED JAVA APPLICATIONS". At that moment I didn't know the implications of that message, even though I already added "*(...) I've been thinking for long time about the possibility of continuing in the university once graduated to do research and to obtain a PhD at some point (...)* ".

At the beginning of 2008, I still see that moment as an inflection point in my life. I'm nearly done with my PhD now, and behind me I leave a few years of research training. That decision meant, at that moment, that I wanted to spend a couple of years in the university to extend the formation I received during my not enough challenging master studies, before entering the industry to carry on important projects. But what that decision really meant is that I started doing academic research, while I suspected that industrial research in Spain had not much to offer. That feeling pulled me away from my idea of working in the industry, and made me discover that the academic research and teaching are exciting tasks I really enjoy. At the end of the day, I found that doing research in the university, but in collaboration with top companies, is a good way to combine industrial and academic research. Not to mention, the salary remains at university standards.

Obviously, over last years I met many people, some of them helped me, others not. A lesson I learned about this is that the more people makes the things hard for you, the more you improve as a researcher: fighting to defend your ideas makes you think more carefully about them. But it doesn't mean that people that did actually help me don't deserve my public acknowledgment, so this is for all of you that supported me over last years.

Firstly, my family. The values that I learned in my childhood are my most valuable treasure now, and it is the main thing that makes me keep pushing to constantly improve. They have been always supporting me and they taught me that fighting for the things what you believe is the best way to reach happiness. They deserve this public acknowledgment and much more!

Secondly, Mireia. This precious creature what my whole life revolves around. As I like to remind, once we'll be 32 years old, we will have shared a 100% of the 50% of our whole life. She's the reason my life makes sense, as I hope her life makes sense because of me, and I want to be with her years from now!

Thirdly, my PhD advisors: two totally different personalities that share a good friendship.

Jordi, entrepreneurial and hyperactive, always thinking about tomorrow while quickly forsakes today. And Eduard, more reflexive and pensive, always looking back to plan ahead. Thank you for your contributions.

The *eDragon* team also deserves my acknowledgment. A research team of proud lineage, well known wherever we go, or maybe not! Anyway, the best possible team to work with. As Gosia said time ago, good managers seek best researchers to be in their team. We had the best possible ones. I want to mention Vicenç, a hard worker with who I have lunch on fridays, write brilliant papers beyond midnight, and too often reach absurd reasonings that explain uncomprehensible results.

Also, I want to mention Gosia Steinder and Ian Whalley, as well as the other people from IBM Watson I've worked with in the last years. They're the most brilliant researchers I've ever met, and a source of professional encouragement to continue doing research. Thanks for your time and dedication.

Finally, all my colleagues in the university. From the guys that "lived" in the CEPBA lab with me, to the people who I have lunch with regularly. Juanjo and Alex, the rival *bloggers*; Xavi and Marc, who enjoy discussing news and reading "La razón"; Victor, Norbert, Carles, and Sergio, the DB's of the dining room; Rogeli and Raul, they've been there since the very beginning; and not to mention all the sysadmins, even if they don't play badminton.

Specially, I want to mention Montse, my "*roommie*", my official roommate that shares hours in the office with me, and a person that makes things easy, in Barcelona as well as in New York. Yolanda, pleasant and grateful friend, somebody to trust, always ready to encourage you. And Pau, a great hobby mate and a music lover, he shown me an *amalgam* of new trends.

Contents

Abstract **i**

Agraïments **v**

Acknowledgments **ix**

Table of contents **xvi**

List of Figures **xxi**

List of Tables **xxiii**

1 Introduction **1**
1.1 Contributions 5
1.1.1 Automatic performance monitoring framework 5
1.1.2 Adaptive architecture for application servers 7
1.1.3 Integrated management of heterogenenous workloads 9
1.2 Thesis Organization 11

2 Execution environments for application servers **13**
2.1 Web Applications, Web containers and Application Servers 15
2.1.1 HTTP protocol 15
2.1.2 Web contents and web applications 16
2.1.3 Java Servlets, Java Server Pages and Java 2 Enterprise Edition . . 17
2.2 Workload generators and benchmarking web applications 18
2.2.1 Benchmarking web applications 18
2.2.2 Workload generator: *httperf* 19
2.3 Utility-driven resource management in virtualized environemnts 19
2.3.1 Utility functions 20
2.3.2 Virtualization technology 21

3 Automatic performance monitoring framework **25**
3.1 Introduction 27
3.2 Components 29

3.2.1 Java Instrumentation Suite (JIS) 29
3.2.2 Paraver 29
3.2.3 Java Automatic Code Interposition Tool (JACIT) 30
3.3 Implementation of JIS Linux-IA32 31
3.3.1 Operating system level 32
3.3.1.1 Kernel module 32
3.3.1.2 Kernel patch 34
3.3.2 Java Virtual Machine level 35
3.3.3 Middleware and User application levels 37
3.3.4 Merging data 38
3.3.5 Overheads in the Linux-IA32 implementation 38
3.4 Automatic monitoring 40
3.4.1 Monitoring high-level performance metrics 40
3.4.2 Automatic management of the monitoring infrastructure 41
3.4.3 Case study 42
3.5 Related work 44
3.6 Summary 45

4 Adaptive architecture for application servers 49
4.1 Introduction 51
4.2 Application server architectures 53
4.2.1 Multithreaded architecture with blocking I/O 53
4.2.2 Event-driven architecture with non-blocking I/O 54
4.3 Performance characterization of secure web applications 55
4.3.1 Secure workloads 55
4.3.2 Evaluation platform 57
4.3.3 Scalability Characterization 58
4.3.3.1 Exploring scalability 58
4.3.3.2 Analyzing scalability limits 60
4.4 Hybrid Architecture 65
4.4.1 Implementation on top of Tomcat container 65
4.4.2 Performance evaluation 67
4.4.2.1 Testing platform 67
4.4.2.2 Static content 68
4.4.2.3 Dynamic content 70
4.5 Related Work 73
4.6 Summary 74

5 Integrated management of heterogeneous workloads 77
5.1 Introduction 79
5.2 System architecture 82
5.3 The placement problem 86
5.3.1 Problem statement 86

5.3.2 Algorithm outline 88
5.3.2.1 Placement change method 89
5.3.2.2 Capping application demand 90
5.3.2.3 Maximizing load distribution 90
5.4 Characterization of heterogeneous workloads 91
5.4.1 Transactional workloads 91
5.4.1.1 Calculating application utility 91
5.4.2 Long running workloads 93
5.4.2.1 Job characteristics 94
5.4.2.2 Stage aggregation in a control cycle 95
5.4.2.3 Maximum achievable utility 96
5.4.2.4 Hypothetical utility 97
5.5 Prototype implementation 104
5.5.1 VM management 104
5.5.2 Job management 107
5.5.3 Xen machine organization 108
5.6 Evaluation in a simulator 109
5.6.1 Transactional-only workloads 109
5.6.1.1 Generation of utility functions in the simulator 110
5.6.1.2 Evaluation criterion: minimum utility 110
5.6.1.3 Evaluation criterion: number of placement changes . . 112
5.6.1.4 Evaluation criterion: optimality 112
5.6.2 Long running-only workloads 115
5.6.2.1 Experiment One: Hypothetical utility 115
5.6.2.2 Experiment Two: Baseline 118
5.6.2.3 Experiment Three: Variable deadlines 120
5.6.2.4 Experiment Four: Randomized jobs 123
5.6.3 Heterogeneous workloads 127
5.7 Evaluation in the prototype 131
5.7.1 Transactional-only workloads 131
5.7.1.1 Baseline experiment 132
5.7.1.2 Benefits of a utility-based placement 132
5.7.2 Long running-only workloads 135
5.7.3 Heterogeneous workloads 138
5.8 Related Work 140
5.9 Summary 143

6 Conclusions and future work **145**
6.1 Conclusions 147
6.1.1 Automatic performance monitoring framework 147
6.1.2 Adaptive architecture for application servers 148
6.1.3 Integrated management of heterogenenous workloads 149
6.2 Future work 151

Bibliography 153

List of Figures

1.1 Summary of contributions 5

2.1 Evolution of utility for long running jobs 21

3.1 JIS compared to other monitoring and tracing tools 28
3.2 JIS architecture 30
3.3 JACIT screenshot 31
3.4 JIS instrumentation process 32
3.5 Thread states considered by JIS and intercepted functions to detect transitions 33
3.6 CPU intensive application overhead results 39
3.7 I/O intensive application overhead results 39
3.8 WAS Control center operation diagram for automatic system tracing . . . 42
3.9 View of the WAS Control Center GUI 43
3.10 Observed response time for a servlet of the RUBiS Benchmark 44

4.1 Operation of a multithread architecture 54
4.2 Tomcat persistent connection pattern 55
4.3 Throughput of the original Tomcat with different numbers of processors . 56
4.4 Response time of the original Tomcat with different numbers of processors 59
4.5 Completed sessions by the original Tomcat with different numbers of processors 60
4.6 Average time spent by the server processing a persistent client connection 61
4.7 Incoming server connections classification depending on the SSL handshake type performed 62
4.8 Throughput of Tomcat with overload control with different numbers of processors 63
4.9 Completed sessions by Tomcat with overload control with different numbers of processors 64
4.10 Operation of the hybrid architecture 66
4.11 Throughput comparison under an static content workload 68
4.12 Response time under an static content workload 69
4.13 Number of connections closed by the server by a timeout expiration . . . 69
4.14 Reply throughput comparison under a dynamic content workload 71

4.15 Successfully completed session rate under a dynamic content workload . 71
4.16 Lifetime comparison for the sessions completed successfully under a dynamic content workload 72

5.1 Architecture of the system 82
5.2 Real utility function that corresponds to a transactionl application 92
5.3 Evolution of maximum achievable utility for long running jobs 96
5.4 Allocation as a function of target utilities 98
5.5 Estimating $\tilde{\omega}_m$ using $ratio_g$ 100
5.6 Estimating $\tilde{\omega}_m$ using $ratio_g$ and app_ratio_m 101
5.7 Hypothetical utility: effect of resource competition 102
5.8 Management architecture for Xen machines. 105
5.9 Life-cycle of a Xen domain. 106
5.10 Example of utility functions 111
5.11 Maximizing minimum utility across applications 113
5.12 Minimizing placement changes: generated workload 114
5.13 Minimizing placement changes 114
5.14 Experiment One: Description 117
5.15 Experiment Two: Jobs in the system and jobs placed 119
5.16 Experiment Two: Average hypothetical utility over time and actual utility achieved at completion time 120
5.17 Experiment Two: Algorithm execution time 121
5.18 Experiment Three: jobs in the system and jobs placed 122
5.19 Experiment Three: average hypothetical utility over time and actual utility achieved at completion time 122
5.20 Experiment Three: total number of virtualization operations over time . . 123
5.21 Experiment Four: Percentage of jobs that met the deadline 124
5.22 Experiment Four: Number of virtualization operations 124
5.23 Experiment Four: distribution of distance to the goal at job completion time, for five different mean interarrival times (50s to 400s) 126
5.24 Heterogenenous workload: utility function for the transactional workload (utility as a function of allocated CPU power) 127
5.25 Heterogenenous workload: CPU power allocated to each workload and CPU demands to achieve maximum utility 128
5.26 Heterogenenous workload: actual relative performance for the transactional workload and average calculated hypothetical relative performance for the long-running workload 129
5.27 Heterogenenous workload: CPU power allocated to each workload for the three system configurations 130
5.28 Demand 133
5.29 Response time 134
5.30 Utility 135
5.31 Per node allocation 136

5.32 Node utilization by long running jobs. 137
5.33 Response time for *StockTrace* and job placement on nodes. 139

List of Tables

4.1 Number of clients that overload the server and maximum throughput achieved before overload occurs 57
4.2 Average server's throughput when overloaded 58

5.1 Cost of virtualization operations 115
5.2 Properties of Experiment One 116
5.3 Properties of Experiment Two 119
5.4 Node properties 131
5.5 Application properties 132
5.6 Jobs used in experiments 136

Chapter 1

Introduction

The growth experienced by the web and by the Internet over the last years has fueled web application servers to break into most of the existing distributed execution environments. Web application servers take distributed applications one step forward in their accessibility, easiness of development and standardization, by using the most extended communication protocols and by providing rich development frameworks.

Once the initial stages of technology standardization and market penetration were completed, the wide acceptation of application servers as an effective and efficient platform to deploy distributed applications has brought new challenges to the application servers market. Customers come with increasing requirements about performance, robustness, availability and manageability; and at the same time the applications that they want to deploy on the application servers introduce unprecedented levels of complexity and resource demand.

The execution environment of application servers has evolved in the last years, both in terms of complexity and size, and also the difficulty to understand and optimize their performance. The number of APIs and features offered by the application server execution stack has grown dramatically, easening the development of web applications but also impacting on their performance. And the execution platform has evolved from single machine environments, to moderately small clusters that eventually became large and shared data centers.

In addition, modern-day companies run a large number of applications, including transactional web applications as well as batch processes. Many organizations rely on such a set of heterogeneous applications to deliver critical services to their customers and partners. For example, in financial institutions, transactional web workloads are used to trade stocks and query indices, while computationally intensive non-interactive workloads are used to analyze portfolios or model stock performance. Due to intrinsic differences among these workloads, today they are typically run on separate dedicated hardware and managed using workload specific management software. Such separation adds to the complexity of data center management and reduces the flexibility of resource allocation. Therefore, organizations demand management solutions that permit these kinds of workloads to run on the same physical hardware and be managed using the same management software. Virtualization technologies can play an important role in this transition; running applications inside of virtual containers simplifies enormously the complexity of management and deployment associated to a datacenter. And even more, opens up a new scenario in which both transactional and long-running workloads can be collocated in the same set of machines with performance isolation guarantees whereas more of the data center capacity is exploited.

This new scenario puts on the table a new challenge for the researchers: the management of complex and resource demanding applications running on large and complex clustered execution environments. This challenge adds a new focus of attention to the computer science: the management of the complexity of the systems, what has become, by itself, a new research area. The answer given by the research community to the complexity management problem is a new computing paradigm that focuses on the introduction of autonomic functions into the complex systems: the autonomic computing. It defines a number of characteristics that any system should meet to be able to overcome its complexity, including the capacity to know itself, to configure and reconfigure itself, to self-protect, to recover itself from errors and prevent new incidences and to manage its interaction with other systems.

Following the evolution of the application server execution environment, the factors that determine their performance have evolved too, with new ones that have come out with the raising complexity of the environment, while the already known ones that determined the performance in the early stages of the application server technology remain relevant in modern scenarios. In the old times, the performance of an application server was mainly determined by the behavior of its local execution stack, what usually resulted to be the source of most performance bottlenecks. Later, when the middleware became more efficient, more load could be put on each application server instance and thus the management of such a large number of concurrent client connections resulted to be a new hot spot in terms of performance. Finally, when the capacity of any single node was exceeded, the execution environments were massively clusterized to spread the load across a very large number of application server instances, what meant that each instance was allocated a certain amount of resources. The result of this process is that even in the most advanced service management architecture that can be currently found, 1) understanding the performance impact caused by the application server execution stack, 2) efficiently managing client connections, and 3) adequately allocating resources to each application server instance, are three incremental steps of crucial importance in order to optimize the performance of such a complex facility. And given the size and complexity of modern data centers, all of them should operate automatically without need of human interaction.

Following the three items described above, this thesis contributes to the performance management of a complex application server execution environment by 1) proposing an automatic monitoring framework that provides a performance insight in the context of a single machine; 2) proposing and evaluating a new architectural application server design that improves the adaptability to changing workload conditions; and 3) proposing and evaluating an automatic resource allocation technique for clustered and virtualized

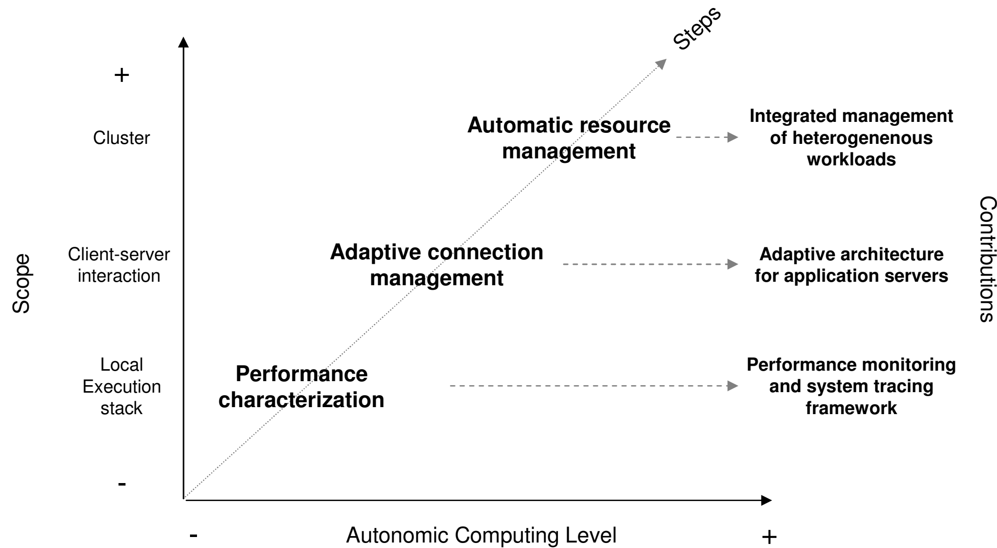


Figure 1.1 Summary of contributions

execution environments. The sum of the three techniques proposed in this thesis opens up a new range of options to improve the performance of the system both off-line (1) and on-line (2 and 3). Figure 1.1 summarizes the main ideas presented in this thesis.

1.1 Contributions

1.1.1 Automatic performance monitoring framework

The execution stack of a complex middleware such a modern application server involves many components and APIs that directly impact on the overall performance offered by the server. A deep knowledge of the layers that compose the execution stack is a key point in order to characterize the performance of an application server. But a detailed analysis of such a complex environment can only be achieved by combining 1) the adequate tools to monitor in detail the behavior of the system over time, and 2) the certainty that the system will be monitored at the exact moment a performance issue comes out.

The **first contribution** of this thesis is the creation of an automatic deep monitoring framework that produces extremely detailed insight on the system behavior – ranging from the operating system level to the user application code. Java Instrumentation Suite (JIS) is the monitoring tool developed for this thesis. Its novelty resides in the fact that it collects information of all the levels of the execution stack of a J2EE application server at runtime,

and correlates this information to produce a global view of the state and performance of the application server. This information can be processed with the adequate analysis and visualization tools, Paraver[38] in this particular case, to perform in-depth post-mortem performance studies.

The monitoring framework can operate automatically, without need of human interaction in the process of data acquisition, whereas it is driven by some user-defined rules. Such a monitoring framework operates by continuously observing some high-level performance metrics delivered by the application server, and triggering the in-depth tracing process of the whole application server execution stack when it is observed that some minimum user-defined performance objectives are not met. System administrators and software developers can take especial advantage of an automatic monitoring tool such as the one proposed in this work, especially if they run their complex systems under high-availability requirements that force them to keep their environments up and running 24x7. The automatization of the monitoring tool has been developed to work with Websphere Application Server, and the resulting automatic monitoring environment has been named WAS Control Center.

The performance monitoring framework has been applied to study the performance of several execution environments, such as those described in [22], [44], [88], [46] and [77]. In particular, the tool was applied to study and characterize the performance of application servers running secure applications, what resulted a successful way to diagnose a connection management problem present in the architectural design of most application servers, which was successfully addressed and is now presented as the second contribution of this thesis.

The work performed in this area has resulted in the following publications:

[23] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, and J. Labarta. **Complete instrumentation requirements for performance analysis of web based technologies**. In Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), 2003

[21] D. Carrera, D. García, J. Torres, E. Ayguadé, and J. Labarta. **WAS Control Center: An autonomic performance-triggered tracing environment for Websphere**. In Proceedings of 13th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP'05), 2005

as well as resulted in the following derived work:

[44] J. Guitart, V. Beltran, D. Carrera, J. Torres and E. Ayguadé. **Characterizing Secure Dynamic Web Applications Scalability**. 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), April 2005

[45] J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. **Designing an overload control strategy for secure e-commerce applications**. Computer Networks 51, 15 (Oct. 2007), 4492-4510

[22] D. Carrera, J. Guitart, V. Beltran, J. Torres and E. Ayguadé. **Performance Impact of the Grid Middleware**. In Engineering the Grid: Status and Perspective, American Scientific Publishers, January 2006. ISBN: 1-58883-038-1

[88] R. Nou, F. Julià, D. Carrera, K. Hogan, J. Labarta, J. Torres. **Monitoring and analysis framework for grid middlewares**. In Proceedings of 15th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'07), 2007

[46] J. Guitart and D. Carrera and J. Torres and E. Ayguadé and J. Labarta, **Tuning Dynamic Web Applications using Fine-Grain Analysis**, Proceedings of 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05)

1.1.2 Adaptive architecture for application servers

The piece of code that isolates the core of an application server from the network protocols used to interact with the clients is known as the a network connector. A connector can be created to use any existing network protocol, being the HTTP connector the most extended version. In most application servers, the threading model used to support concurrency in the request processing pipeline is defined within the network connector.

Most web application servers use the well-known multithread paradigm to manage client connections, with an ideal model for which one client is associated to a server thread all over the client lifecycle. The main concern setting up a multithreaded network connector is to decide an adequate number of threads to be created in order to efficiently manage all the active clients without introducing an unnecessary overhead into the system. Running more threads than necessary results in an extra management cost, while running out of threads means that many incoming connections will be refused. The second most important concern when adjusting the configuration of such a multithreaded server is to decide how long an inactive connection will be kept alive (the keep-alive timeout). Setting up a too high value for the keep-alive timeout may result in underutilization of the system resources and many incoming connection requests being rejected. On the other hand, too

short timeout values result in a higher number of clients being allowed to enter the system, but at a cost of an excessive connection management overhead.

The workload to which an application server is subject is not only dynamic in terms of intensity over time, but also in the mix of resources requested. This dynamism, added to the inherent complexity of an application server, makes the task of adapting the configuration of the application server to the workload conditions a hard challenge. And it becomes even more complex if the performance and service level of the application server is measured in terms of high level performance goals instead of low level objectives. An alternative to the multithread architecture is an event-driven model. This model solves some of the problems present in the multithreaded architecture but transforms the development of the web container into a hard task.

In this work we introduce a new hybrid server architecture that exploits the best of both the multithread and the event-driven models. With this hybrid schema, an event-driven model is applied to manage the client connections, although a multithreaded model is followed to process requests. This combined use of the two basic architectures results in a more efficient connection management without an increased complexity in the development of application server middleware. The design of such a new architecture is motivated by a preliminary study of Tomcat's vertical scalability when subject to secure workloads. The results confirmed that the server can be easily overloaded if connections are not properly managed, demonstrating the convenience of developing advanced connection management strategies to overcome such a complicated scenario.

In summary, the **second contribution** of this thesis is the proposal of a new network connector architecture that eliminates the need of setting up neither the number of threads in the system nor the keep-alive timeout to deliver a good performance, what is achieved because of its adaptive design. The proposed architecture makes combined use of the multithread paradigm alongside with the best properties of an event-driven architecture based on non-blocking input/output operations. The result is a new design of the network connector that brings an unprecedented natural adaptability to the workload conditions without need of manual tuning. In our work we have focused on a HTTP network connector because this is the most extended network protocol used by application servers, but the proposal is extensible to any existing network connector.

The work performed in this area has resulted in the following publications:

[19] D. Carrera, V. Beltran, J. Torres, E. Ayguade. **A Hybrid Web Server Architecture for e-Commerce Applications**. 11th International Conference on Parallel

and Distributed Systems (ICPADS'05), 2005

[20] D. Carrera, V. Beltran, J. Torres, E. Ayguade. **A Hybrid Connector for Efficient Web Servers**. Special Issue on High Performance Computing in Parallel and Distributed Systems of the International Journal of High Performance Computing and Networking (IJHPCN). Issue 5/6 of 2007, Vol. 5. ISSN: 1740-0562. *To appear*.

has been motivated by the work described in:

[45] J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. **Designing an overload control strategy for secure e-commerce applications**. Computer Networks 51, 15 (Oct. 2007), 4492-4510

and has resulted in the following derived work:

[13] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. **Evaluating the Scalability of Java Event-Driven Web Servers**. In 2004 International Conference on Parallel Processing (ICPP'04), 2004

[12] V. Beltran , D. Carrera, J. Guitart, J. Torres and E. Ayguadé. **A Hybrid Web Server Architecture for Secure e-Business Web Applications**. 1st International Conference on High Performance Computing and Communications (HPCC'05), 2005 Lecture Notes in Computer Science, pp. 366-377, vol. 3726, no. 3726, September 2005. ISSN: 0302-9743 ISBN: 978-3-540-29031-5.

[14] V. Beltran , J. Torres and E. Ayguadé. **Understanding Tuning Complexity in Multithreaded and Hybrid Web Servers**. 22nd International Parallel and Distributed Symposium (IPDPS'08), 2008

1.1.3 Integrated management of heterogenenous workloads

Modern-day companies run a large number of applications, including transactional web applications as well as batch processes. Many organizations rely on such a heterogeneous set of applications to deliver critical services to their customers and partners. For example, in financial institutions, transactional web workloads are used to trade stocks and query indices, while computationally intensive non-interactive workloads are used to analyze portfolios or model stock performance.

The resource demand associated to transactional applications exceeded by far the capacity of any available single machine time ago already. The extra capacity required was

found by horizontally scaling-up the execution environments, and thus the most extended execution environment for resource-demanding web applications is massively clustered environments. At the same time, batch processes usually run in isolation conditions within dedicated machines.

In an attempt to save energy and management complexity, most companies are taking the challenge of consolidating their hardware facilities into large data centers. As a result, workloads (both transactional and long-running) are being consolidated too. Due to intrinsic differences among these workloads, today they are typically run on separate dedicated hardware and managed using workload specific management software. Such separation adds to the complexity of data center management and reduces the flexibility of resource allocation. Therefore, organizations demand management solutions that permit these kinds of workloads to run on the same physical hardware and be managed using the same management software.

Virtualization technologies can play an important role in this transition, specially in the area of deployment, update, configuration, and performance and availability management. Many of these challenges are addressed by virtualization, which provide a layer of separation between a hardware infrastructure and workload, and provide a uniform set of control mechanisms for managing these workloads embedded inside virtual containers. Virtualization technologies also enable separation between management concerns, permitting software and configuration tasks inside virtual machines to be done a priori. The runtime management system is only responsible for the runtime performance and availability of virtualized workloads. In a virtualiualized data center both transactional and long-running workloads can be collocated in the same set of machines with performance isolation guarantees whereas more of the data center capacity is exploited.

Two particular issues make the integrated management of heterogeneous workloads specially challenging. First, performance goals for different workloads tend to be of different types. For interactive workloads, goals are typically defined in terms of average or percentile response time or throughput over a certain time interval, while performance goals for non interactive workloads concern the performance (e.g., completion time) of individual jobs. Second, due to the nature of their performance goals and short duration of individual requests, interactive workloads lend themselves to automation at short control cycles. Non-interactive workloads typically require calculation of a schedule for an extended period of time.

The third contribution of this thesis is the proposal of a technique that allows an integrated management of heterogenenous workloads in the context of virtualized

data centers, whereas it observes high-level performance goals. The technique manages transactional workloads (and the corresponding application servers) collocated with long running workloads whereas it achieves equalized satsifaction for both workloads. The objective of the approach is to provide fair differentiation of performance among all workloads in response to varying workload intensities. We use utility functions to model the satisfaction of both long-running jobs and transactional workloads for a particular resource allocation – the different types of workload have different characteristics, and different performance goals, and utility functions offer a mechanism to make their performance comparable. We run both workloads inside virtual machines, in order to properly manage their performance, and our management also exploits the clustering nature of transactional workloads. This the first proposal that combines an explicit support for heterogeneous workloads in virtualized environments, using a utility-driven scheduling mechanism with fairness goals.

The work performed in this area has resulted in the following publications:

[25] D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguadé. **Utility-based Placement of Dynamic Web Applications with Fairness Goals**. In 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), April 7-11, 2008

[24] D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguadé. **Managing SLAs of heterogeneous workloads using dynamic application placement**. In the 7th IEEE Symposium on High Performance Distributed Computing (HPDC 2008). An extended version is available as Technical Report RC24469, IBM Research, Jan. 2008.

[95] M. Steinder, I. Whalley, D. Chess, D. Carrera, I. Gaweda, **Server virtualization in autonomic management of heterogeneous workloads**. 10th IFIP/IEEE International Symposium on Integrated Management (IM 2007), 2007

1.2 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces some basic concepts about web application servers and their associated technologies; about the concepts of resource management, utility-based computing and virtualization; and about the environment and methodology used in our experiments. Chapter 3 presents the automatic performance monitoring and system tracing used to understand the performance

of a single instance of a web application server. Chapter 4 introduces and evaluates a new architectural design of network connector for application servers. Chapter 5 presents and evaluates an integrated management technique for heterogeneous workloads running in virtualized and clusterized data centers. Finally, Chapter 6 presents the conclusions and the future work of this thesis.

Chapter 2

Execution environments for application servers

2.1 Web Applications, Web containers and Application Servers

The World Wide Web (usually known simply as the Web) is composed by a large set of hypertext documents accessible through a set of standard and extended network protocols, including HTTP[41] as the application protocol, TCP[93] as the transport protocol and IP[92] as the network protocol. The contents of the web can be classified into static and dynamic, depending on whenever they are directly accessed by the web clients or if they need to be preprocessed in the server before being sent to the clients.

2.1.1 HTTP protocol

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypermedia information systems. It is the basis for most of the web-based technologies and applications and is one of the most extended application-level protocols used in the Internet.

HTTP defines, in its current version 1.1, a number of operations to retrieve or send resources from or to servers. A resource is defined using a Uniform Resource Identifier (URI). The most usual operations to retrieve resources from a server using HTTP are GET and POST.

In the scope of this thesis, the most interesting properties of the HTTP protocol are those related to the way how HTTP manages the TCP connections used to communicate the client and the server through the network. HTTP/1.1 supports two major models in the management of TCP connections: persistent connections and non-persistent connections. When a HTTP client, usually a browser, wants to send an operation to a HTTP server, usually known as web servers, it has to establish a TCP connection with the server. Once the connection is established, the HTTP request send and the corresponding HTTP response received, the client and the server have to decide whenever the TCP connection is going to be closed or by contrary they keep it established to facilitate future communications. In the case they decide to keep the connection established, it is said that they follow a persistent connection model. If they decide to close the connection after each interaction, it is said that they follow a non-persistent connection model.

The importance of the HTTP connection model is deeply related to the properties present in a workload composed by the aggregation of multiple client-server HTTP interactions. As will be explained in chapter 4, the connection model can have a big impact in the performance obtained by an HTTP server under severe load conditions.

2.1.2 Web contents and web applications

A web application is an application that can be accessed through the common technologies that support the Web, and that is composed by a mix of static and dynamic contents. Web applications have become extremely popular in the last years, and have become the most significant example of distributed applications. This great success can be explained due to the fact that the protocols they rely on are widely extended, which in turn results in a dramatic reduction in the problems that can come out at deployment time, as well the everytime higher familiarity of users with their web browsers.

Web contents can be basically divided depending on their nature as static or dynamic. Static contents are those served to clients without any kind of process. HTML [113] files are the best example of static contents: when requested, they are read from disk and sent to clients directly and without any modification. Dynamic contents are those requiring some process before being sent to clients. Typical dynamic contents are server scripts, which are processed and generated results (typically formatted as a HTML page) are sent to clients.

The development from scratch of a web application is not an easy task: the stack of protocols involved in the operation of this kind of applications is complex and the number of issues that should be resolved is high. For this reason, usually web applications rely on a piece of software that isolates the application from the underlying execution environment. This piece of software is known as middleware.

Depending on the functionalities and support a middleware for web applications offer, it can be classified as a simple web container or as a much more complex application server. A web container is mainly limited to deal with the network protocols that can be used to access the web applications deployed on it. This way, the application developers can focus their efforts on the application logic and let the web container manage the network accessibility of the application.

Web containers proved to be a great platform for the development of many simple web applications, but when more complex applications were to be developed, some extra support from the underlying middleware was expected. A complex web application comprises a large number of technical requirements, such as isolation requirements, interaction with back-end systems and support for asynchronous messaging, that dramatically increase the hardness of its development. Additionally it is supposed to be concurrently accessed by thousands of clients. As far as these requirements are present in most of the complex web applications, their support was directly introduced in the middleware where these applications are usually deployed on. The middleware that hosts complex web applications and that offers a large number of facilities for the deployment and execution

of such applications, is known as an Application Server.

2.1.3 Java Servlets, Java Server Pages and Java 2 Enterprise Edition

The Java [99] platform, developed by Sun Microsystems and appeared in early 1990s, introduced an unprecedented level of portability and easiness of development to the software development arena. The fact that it was based on the use of a virtual execution environment, the Java Virtual Machine (JVM), made Java applications become platform-independent.

The portability, amongst others, is one of the properties that made the Java platform become an interesting option for the development of web applications and the corresponding middleware. Applications and middleware written for the Java platform could potentially be run in any existing hardware platform. Sun microsystems, aware of this situation, developed a number of specifications to define a new set of standards for the development of web applications and middleware.

The first step in the Java technologies for the development of web applications is composed of the Java Server Pages [102] (JSP) and the Java Servlet [101] specifications. A JSP is a HTML document with embedded pieces of Java code. This way when a JSP document is requested, it is first the embedded Java code is pre-processed by the web container where it is hosted and later, the HTML code and the output produced by the execution of the Java code are sent all together to the client. The JSP technology only supports a subset of the common Java features.

The Java Servlet technology defines an API to write simple web applications accessible through the web protocols, and can take advantage of all the Java features. Applications developed following the Servlets technology can easily overcome higher levels of complexity, and introduce an unprecedented level of isolation with respect to the network protocols used to access them.

Apache Tomcat [6] is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and also to be a quality production servlet container. Tomcat can work as a standalone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content).

Some extremely large and complex applications require even more support from the execution platform than what the JSP and Servlet technologies can provide. For this kind of applications, Sun Microsystems developed a set of APIs that provide an even more rich execution framework: the Java 2 Enterprise Edition [98] (J2EE). This development and execution framework provides easy access to many advanced features, such as transaction

and isolation support, asynchronous messaging and database access amongst others. The J2EE platform is widely used for the development of the most large and complex web applications, and several vendors offer advanced products to develop and host these applications.

Enterprise Java Beans (EJB) are more complex objects than servlets and are the basis for porting the object components paradigm to application servers. EJBs allow developers to implement real distributed applications based on the Web easily and rapidly. J2EE compliant Application Servers contain Web containers and EJB containers.

2.2 Workload generators and benchmarking web applications

2.2.1 Benchmarking web applications

Three web applications have been used for benchmarking application servers in this thesis. Surge [9] emulates an static content website. RUBiS [4] emulates a dynamic auction website. Both of them include not only the web application but a client emulator too. Finally, in Chapter 5 we use a synthetic micro-benchmark to emulate the computational cost of processing dynamic content workloads.

The distributions produced by Surge (Scalable URL Reference Generator) are based on the observation of some real web server logs, from where it was extracted a data distribution model of the observed workload. The workload generated by Surge client emulator follows 6 different realistic models to repeat factors such as page popularity and file size distribution, found in real web server logs.

RUBiS (Rice University Bidding System) implements the core functionality of an auction site: selling, browsing and bidding. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB. RUBiS defines 27 interactions. Among the most important ones are browsing items by category or region, bidding, buying or selling items and leaving comments on other users. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. The RUBiS client emulator uses a Markov model to determine which subsequent link from the response to follow. RUBiS client emulator defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions. Each emulated client waits for an amount of time, known as the think time, before initiating the next interaction. This emulates the period of time a real client takes from one request to the

next one. The think time is generated from a negative exponential distribution.

2.2.2 Workload generator: *httperf*

Although both Surge and RUBiS include workload generation tools, in the experiments presented in Chapter 4 were carried on using a more tunable workload generator: httperf [74]. The tool was set up to follow the emulated client behavior produced by the workload generators included with Surge and RUBiS. This way, we could take full advantage of the workload characteristics present in both workload emulators at the same time that exploited the powerful characteristics of httperf.

Httperf supports both HTTP [41] and HTTPS [91] protocols, allows the creation of a continuous flow of HTTP/S requests issued from one or more client machines and processed by one server machine, whose behavior is characterized with a complete set of performance measurements returned by Httperf. The configuration parameters of the tool used for the experiments presented in this thesis were set to create a realistic workload, with non-uniform reply sizes, sustaining a continuous load on the server. One of the parameters of the tool represents the number of new clients per second initiating an interaction with the server. Each emulated client opens a session with the server. The session remains alive for a period of time, known as session time, at the end of which the connection is closed. Each session is a persistent HTTP/S connection with the server. Using this connection, the client repeatedly makes a request (the client can also pipeline some requests), parses the server response to the request, and follows a link embedded in the response. Httperf allows also configuring a client timeout. If this timeout is elapsed and no reply has been received from the server, the current persistent connection with the server is discarded, and a new emulated client is initiated.

2.3 Utility-driven resource management in virtualized environemnts

Most of the existing clustered execution platforms for application servers must perform some kind of resource management in order to improve the overall performance of the system through the better allocation of the available resources. The resource management activity can be performed statically as well as dynamically, in the case that the allocation policy reacts to the changing workload conditions. The execution platforms can be divided into dedicated hosting platforms and shared hosting platforms depending on whenever different applications can be sharing or not the available resources. Existing

studies report considerable benefits when using dynamic resource allocation policies.

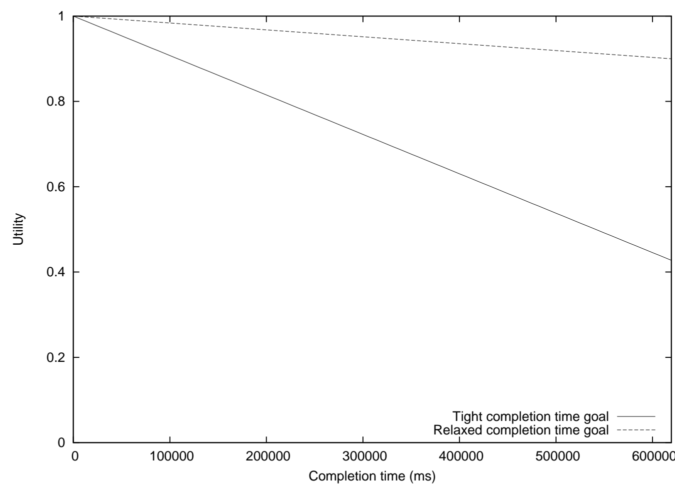
Any resource management policy has a goal, that can be described as an objective function to maximize. Traditionally, the resource allocation decisions were made to better exploit the available resources. A common example of this kind of goals is the maximization of the CPU usage. It is currently usual that the performance of an execution platform is no more measured following low-level metrics but using hihg-level measurements. This way, for instance, the performance of an e-commerce web application can be measured in terms of throughput (requests processed per second, what is a low-level metric) but also in terms of service level offered to the clients. Shifting from one kind of measurement to the other also implies that the objective for any resource management policy is shifted to focus on the service level offered to any particular client.

2.3.1 Utility functions

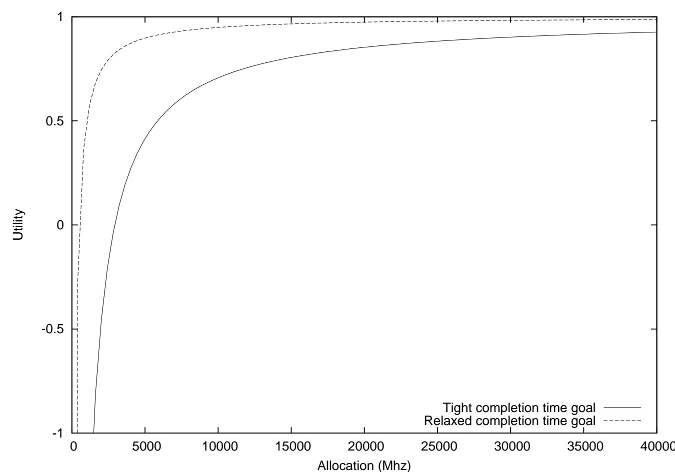
Utility functions translate the satisfaction achieved by an application into a numerical value, usually ranging from $-\infty$ to 1. For example, an application completely satisfied is represented with an utility value of 1, whereas an application completely unsatisfied is represented by a utility value of $-\infty$. The satisfaction of an application is usually measured in relation to a performance objective, such as response time, resource allocation or completion time, amongst others. For example, a web application that aims to offer an average response time no higher than 200ms, will be very unsatisfied when it is allocated resources that permit a response time no better than 600ms, and will be completely satisfied if it is allowed to achieve an actual response time of 50ms.

Figure 2.1 shows an example of a utility function for a job that is running in a system, and for which two different completion time goals have been defined – one is tight and hard to achieve, whereas another one is more relaxed and easily achievable. The figure shows an example of the possible evolution of the utility achieved by the job as a function of the actual completion time (Figure 2.1(a)) and as a function of the speed at which the job is running (Figure 2.1(b)). In this work, we assume that a utility value of 0 corresponds to an application for which the actual performance exactly matches its performance goal.

The use of utility-driven strategies to manage workloads was first introduced in the scope of real-time work schedulers to represent the fact that the value produced by such a system when a unit of work is completed can be represented in more detail than a simple binary value indicating whether the work met or missed its goal. In our work we use monotonic and continuous utility functions to represent the satisfaction of both transactional and long-running workloads, but other approaches could be adopted.



(a) As a function of completion time



(b) As a function of allocation

Figure 2.1 Evolution of utility for long running jobs

2.3.2 Virtualization technology

Virtualization is a technique for hiding physical computing resources from the entities that use them, what usually is achieved through encapsulation. Virtual containers create virtual execution environments that can be used to run embedded applications that use virtual resources to make progress. Virtual containers expose a set of control knobs that can be used to perform management operations on them. A virtual machine is a tightly isolated software container that can run its own operating systems and applications as if it were a physical computer. A hypervisor or virtual machine monitor (VMM) is a virtualization platform that allows multiple operating systems to run on a host computer at the same time. It can run on top of the physical hardware or hosted within an operating system environment, and is in charge of enforcing physical resource allocation decisions

across virtual machines.

Virtualization is not a new concept, but only in the recent years the computing capacity of systems has become high enough to consider it seriously in production environments. Nowadays, dealing with the complexity of managing the deployment and execution of applications and servers in large computing facilities is a big concern, and virtualization has become a key player for these scenarios. Virtual machines can be created and altered on real-time, whereas virtual appliances can be used to deploy complex applications in virtualized environments, independently of the physical platform in which they run.

Virtual containers run guest operating systems that include privileged instructions. Virtualization technology must deal with this issue to be able to run such a virtual machine while keeping control on how each virtual container accesses resources. Virtualization technologies act differently depending on the presence or not of hardware support to run virtual machines.

If no hardware support is present, several techniques can be used to avoid privileged instructions from being executed. Paravirtualization consists in offering software interfaces through the VMM to virtual machines similarly to the interfaces offered by the underlying hardware. This technique requires the explicit modification of the guest operating system code to replace privileged instructions with calls to the VMM interface. Alternatively, the privileged instructions can be replaced on-the-fly with binary translation. If the hardware offers support for virtualization, such as Intel-VT [55] and AMD-V [1], privileged instructions executed by guest operating systems receive a special treatment by the hardware in collaboration with the VMM.

Most remarkable examples of virtualization technologies are Xen [10] and VMware [121], which have strongly contributed to the popularization of the virtualization technology in both desktop and server environments.

The most common operations that can be applied over virtual machines can be briefly summarized, following the terminology described in [10], as:

- PAUSE When a virtual machine is paused, it does not receive any processor time, but remains in memory.

- RESUME Resumption is the opposite of pausing—the virtual machine is once again allocated processor time.

- SUSPEND When a virtual machine is suspended, its memory image is saved to disk, and it is unloaded.

- RESTORE Restoration is the opposition of suspension—an image of the virtual

machine's memory is loaded from disk, and the virtual machine is permitted to run again.

- MIGRATE The virtual machine is first paused, then the memory image is transferred across the network to a target node, and the virtual machine is resumed.
- LIVE_MIGRATE A variant of migration in which the virtual machine is not paused. Instead, the memory image is transferred over the network whilst running.
- MOVE_AND_RESTORE When a virtual machine has been suspended, and needs to be restored on a different node, the saved memory image is moved to the target node, and the virtu machine is then restored.
- RESOURCE_CONTROL Resource control modifies the amounts of various resources that a virtual machine can consume. We consider CPU and memory.

While virtualization can be provided using various technologies, in the experiments presented in Chapter 5 we use Xen as it is capable of providing the wide variety of controls discussed above—all of these controls are most directly accessible from a special domain on each node, labeled domain 0.

Chapter 3

Automatic performance monitoring framework

3.1 Introduction

The extremely complex execution environment of application servers provides a rich framework to develop and run web applications, but makes enormously difficult the task of studying and improving their performance. The parameters that affect the output level of an application server can change depending on many factors – i.e. the workload characteristics can move a performance bottleneck from the network bandwidth to the CPU capacity. This dynamism dramatically increases the complexity of tuning these environments to obtain a maximum output level. It becomes specially true when the desired throughput of an application server is not defined in terms of low level performance goals but in terms of high level performance objectives defined in Service Level Agreements [60].

The first contribution of this thesis is the creation of a deep monitoring framework that produces extremely detailed insight on the system behavior – ranging from the operating system level to the user application code. The monitoring framework is composed of three differentiated tools that work coordinately to create a powerful performance analysis environment. The framework components are 1) JIS [23], the instrumentation tool; 2) Paraver [38], a powerful analysis and visualization tool; and 3) JACIT a code interposition tool that doesn't require source code availability. The framework is able to monitor in real-time some high-level performance metrics of an application server, compare them to some user-defined rules and trigger the appropriate actions if necessary. This way, JIS is be dynamically started and stopped following high-level performance metrics without need of human interaction, with the corresponding reduction of performance overheads, that remain limited to the poor-performance periods that must be carefully analyzed by system administrators.

Java Instrumentation Suite [23] (JIS) is a deep monitoring tool that produces extremely detailed insight on the system behavior – ranging from the operating system level to the user application code. The data collected by the monitoring tool can be later studied using Paraver [38], a powerful analysis and visualization tool. The JACIT tool (Java Automatic Code Interposition Tool) can be used to modify existing bytecodes of a Java application without need of source code availability.

Figure 3.1 shows how JIS compares to other existing tools in terms of coverage of the execution stack layers. Some tools oriented to the study of application servers, such as Hprof [78], OptimizeIt [17] or JProbe [86], report different metrics that measure the application server performance, collecting information through the JVM Profiler Interface (JVMPI [112]). JVMPI exposes JVM's state information through a Java Native Interface [100] (JNI). As it is mostly focused on JVM data, the information that the

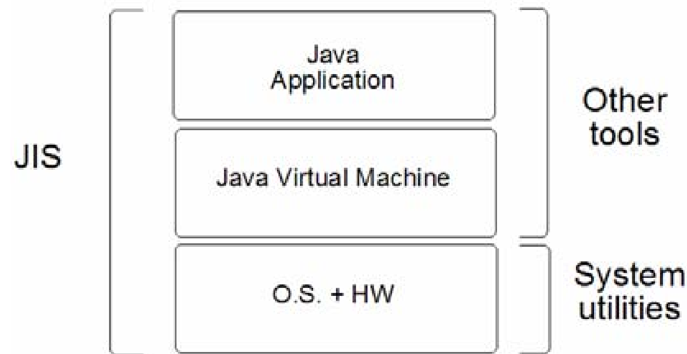


Figure 3.1 JIS compared to other monitoring and tracing tools

JVMPI can offer is limited and therefore, the tools that mainly rely on it to collect performance data show serious limitations in their ability to perform a fine-grain analysis of the performance delivered by Java multithreaded applications.

The basic actions performed by a web application can be summarized as 1) reading/writing contents from/to disks; 2) receiving/sending data from/to networks; and 3) processing data coming from disks and networks. In summary, web applications mostly consume processor, disk and network capacity. Thus, a general requirement in order to successfuly analyze the performance of web applications is to be able to obtain detailed information about the usage of these resources. For long time, users relied on system utilities (such as *sar*, *mpstat*, or *iostat* on Linux, as proposed in [26]) to collect system usage data that was later correlated with the information provided by the above-mentioned tools, thus restricting the analysis possibilities. As opposed to this approach, the combined use of JIS and Paraver offers an integrated performance analysis framework that provides an in-depth insight in the performance of web applications.

JIS has been applied to several environments as a proof of concept. Examples can be found in [22], [44], [88], [46] and [77]. In particular, the work described in [44] and summarized in section 4.3 illustrates how the combined use of JIS and Paraver allowed an in-depth characterization of a secure web middleware. This study uncovered a connection management problem present in the architectural design of most application servers, which was successfully addressed as discussed in in Chapter 4 and is the second contribution of this thesis.

3.2 Components

The three framework components 1) JIS, the instrumentation tool; 2) Paraver, the analysis and visualization tool; and 3) JACIT a code interposition tool, are next described in following subsections.

3.2.1 Java Instrumentation Suite (JIS)

Instrumentation is the first step in the study of an existing web application. JIS (Java Instrumentation Suite) is the instrumentation tool developed to study Java-based applications, covering different available platforms. The result of instrumenting an application with JIS is a post-mortem execution trace, that can be later analyzed with Paraver. Execution traces contain the activity of each system thread in the JVM process as well as the occurrence of some predefined events that take place over the instrumentation period. The execution traces produced by JIS are formatted following the Paraver tracefile specification.

JIS differs from other analysis environments in the offered degree of detail. Most of existing tools focus on offering detailed information about the behavior of studied applications forgetting the interaction of these applications with underlying systems. JIS comes to cover this lack of detail on system status when instrumenting Java applications. Covered resources are diverse, going from thread status (in relation to CPU state) up to the length and duration of I/O operations.

JIS was initially created to support three different execution platforms: Linux-IA32, AIX-PowerPC and IRIX-MIPS. Particularities of each platform have been overcome on JIS by dividing its architecture in three layers; two of them are system-independent and the other one depends on specific system characteristics, as shown on Figure 3.2. In this design, independent layers can be reused on different versions of the tool. This chapter focuses on the Linux-IA32 implementation, running JDK1.3.

3.2.2 Paraver

Paraver[38] is a flexible performance visualization and analysis tool based on an easy-to-use Motif GUI. Paraver was developed to respond to the need to have a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information useful to improve the decisions on whether and where to invert the programming effort to optimize an application.

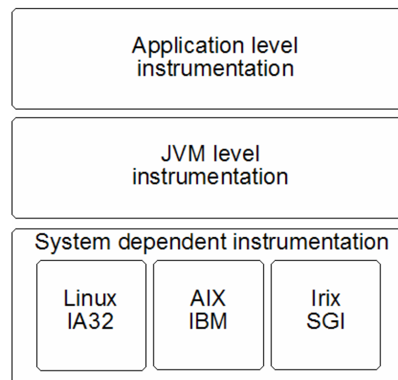


Figure 3.2 JIS architecture

Paraver tracefiles contain a set of records that belong to one of three different options: state, events and communication records. State records indicate the state of system threads over time (running, ready or blocked on JIS); event records represent punctual occurrences of any events that may help understanding the execution tracefile, such as system calls, and can be originated in any level of the execution stack; and communication records represent data flows between system threads, and are used on JIS to track resource sharing among threads as, for example, sockets being (re)used by different threads.

Trace file analysis with Paraver can be done as a manual process, viewing (with the Paraver visualization module) the graphical trace representation and looking for some performance problems, or using the statistical tools provided with Paraver to make an automatized analysis of the trace. In any case, all the possible views and statistical calculations made on a trace file can be saved as Paraver configuration files. It allows users to create a large amount of preset views of the trace file that can point out some performance indexes or conflictive situations in an automatic way.

The graphical views of the trace files are based on the representation of threads, characterized by their state over time and by some punctual events. The combination of states and events makes possible to do a detailed and intuitive representation of an application behavior. The analysis views apply statistical calculations to the trace file information and summarizes the results as a table. These calculations can be done as a function of thread state values, punctual events and tread state values of one window in relation to thread state values (known as categories) of another window.

3.2.3 Java Automatic Code Interposition Tool (JACIT)

The JACIT tool (Java Automatic Code Interposition Tool) can be used to apply the aspect[111] programming paradigm to the modification of existing bytecodes of an

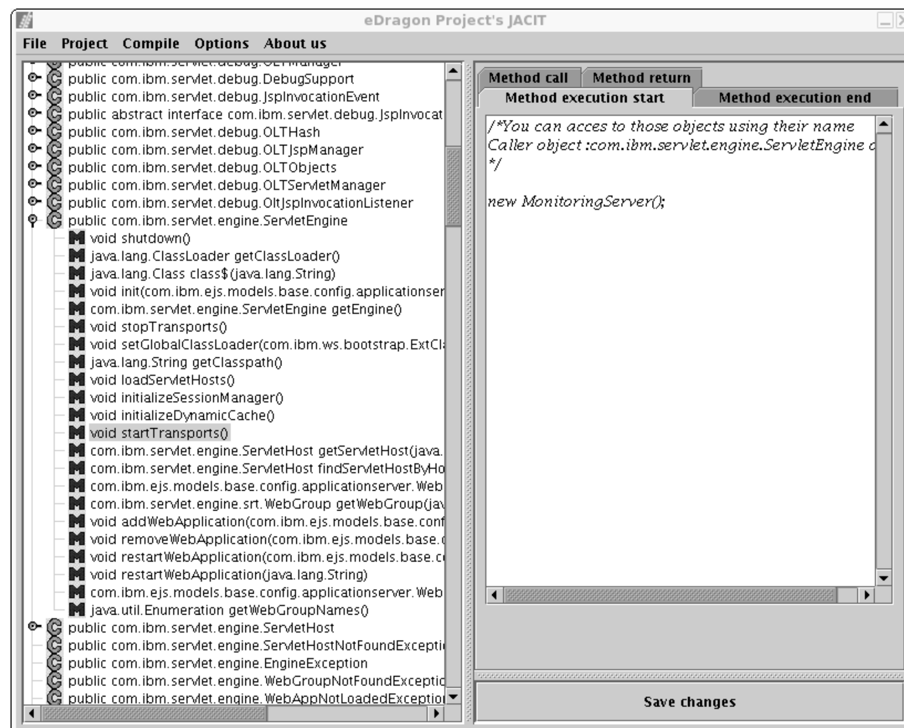


Figure 3.3 JACIT screenshot

application without need of source code availability.

With the JACIT tool it is possible to open a jar file from any application, choose one of the classes contained in the jar file, select one of the methods or interfaces of the method and decide to add some code before or after invoking it. The inserted code can use anyone of the parameters of the Java method. Later, the code can be compiled to test its correctness and after that, an equivalent aspect programming file is generated (if wanted) and the needed changes are applied to the jar file to execute the added code when required. Finally the jar file is saved and a backup jar file is also produced. Figure 3.3 shows an screenshot of the moment in which some code is interposed in the Servlet engine initialization code of IBM's Websphere Application Server [53] without need of source code access.

3.3 Implementation of JIS Linux-IA32

Four levels are considered by JIS when tracing a system: 1) operating system, 2) JVM, 3) middleware (application server) and 4) user application. Information collected by all levels is finally correlated and merged to produce an execution trace file following the general schema shown in Figure 3.4. The level of detail of the information produced

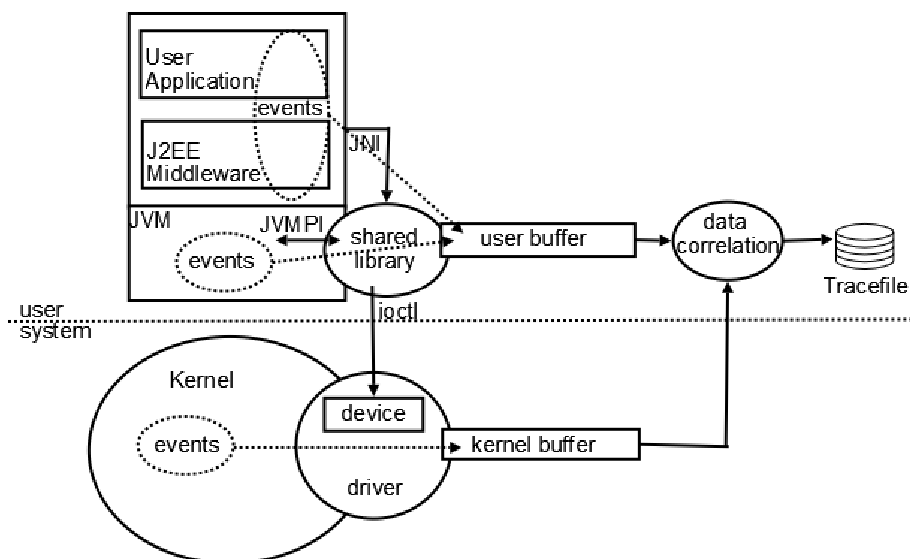


Figure 3.4 JIS instrumentation process

by each JIS level can be dynamically configured. The responsibilities for each JIS level as well as their development technology are discussed in following subsections. The implementation details correspond to a Linux-IA32 kernel version 2.5.63.

3.3.1 Operating system level

Information collected from the operating system level covers threads' state and system calls. Thread information is obtained directly from the Linux scheduler routine and information from syscalls (I/O, sockets, memory management, thread management) is obtained by intercepting some entries of the syscall table. When working with Java-based applications, collected information is limited to the JVM process, and other processes on the system are ignored.

To perform useful application instrumentation, continuous system state information must be offered to developers. On the Linux version of JIS, considering the open platform characteristics of Linux systems, we decided to extract system information directly from inside kernel. This task was divided in two layers: one based in a kernel source code patch and the other in a system device and its corresponding driver (implemented in a Linux Kernel Module, LKM). Both of them are described next.

3.3.1.1 Kernel module

The kernel module implements four basic functionalities of JIS:

1. Interception of system calls: This is done by modifying the global system call table

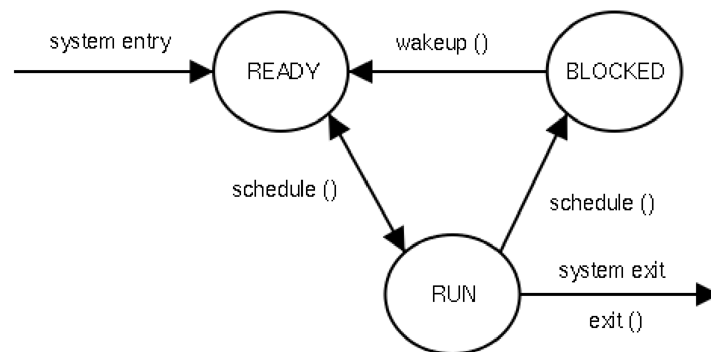


Figure 3.5 Thread states considered by JIS and intercepted functions to detect transitions

in order to use a modified function instead of the original system call. After the call is intercepted, the original system call function is invoked in order to preserve the original system behavior. Listing 3.1 shows an example of the interception code that corresponds to the read system call.

2. Implementation of a virtual control device: The instrumentation infrastructure can be controlled from user space. Basic implemented functions are: *open*, *close* and *ioctl*. Open and close calls are used to be able to work with the device. *Ioctl* call is used to send control commands to the kernel module, such as monitoring start and stop actions.

3. Creation of a kernel buffer: It is used to store the kernel-generated events such as context-switches, system calls, specific driver operations and memory management, that will be later correlated with the user space performance data. The buffer is implemented as a circular buffer what makes possible that a half of the buffer can be dumped to disk while the other half is being used to register new kernel events.

4. Creation of the dumping mechanism: It allows the system side of JIS to dump the kernel buffer data to a filesystem more quickly than the buffer is filled again. This task is specially challenging and requires the use of a kernel thread that dumps data into the file asynchronously while the other system threads keep running and producing kernel events. This mechanism makes extensive use of the *workqueue* interface present in the Linux kernel. Listing 3.2 shows how a the kernel buffer is dumped to the filesystem. Notice the use of a kernel thread, with no associated user context data, and the kernel *workqueue* interface to manage such a critical task. The need of the FS segment switch is due to the fact that file operations are not

```
void enable_rw_sys_call_hooks(void) {
        orig_sys_read   = sys_call_table[__NR_read];
        orig_sys_write  = sys_call_table[__NR_write];

        sys_call_table[__NR_read]       = jis_sys_read;
        sys_call_table[__NR_write]      = jis_sys_write;
}


asmlinkage ssize_t jis_sys_read(unsigned int fd,
                                char *buff,
                                size_t count) {

        if(!trace_enabled)
                return orig_sys_read(fd, buff, count);

        register_syscall_in(current->pid,
                        smp_processor_id(),
                        JIS_IO_READ);

        long res = orig_sys_read(fd, buff, count);

        register_syscall_result(current->pid,
                                smp_processor_id(),
                                res, fd);
        return res;
}
```

Listing 3.1 Read system call interception and syscall table modification

intended to be invoked from inside the kernel, but as a result of a trap operation. Similar behavior can be observed in other proposals appeared after JIS, such as the RelayFS [126] used in recent versions of the Linux Trace Toolkit [125] (LTT).

3.3.1.2 Kernel patch

Some system events cannot be extracted by any other way than inserting hooks inside the kernel source. These special events are related to kernel threads state and other ways of obtaining this information are not flexible enough. For instance, Linux exposes process [1] status through the *proc* file system, but they are reported as being only Runnable or Blocked. Runnable implies that a process is ready to run on a processor, but doesn't give information about if it is actually running or if it is still waiting for a processor to start execution. This issue makes the *proc* file system insufficient to determine process

[1] On Linux systems using the linuxthreads implementation of POSIX threads, talking about processes is equivalent to talking about threads because this concrete implementation uses the Linux clone system call to create new threads, which means that threads are, in fact, cloned processes sharing required resources.

```
struct work_struct dump_job;


void start_kernel_thread_dump (struct dump_descriptor *dd) {
        PREPARE_WORK(&dump_job, kernel_thread_dump, (void *)dd);
        schedule_delayed_work(&dump_job,0);
}


// dump_lock must be acquired before starting the kernel thread
static void kernel_thread_dump(void *__dump_info)
{
        struct dump_descriptor *dump_info;
        dump_info = (struct dump_descriptor *)__dump_info;

        oldfs = get_fs(); set_fs(KERNEL_DS);
        int ret = f->f_op->write(f, (char *)dump_info->src,
                                  dump_info->size, &f->f_pos);
        set_fs(oldfs);
        buf.free += (dump_info->size/RECORD_SIZE);

        spin_unlock(&dump_lock);
}
```

Listing 3.2 Using a kernel thread to dump the kernel buffer into a file

status continuously.

The kernel patch captures 4 different kernel events that are inserted in kernel buffer. Captured events, and their corresponding modified kernel functions, are 1) thread creation in the *schedule* function; 2) thread destruction in the *exit* function; 3) thread preemption in the *schedule* function; and 4) thread resumption in the *wakeup* function. Listing 3.3 shows the code invoked when a context switch is completed. Notice how the context switch is registered by calling the JIS routine. Figure 3.5 shows the thread state graph considered by JIS. Other states are not considered relevant to study the behavior of applications in this environment, since the thread state information is complemented in JIS with events that provide the required information to fully understand the activity of a thread.

3.3.2 Java Virtual Machine level

Java internal semantics are only visible from within the JVM, but describe elements, such as Java threads and internal monitors, that can seriously impact the performance of an application. Because of this, comprehensive instrumentation of Java applications must be composed, in part, by internal JVM information. Current versions of JVM implement a Profiler Interface called JVMPI [112], recently substituted by the JVMTI [103]. This facility is used by JIS to include information about Java application semantics on its

```
static inline void finish_task_switch(task_t *prev)
{
        runqueue_t *rq = this_rq();
        struct mm_struct *mm = rq->prev_mm;

        rq->prev_mm = NULL;
        finish_arch_switch(rq, prev);
        if (mm)
                mmdrop(mm);
// JIS
        register_switch_on_jis(prev, current, smp_processor_id(),
            task_timeslice(prev));
// ENDJIS

        if (prev->state & (TASK_DEAD | TASK_ZOMBIE))
                put_task_struct(prev);
}
```

Listing 3.3 End of context switch event

instrumentation process. This means that a developer analyzing own applications will be able to see system state information during execution expressed in relation to some Java semantics defined at application development time.

The JVMPI is based on the idea of creating a shared library which is loaded on memory together with the JVM and which is notified about selected internal JVM events. Choosing hooked events is done at JVM load time using a standard implemented method on the library that is invoked by the JVM. Events are notified through a call to a library function that can determine, by parsing received parameters, what JVM event is taking place. Listing 3.4 shows an extract of the code the enables the JVMPI and sets the event notification routine. The treatment applied to each notified event is decided by the profiler library, but should not introduce too much overhead in order to avoid slowing down instrumented applications in excess. Some of available events are: start and end of garbage collecting, class load and unload, method entry and exit and thread start and end.

On JIS, only thread start and thread end events are considered by default. Importance of these events comes from their associated information: they contain information about the internal JVM thread name (that one defined by the developer) and allow JIS to match Java threads with kernel threads. Both of them are very useful for developers to understand system information when visualized. Notice that JVM implementations for Linux-IA32 use a 1:1 threading model, in which one Java thread is supported by one system thread.

Optionally, other JVM events can be chosen to be incorporated on instrumented information depending on developers' requirements. Activation of many event notifications can result in severe overheads like in the case of the method entry and method exit events,

```
#include <jvmpi.h>

static JVMPI_Interface *jvmpi_interface;


JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options, void
*reserved) {

    jvmpi_interface->NotifyEvent = notifyEvent;

    jvmpi_inter->EnableEvent(JVMPI_EVENT_MONITOR_WAIT, NULL);

    return JNI_OK;
}



void notifyEvent(JVMPI_Event *event) {
  switch(event->event_type) {
    ...
    case JVMPI_EVENT_MONITOR_WAIT:
    ...
  }
}
```

Listing 3.4 Enabling the JVMPI interface

because of their high notification frequency.

3.3.3 Middleware and User application levels

Both the middleware level and the user application level rely on the use of a system shared library, provided with JIS, that implements a Java Native Interface [100] (JNI) interface accessible from the Java code to inject user events in the final tracefile. Notice that the importance of these events is that they are generated on execution time and are automatically correlated with all the other performance data. The events can be injected by both modifying the middleware source code (if available) or using the JACIT tool, briefly discussed in section 3.2.3.

Information relative to services (i.e. servlets and EJBs) or transactions can be obtained from the middleware level. With this information, system performance can be put in relation to the middleware performance at any moment in time (i.e. servlet execution time in relation to the percentage of running time of the kernel thread which provided the servlet execution). An example of such a code injection is shown in Listing 3.5, where the beginning and completion of the processing of a HTTP request are indicated with two events in the Tomcat [6] Servlet container. This information can be used later with

```
package javax.servlet.http;

public abstract class HttpServlet extends GenericServlet
                                  implements java.io.Serializable
{
  ...
  protected void service(HttpServletRequest req,
                         HttpServletResponse resp)
                throws ServletException, IOException
  {
    edragon.WebApps.UserEvent(SERVICE,BEGIN);
    ...
    doGet(req, resp);
    ...
    edragon.WebApps.UserEvent(SERVICE,END);
  }
  ...
}
```

Listing 3.5 Injecting user-level events

Paraver, to calculate statistics in the scope of one single servlet invocation.

Additionally, the user application code can be modified to inject events into the tracefile too. This process is extremely helpful for the correct understanding of the real cost and effects of any portion of the user application code.

3.3.4 Merging data

System space and user space captured events must be put together to generate the final trace. The merging process is done either when the JVM is shut down or when it is requested by the user. Both the user-space and the system-space buffers are regularly dumped into files. Generated records contain an associated timestamp measured in cycles (using the *rdtscll* instruction present in IA32 systems). As the timestamps in the two buffers are consistent, data can be safely merged and sorted. Finally, the tracefile is converted to the Paraver format to be later processed.

3.3.5 Overheads in the Linux-IA32 implementation

The instrumentation process of JIS introduces some overheads during the execution of the application. Nevertheless, this overhead is low enough not to affect the conclusions extracted from applications analysis. In order to measure the overhead introduced by the tool two applications were run without instrumentation and with different levels of instrumentation, and execution times have been studied to determine the impact of

	Execution Time (ms)		
matrix size	No instrumentation	System instrumentation	System + JVMPI instrumentation
1000x1000	20662 ± 69,07	20710 ± 100,39 (+0,23%)	20713 ± 23,45 (+0,24%)
750x750	9478 ± 32,37	9492 ± 20,26 (+0,15%)	9567 ± 20,81 (+0,93%)
500x500	3434 ± 14,82	3450 ± 9,79 (+0,47%)	3506 ± 8,16 (+2,08%)
250x250	699 ± 9,02	722 ± 13,43 (+3,29%)	750 ± 1,944 (+8,89%)

Figure 3.6 CPU intensive application overhead results

	Execution Time (ms)		
Moved size	No instrumentation	System instrumentation	System + JVMPI instrumentation
100 Mb	1918 ± 10,76	1995 ± 40,52 (+4,01%)	2091 ± 48,27 (+9,02%)
200 Mb	3734 ± 40,26	3899 ± 315,67 (+4,43%)	3957 ± 47,02 (+5,99%)
400 Mb	14281 ± 201,25	14337 ± 165,09 (+0,39%)	14896 ± 41,67 (+4,31%)
800 Mb	32093 ± 266,74	34931 ± 1547,26 (+8,84%)	35639 ± 526,87 (+11,05%)

Figure 3.7 I/O intensive application overhead results

instrumentation on the performance of applications.

Used applications are distinguished by their focus of study: one is CPU intensive and the other I/O intensive. This first one is a LU decomposition code and the second one is the core of the Tomcat [6] server used in this test to transmit data (html files) in 2Kb chunks. Tests have been repeated with different configurations of the applications, and obtained results are presented in Figure 3.6 and Figure 3.7. Execution times with no instrumentation, with system instrumentation only and with coordinated system and JVMPI instrumentation are presented. Times are mean values with corresponding standard deviations. Overheads are indicated between parentheses. As it can be seen, low-order overheads are introduced to execution times when instrumenting applications. Activating JVM information through the JVMPI results in an increase of overheads respect to produced ones with only system level instrumentation. Observed overheads can be considered acceptable in order to not to perturb the behavior of applications when instrumenting them.

3.4 Automatic monitoring

In this section we present an extension to the monitoring infrastructure that allows it to operate automatically, without need of human interaction in the process of performance data acquisition, whereas it is driven by some user-defined rules. Such a monitoring framework operates by continuously observing some high-level performance metrics delivered by the application server, and triggering the in-depth tracing process of the whole application server's execution stack when it is observed that some minimum user-defined performance objectives are not met. System administrators and software developers can take especial advantage of an automatic monitoring tool such as the one proposed in this work, especially if they run their complex systems under high-availability requirements that force them to keep their environments up and running 24x7. This automatization of the monitoring tool has been developed to work with WebSphere Application Server[53] (WAS), and the resulting autonomic monitoring environment has been named WAS Control Center

The WAS Control Center environment is fully developed for the Java platform so it has no system dependencies, although it relies on the architecture of WebSphere Application Server[53] (WAS), so its use is limited to this application server. The particular platform used for the development was WebSphere Application Server v4.0 Advanced Edition running on a 1.3 IBM Java Virtual Machine.

3.4.1 Monitoring high-level performance metrics

Some different approaches can be taken to continuously measure the performance of an application server. In this work we measure the performance of the WebSphere Application Server using the Performance Monitoring Infrastructure[89] (PMI).

The Performance Monitoring Infrastructure consists in a set of libraries and packages developed to simplify the task of collecting, processing and visualizing performance information regarding the application server. PMI gets information from all WAS components and makes it available to users. It offers a rich set of performance indexes of the application server. Some examples of indexes offered by PMI are the total number of requests to an object, response time of a web accessible object and number of concurrent active requests. These indexes can be obtained for both individual objects (servlets and EJBs) and the global system.

The performance data collected by the PMI is made accessible in different ways. In particular, WebSphere has a servlet deployed on it that exposes the information acquired by the PMI. WAS Control Center environment obtains continuous performance

information from WebSphere by periodically polling the PMI Servlet. This servlet, when accessed, queries the WAS Performance Monitoring Infrastructure to obtain performance indexes of the application server and returns them to the client summarized in a XML file that describes the current performance indexes for the different components of the application server. Each time the PMI servlet is accessed, an updated version of the XML-formatted performance report is returned.

3.4.2 Automatic management of the monitoring infrastructure

The major feature of WAS Control Center is that it is able to automatically detect poor performance periods on the application server according to a set of performance objectives defined by the user. Each objective is related to a PMI performance index, and can be defined in terms of maximum value, minimum value, deviation over time and basic logic combinations of these values. An example of such a performance objective is the response time observed for a particular web component. When one of the performance objectives is not met, JIS is automatically started to capture the execution trace that may contain the root cause of the observed performance, and it keeps running until the performance objective is met again.

WAS Control Center is composed of two main components: the Control Center and the Monitoring Server. The Control Center collects performance data through the PMI servet and detects anomalous performance periods. The Monitoring Server is a remote agent for the Control Center, and runs within WebSphere. It can be used to take any action required by the Control Center in response to a performance-triggered request, and in particular, to control JIS locally. A general operation diagram of WAS Control Center is shown in figure 3.8.

The Monitoring Server is integrated with Webshpere by using JACIT to modify the *startTransports()* method of the *ServletEngine* class, a process that is shown in Figure 3.3. Once this method is invoked, the Monitoring Server is loaded. Once the application server instance is completely up and running, the PMI engine is started and the PMI servlet is enabled. At this point, the Monitoring Server is ready to receive requests from the Control Center. Requests can be manually submitted using the GUI or can be an automatically triggered.

The Control Center Application works remotely from the application server machine and also offers a graphical interface to remotely manage the configuration parameters of JIS, as for example, the level of detail captured by the tool. It is composed of four cooperative components: a GUI, an XML parser module, an application logic module and a communication stub.

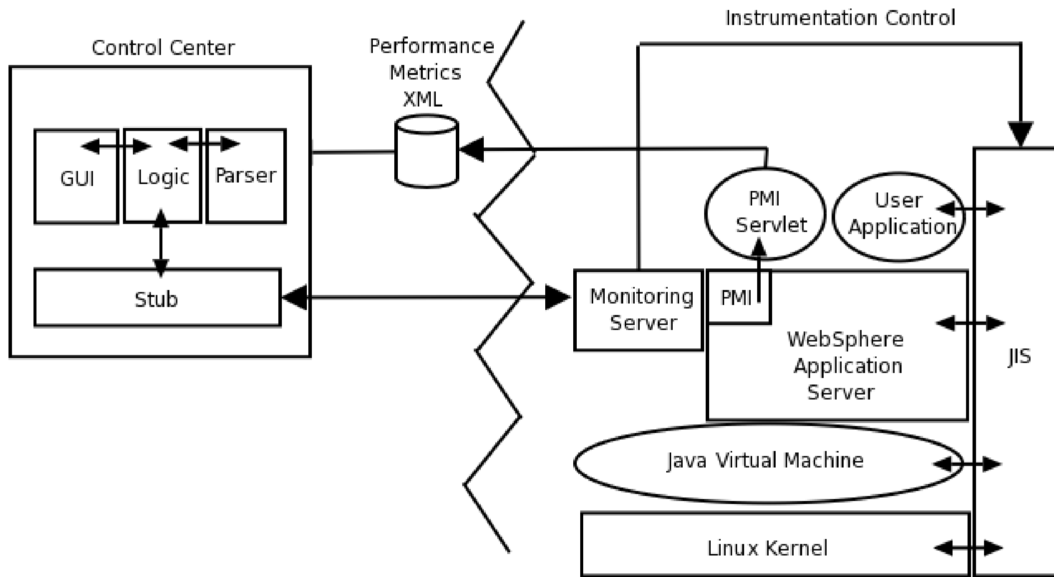


Figure 3.8 WAS Control center operation diagram for automatic system tracing

The XML parser processes the information provided by the PMI Servlet. The amount of information obtained from the PMI servlet depends on the configuration of the WAS Control Center. So, the XML parser module not just parses the XML file but also filters the performance data according to user preferences. As long as the structure of the XML file is dynamic (i.e. new web components each time the PMI servlet is accessed), the XML parser detects the new structure and modifies the GUI data accordingly. The logic module of the Control Center contains the application logic required to make the environment work. The periodicity in which the PMI servlet is polled is a user-configured parameter. Each time new performance data is obtained, all the performance objectives are checked. The GUI module offers an overall configuration interface for the entire environment. Figure 3.9 shows an example of the Control Center GUI can be observed. According to this example, JIS will be activated when the number of active servlets in the server suffers a variation higher than 5% with respect to the four last observed values. The communication stub isolates the Control Center logic from the PMI interface. New stubs could be developed to integrate it with other Application Servers.

3.4.3 Case study

In this section we present a simple but illustrative example of use of the WAS Control Center. We used a 4-way application server machine running WebSphere 4.0, a 2-way database host machine running MySQL and a client machine to run the servlet-based version of RUBiS[4]. Each one of these machines disposes of 2Gb of physical RAM. The

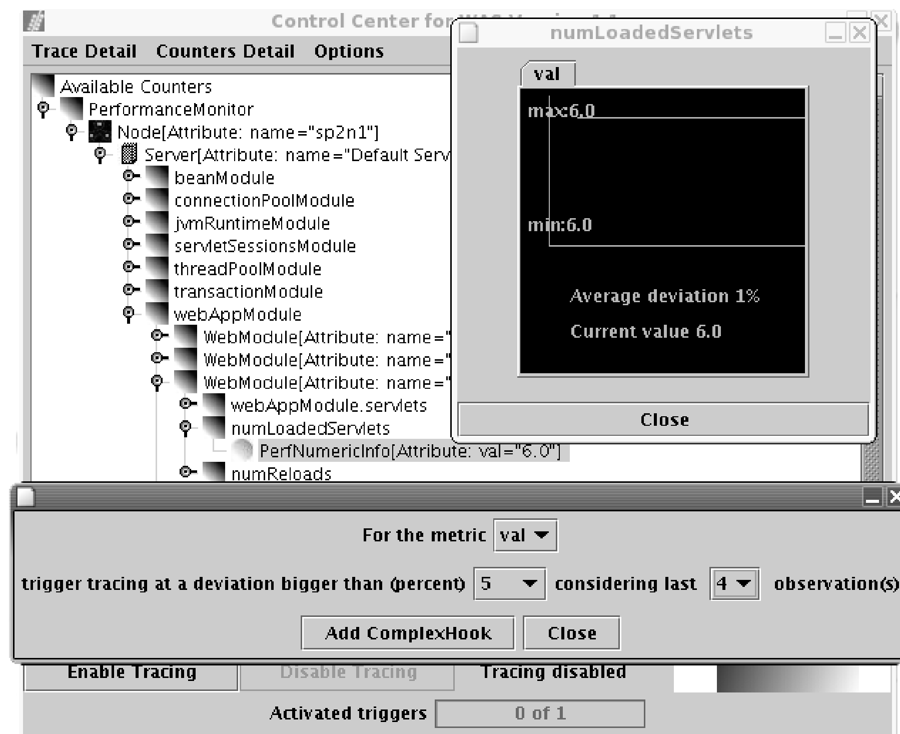


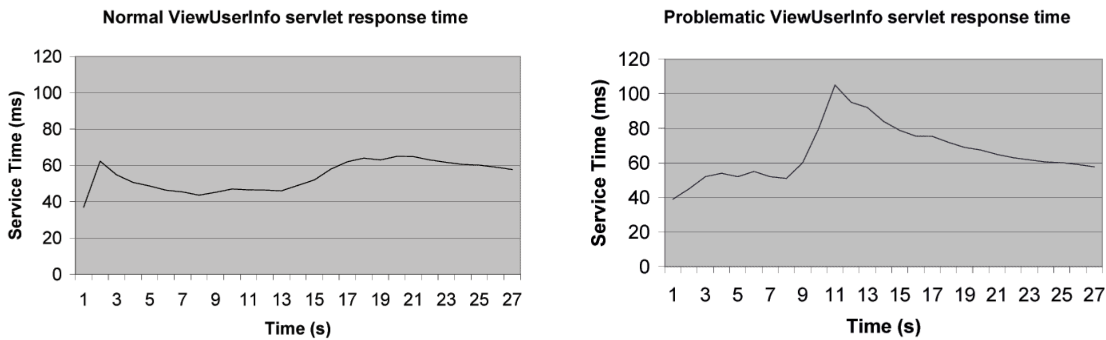
Figure 3.9 View of the WAS Control Center GUI

database server hosted RUBiS data.

After a warm-up period, the system became stable, and the response time for the *ViewUserInfo* servlet was measured, as it can be observed in figure 3.10(a). At this point, we defined a performance objective for this servlet that was 10% higher than the actual response time observed in normal conditions. Although the performance objective was set only for the *ViewUserInfo* servlet, it could be extended to consider the whole application server or to measure any other available performance metric.

Next step in the experiment was the introduction of some CPU-consuming processes in the machine hosting the database server. As it was expected, the service level offered by MySQL server started worsening with this process and in turn the response time offered by WebSphere too, as it can be seen in figure 3.10(b). When the disturbance was removed from the database machine, the response time came down again as expected.

At the time the response time reported by the PMI servlet for the *ViewUserInfo* servlet increased and the performance objective defined for this servlet started being consistently missed, the Control Center engine sent a request to start running JIS on the application server machine. The request was processed by the Monitoring Server. When the response time went back to normal values, JIS was automatically stopped and an output tracefile generated.



(a) Normal response time during a benchmark run (b) Degraded response time

Figure 3.10 Observed response time for a servlet of the RUBiS Benchmark

The study of the execution tracefile automatically generated by JIS clearly indicated that the *Servlet.Engine.Transport* worker threads of WebSphere spent long periods awaiting data to come from the sockets that connect WebSphere and MySQL, grouped in a connection pool. In particular, threads were blocked in socket read system calls, and the time spend there was clearly longer than the time observed in normal conditions.

Although this is a simple testing example for the environment, the experience proved usefulness as an environment for system administrators which are in charge of "24x7" environments to obtain valuable information about the factors that can modify the performance of an application server.

3.5 Related work

Many tools aim to offer deep tracing of multithreaded web applications, but few of them cover all the levels of the execution stack. Our framework represented a first proposal that provided complete correlation of information among all the different levels involved in the execution of an application server middleware running on top of the Java Virtual Machine [99]. Other tools, like Jinsight [84], JaViz [59] and DejaVu [29] generate execution traces of Java applications but none of them with the level of detail delivered by JIS. Jinsight and DejaVu work with single instances of JVMs, while JaViz is focused on client/server Java applications that make extensive use of RMI [105]. All these tools use modified JVMs to collect the execution information.

Other tools such as Hprof [78], OptimizeIt [17] and JProbe [86] make extensive use of the JVM Profiling Interface (JVMPI [112]) to collect performance information from the JVM. All these tools are focused on offering performance insight in terms of hotspots and bottleneck detection, memory consumption and all other kind of aggregated metrics, but

they don't offer a detailed view of the execution behavior of the application. JIS collects data in such a way that can be later visualized with a performance analysis tool such as Paraver [38]. Using Paraver, the information collected in the trace files can be processed to offer aggregated metrics in a similar way that the mentioned tools do, but also can be used to make much more detailed performance analysis in some sections of the monitored period or by-thread analysis. The data collected by hprof can also be post-processed with tools like PerfAnal [104] and HAT [15], but these tools are clearly limited in tems of functionalities when compared to Paraver.

In the last years, a number of new tools and monitoring frameworks have appeared to meet the requirements of new execution environments, with special focus on the J2EE [98] platform. These tools can be classified into the new Application Performance Management (APM) category as they try to help in a wider environment than the tools presented before. These tools are integrated into IDEs and guide developers all over the production of a piece of software to meet their performance goals. Some examples of such an environment are Quest's APM Suite for J2EE [87], Borland's Optimizeit ServerTrace [18], and CA Wily Introscope[120]. The scope and objectives of these tools are beyond those of JIS and Paraver. Still, they don't offer the options in terms of fine-grain analysis that JIS does, but on the other hand they offer many extended features not covered by the combination of JIS and Paraver.

In relation to WAS Control Center, similar proposals have been previously done on the area of distributed systems, like in [16] and in [63], but they are not focused on application servers like ours. The concept of autonomic computing has been widely explored and categorized by IBM, as described in [62]. There is some work published in the field of application servers' performance modeling, like in [49] and in [66], showing different approaches to the complexity of the problem. Also some works face up to the performance prediction for application servers, like in [72]. Some discussion about the creation of self-managed systems for web or application servers can be found, working with agents like in [35] or with other architectures, like in [73], in [109] and in [27].

3.6 Summary

In this Chapter we have summarized the first contribution of this thesis that consists in the creation of an automatic monitoring framework specially focused on web applications. It correlates detailed system information with high-level performance data in order to make possible a complete performance analysis of web applications. It is able to monitor in real-time some high-level performance metrics of an application server, compare them

to some user-defined rules and trigger the appropriate actions if necessary. This way, it can be dynamically started and stopped following high-level performance metrics without need of human interaction.

The monitoring framework is composed of three differentiated tools that work coordinately to create a powerful performance analysis environment. Java Instrumentation Suite [23] (JIS) is a deep monitoring tool that produces extremely detailed insight on the system behavior – ranging from the operating system level to the user application code. The data collected by the monitoring tool can be later studied using Paraver [38], a powerful analysis and visualization tool. The JACIT tool (Java Automatic Code Interposition Tool) can be used to modify existing bytecodes of a Java application without need of source code availability. JIS is the major component in the first contribution of this thesis.

Four levels are considered by JIS when tracing a system: 1) operating system, 2) JVM, 3) middleware (application server) and 4) user application. Information collected by all levels is finally correlated and merged to produce an execution trace file.

Information obtained from the operating system level covers threads' state and system calls. Thread information is obtained directly from the Linux scheduler routine and information from syscalls (I/O, sockets, memory management, thread management) is obtained by intercepting some entries of the syscall table. This task was divided in two layers: one based in a kernel source code patch and the other in a system device and its corresponding driver. When working with Java-based applications, collected information is limited to the JVM process, and other processes on the system are ignored.

Java semantics are just considered inside the JVM. Because of this, comprehensive instrumentation of Java applications must be composed, in part, by internal JVM information. This information is used by JIS to include Java application semantics on its instrumentation process

JIS allows the generation of events from both the middleware and the user application levels, that are later available in the tracefiles. Notice that the importance of these events is that they are generated on execution time and are automatically correlated with all the other performance data. Information relative to services (i.e. servlets and EJBs) or transactions can be obtained from the middleware level. Additionally, the user application code can be modified to inject events into the tracefile too, what improves the understanding of the real cost and effects of any portion of the user application code.

Finally, we have shown a how the framework can be automatized to work with WebSphere, and how it can be used to control JIS without need of human cooperation until the analysis step.

The monitoring framework has been tested in different environments, such as those described in [22], [44], [88], [46] and [77]. In particular, the experiment described in [44] and described later in Section 4.3, shows a successful example of how such a fine-grained monitoring environment can provide a new insight in the performance characterization of web applications. This particular example demonstrates how crucial a connection management policy results in the performance delivered by an application server, what motivated the second contribution of this thesis that is presented in the next Chapter.

The work performed in this area is described in the following publications:

[23] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, and J. Labarta. **Complete instrumentation requirements for performance analysis of web based technologies**. In Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), 2003

[21] D. Carrera, D. García, J. Torres, E. Ayguadé, and J. Labarta. **WAS Control Center: An autonomic performance-triggered tracing environment for Websphere**. In Proceedings of 13th Euromicro Conference on Parallel, Distributed and Networkbased Processing (PDP'05), 2005

as well as resulted in the following derived work:

[44] J. Guitart, V. Beltran, D. Carrera, J. Torres and E. Ayguadé. **Characterizing Secure Dynamic Web Applications Scalability**. 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), April 2005.

[45] J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. **Designing an overload control strategy for secure e-commerce applications**. Computer Networks 51, 15 (Oct. 2007), 4492-4510

[22] D. Carrera, J. Guitart, V. Beltran, J. Torres and E. Ayguadé. **Performance Impact of the Grid Middleware**. In Engineering the Grid: Status and Perspective, American Scientific Publishers, January 2006. ISBN: 1-58883-038-1

[88] R. Nou, F. Julià, D. Carrera, K. Hogan, J. Labarta, J. Torres. **Monitoring and analysis framework for grid middlewares**. In Proceedings of 15th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'07), 2007

[46] J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. **Tuning Dynamic Web Applications using Fine-Grain Analysis**. In 13th Euromicro Conference on

Parallel, Distributed and Network-based Processing (PDP'05), pp. 84-91, February 2005.

Chapter 4

Adaptive architecture for application servers

4.1 Introduction

Application servers are based on the well-known client/server paradigm. Usually, the client and the server communicate using the HTTP protocol over TCP connections. The component of an application server that is in charge of managing the communications with the client as well as dealing with the network protocols is the web container.

In the old times, the performance of web applications was measured following low-level performance goals only, such as replies per second. In contrast, more advanced high-level goals are considered when measuring the performance of modern-day applications. High-level objectives are deeply related to business metrics such as the number of business transactions that are successfuly completed and the service level delivered for those transactions. In terms of web applications, one business transaction is considered to be completed when a user browsing session successfully finishes. Therefore, web container architectures must be designed to deliver high performance in terms of high-level performance metrics in order to be on the side of the interests of modern web applications.

The development of web containers is always a challenging task because it implies the creation of high performance I/O strategies. A server that is subject to a workload of thousands of clients needs to perform really efficiently in order to to deliver an acceptable service level. There are multiple architectural options to design a web container, depending on the connection management model that is used. The two major alternatives are the multithreaded model and the event-driven model. In both models, the work to be performed by the server is divided into work assignments that are assumed each one by a thread (a worker thread). In the multithread model, one worker thread remains associated to a client connection until it is closed. Alternatively, in an event driven model the worker thread is associated to a HTTP request, and each worker thread is multiplexed among the clients that are connected to the server. In the last years the multithreaded approach was widely used on commercial web containers.

The use of connection persistence in the HTTP protocol results in a dramatic performance impact for highly-loaded multithreaded web containers. Persistent connections, which means client-server connections are kept established between consecutive HTTP requests, force worker threads to remain idle while waiting for new incoming requests or, alternatively, to close connections and re-establish them later, with the corresponding overhead. The former can result in underutilized systems, while the latter degrades the service level offered by the server. The event-driven model is better suited for this kind of scenario, multiplexing a short number of threads among a large number of connections, without need to close them. Unfortunatelly, the multithreaded programming model leads

to a very easy and natural way of programming a web container, whereas the event-driven model makes the task of designing an efficient web container a hard challenge.

The second contribution of this thesis is the proposal of a new hybrid[1] web container design that exploits the best of both the multithread and the event-driven models, resulting in an improved high-level performance without increasing its development complexity. Additionally, it shows an extraordinary adaptability to the workload conditions and reduces noticeably the need of human interaction to fine tune the web container. The design of such a new architecture is motivated by a preliminary study of Tomcat's vertical scalability when subject to secure workloads. The results confirmed that the server can be easily overloaded if connections are not properly managed, demonstrating the convenience of developing advanced connection management strategies to overcome such a complicated scenario. The hybrid architecture is specially well suited for commodity solutions, in which no external load balancing nor overload protection mechanisms are available.

[1]The work presented in this chapter was shared with the open source communities that maintain some of the most popular Java web containers. Some of these projects were considering the possibility of writing a web container with a similar architecture at that moment. At the end, many application servers such as Apache Tomcat [6] and Glassfish [97] (on which Sun Java System Application Server [106] is based) now support similar hybrid solutions.

4.2 Application server architectures

4.2.1 Multithreaded architecture with blocking I/O

The multithreaded programming model leads to a very easy and natural way of programming a web container. The association of each thread with a client connection results in a comprehensive thread lifecycle, started with the arrival of a client connection request and finished with the connection close. This model is especially appropriate for short-lived client connections and with low inactivity periods, which is the scenario created by the use of non persistent HTTP/1.0 connections. A pure multithreaded web container architecture is generally composed by an acceptor thread and a pool of worker threads. The acceptor thread is in charge of accepting new incoming connections, after what each established connection is assigned to one thread of the workers pool, which will be responsible of processing all the requests issued by the corresponding web client. A brief operation diagram for this architecture can be seen on figure 4.1.

The introduction of connection persistence in the HTTP protocol, already in the 1.0 version of the protocol but mainly with the arrival of HTTP/1.1, resulted in a dramatic performance impact for the existing multithreaded web containers. Persistent connections, which means connections that are kept alive by the client between two successive HTTP requests that in turn can be separated in time by several seconds of inactivity (think times), cause that many server threads can be retained by clients even when no requests are being issued and the thread keeps in idle state. The use of blocking I/O operations on the sockets is the cause of this performance degradation scenario. The situation can be solved increasing the number of threads available (which in turn results in a increased contention to acquire exclusion locks) or introducing an inactivity timeout for the established connections, that can be reduced as the server load is increased. When a server is subject to a severe load, the effect of using a short inactivity timeout to the clients lead to a virtual conversion of the HTTP/1.1 protocol into the older HTTP/1.0, with the consequent loss of the performance benefits introduced by connection persistence.

In this model, the effect of closing client connections to free worker threads reduces the probability for a client to complete a session to nearly zero. It is especially important when the server is under overload conditions, where the inactivity timeout is dynamically decreased to the minimum possible in order to free worker threads as quickly as possible, which provokes that all the established connections are closed during think times. This causes a higher competition among clients trying to establish a connection with the server. If we extend it to the length of a user session, we obtain that the probability of finishing it successfully under this architecture is still much lower than the probability of establishing

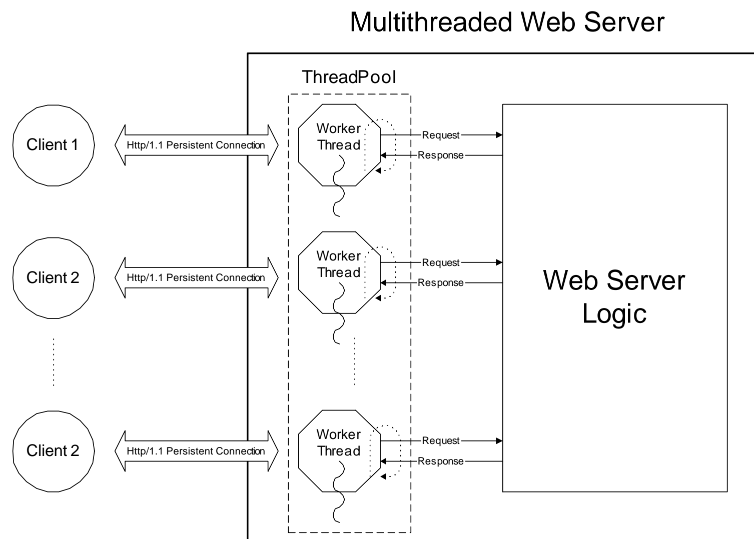


Figure 4.1 Operation of a multithread architecture

each one of the connections it is composed of, driving the server to obtain a really low performance in terms of session completions. This situation can be alleviated increasing the number of worker threads available in the server, but this measure also produces an important increase in the internal web container contention with the corresponding performance slowdown.

4.2.2 Event-driven architecture with non-blocking I/O

On the other hand, the event-driven architecture completely eliminates the use of blocking I/O operations for the worker threads, reducing their idle times to the minimum because no I/O operations are performed for a socket if no data is already available on it to be read. With this model, maintaining a big amount of clients connected to the server does not represent a problem because one thread will never be blocked waiting a client request. With this, the model detaches threads from client connections, and only associates threads to client requests, considering them as independent work units. An example of web container based on this model is described in [83], and a general evaluation of the architecture can be found in [13].

In an event driven architecture, one thread is in charge of accepting new incoming connections. When the connection is accepted, the corresponding socket channel is registered in a channel selector where another thread (the request dispatcher) will wait for socket activity. Worker threads are only awakened when a client request is already available in the socket. When the request is completely processed and the reply has been successfully issued, the worker thread registers again the socket channel in the selector

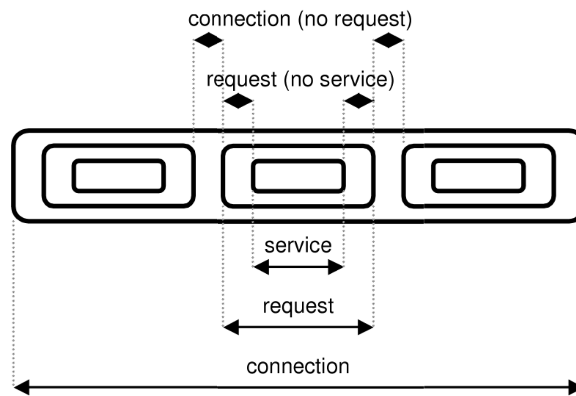


Figure 4.2 Tomcat persistent connection pattern

and gets free to be assigned to new received client requests. This operation model avoids worker threads to keep blocked in socket read operations during client think times and eliminates the need of introducing connection inactivity timeouts and their associated problems.

A remarkable characteristic of the event-driven architectures is that the number of active clients connected to the server is unbounded, so an admission control [28] policy must be implemented. Additionally, as the number of worker threads can be very low (one should be enough) the contention inside the web container can be significantly reduced.

4.3 Performance characterization of secure web applications

In this section we use the monitoring framework presented in Chapter 3 to study the scalability of Tomcat server when subject to a secure workload. The study will demonstrate how crucial a connection management policy results in the performance delivered by an application server. We will illustrate a connection management problem present in the architectural design of most application servers, which was successfully addressed and is now presented as the second contribution of this thesis.

4.3.1 Secure workloads

The SSL [36] protocol provides communications privacy over the Internet using a combination of public-key and private-key cryptography and digital certificates (X.509). This protocol does not introduce a new degree of complexity in web applications' structure because it works almost transparently on top of the socket layer. However, SSL

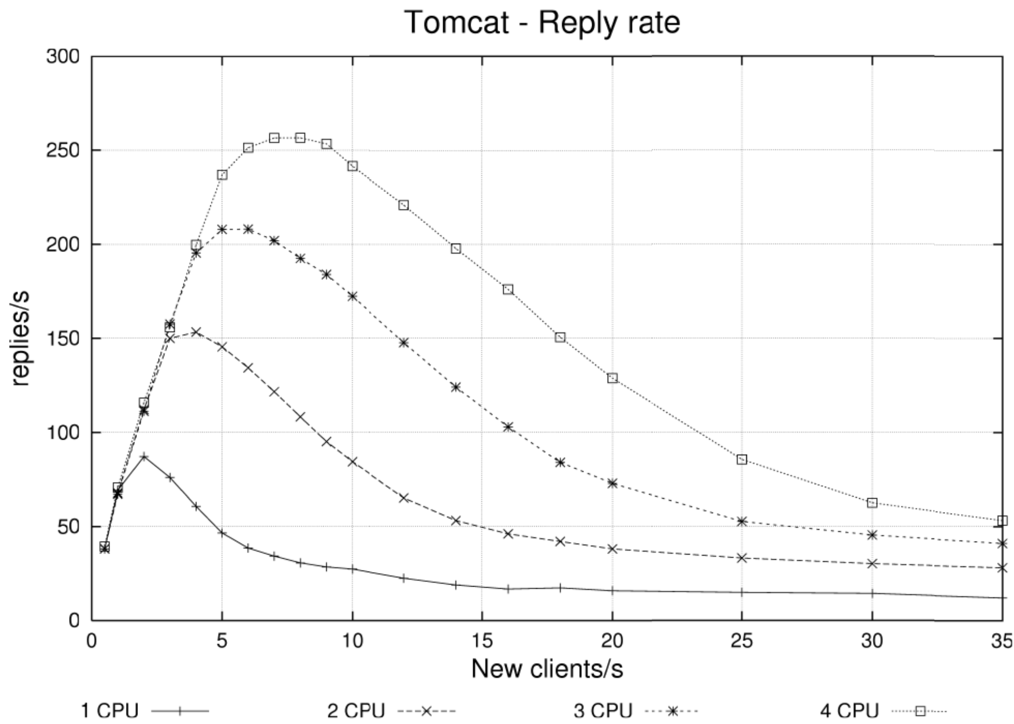


Figure 4.3 Throughput of the original Tomcat with different numbers of processors

remarkably increases the computation time necessary to serve a connection, due to the use of cryptography to achieve its objectives. This increment has a noticeable impact on the server's performance, which has been evaluated in [44]. This study concludes that the maximum throughput obtained when using SSL connections is 7 times lower than when using normal connections. The study also notices that when the server is attending non-secure connections and overloads, it can maintain the throughput if new clients arrive, while if attending SSL connections, the overload of the server provokes degradation of the throughput.

Most of the computation time required when using SSL is spent during the SSL handshake phase, which features the negotiation between the client and the server to establish a SSL connection. Two different SSL handshake types can be distinguished, namely the full SSL handshake and the resumed SSL handshake. The full SSL handshake is negotiated when a client establishes a new SSL connection with the server, and requires the complete negotiation of the SSL handshake, including parts that need a lot of computation time to be accomplished. We have measured the computational demand of a full SSL handshake in a 1.4 GHz Xeon to be around 175 ms. The SSL resumed handshake is negotiated when a client establishes a new HTTP connection with the server

number of processors	new clients/s	throughput (replies/s)
1	2	90
2	4	155
3	6	208
4	8	256

Table 4.1 Number of clients that overload the server and maximum throughput achieved before overload occurs

but resumes an existing SSL connection. As the SSL session ID is reused, part of the SSL handshake negotiation can be avoided, reducing considerably the computation time for performing a resumed SSL handshake. We have measured the computational demand of a resumed SSL handshake in a 1.4 GHz Xeon to be around 2 ms. Note the big difference between the time to negotiate a full SSL handshake compared to the time to negotiate a resumed SSL handshake (175 ms vs. 2 ms).

4.3.2 Evaluation platform

We use Tomcat [6] v5.0.19 as the application server, which follows the multithread paradigm with persistent HTTP connections. The pattern of a persistent connection in Tomcat is shown in Figure 4.2. In this example, three different requests are served through the same connection. The rest of the time (*connection (no request)*) the server is keeping the connection open waiting for another client request. A connection timeout is programmed to close the connection if no more requests are received. Notice that within every *request* the service (execution of the servlet implementing the demanded request) is distinguished from the *request (no service)*. This is the pre and post process that Tomcat requires to invoke the servlet that implements the demanded request.

We run Tomcat (with the RUBiS benchmark deployed on it) on a 4-way Intel XEON 1.4 GHz with 2 GB RAM. Tomcat makes queries to a MySQL [75] v4.0.18 database server using the MM.MySQL v3.0.8 JDBC driver. The MySQL server runs on another machine, which is a 2-way Intel XEON 2.4 GHz with 2 GB RAM. In addition, we have configured a machine with a 2-way Intel XEON 2.4 GHz and 2 GB RAM to run the workload generator (Httperf 0.8). For each experiment performed in the evaluation section (each point in the graphs corresponds to an experiment), the Httperf tool is parametrized with the number of new clients per second that initiate a session with the server (this value is indicated in the horizontal axis of the graphs) and with the workload distribution that these clients must follow (extracted from the Markov model mentioned in the previous section). Then, Httperf emulates the execution of these clients by performing secure

number of processors	throughput (replies/s)
1	13
2	28
3	41
4	53

Table 4.2 Average server's throughput when overloaded

requests to the Tomcat server during a run of 10 minutes. All the machines are connected using a 1 Gbps Ethernet interface and run the 2.6 Linux kernel. For our experiments we use the Sun JVM 1.4.2 for Linux, using the server JVM instead of the client JVM and setting the initial and the maximum Java heap size to 1024 MB. All the tests are performed with the common RSA-3DES-SHA cipher suit, using a 1024 bit RSA key. For the experiments, we configured Tomcat by setting the maximum number of worker threads to be 100 and the connection persistence timeout to be 10 seconds.

4.3.3 Scalability Characterization

The characterization is divided in two parts. The first part, discussed in section 4.3.3.1, is an evaluation of the vertical scalability of the server when running with different number of processors, determining the impact of adding more processors on server overload. The second part, discussed in section 4.3.3.2, consists of a detailed analysis of the server behavior using the performance analysis framework, in order to determine the causes of the server overload when running with different number of processors.

4.3.3.1 Exploring scalability

Figure 4.3 shows the Tomcat throughput as a function of the number of new clients per second initiating a session with the server when running with different numbers of processors. Notice that for a given number of processors, the server's throughput increases linearly with respect to the input load (i.e. the server scales) until a determined number of clients hit the server. At this point, the throughput achieves its maximum value. Table 4.1 shows the number of clients that overload the server and the maximum throughput achieved before overloading when running with different number of processors. Notice that, since secure workloads are CPU-intensive, running with more processors allows the server to handle more clients before overloading, so the maximum throughput achieved is higher. In particular, running with 2 processors increases the maximum throughput achieved by a factor of 1.7, running with 3 processors by a factor of 2.3, and running with 4 processors by a factor of 2.8. Notice that the achieved improvement is not

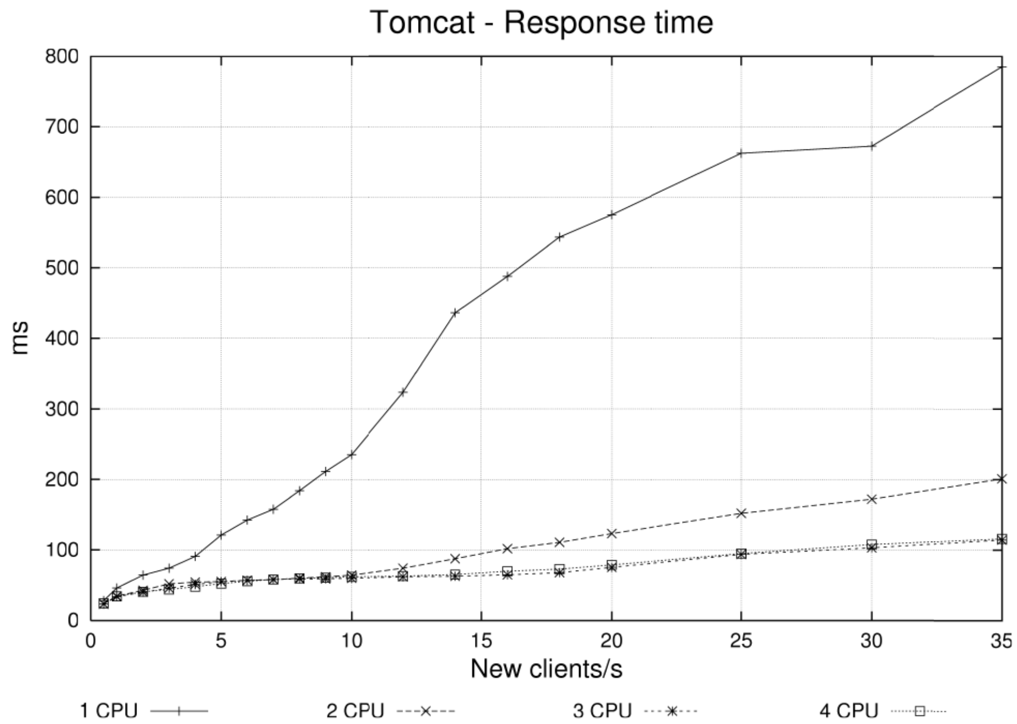


Figure 4.4 Response time of the original Tomcat with different numbers of processors

linear because processors are not the only component limiting the server's performance. When the number of clients that overload the server has been reached, the server's throughput degrades until approximately 20% of the maximum achievable throughput while the number of clients increases, as shown in Table 4.2. This table shows the average throughput obtained when the server is overloaded when running with different numbers of processors. Notice that, although the throughput obtained has decreased in all cases where the server has reached an overloaded state, running with more processors improves the throughput again. When the server is overloaded, running with 2 processors increases the maximum achieved throughput by a factor of 2.1, running with 3 processors by a factor of 3.1, and running with 4 processors by a factor of 4. In this case, a linear improvement is achieved, because processors are the main component limiting the performance.

As well as degrading the server's throughput, the server overload also affects the server's response time, as shown in Figure 4.4. This figure shows the server's average response time as a function of the number of new clients per second initiating a session with the server when running with different numbers of processors. Notice that when the server is overloaded the response time increases (especially when running with one processor) as the number of clients grow.

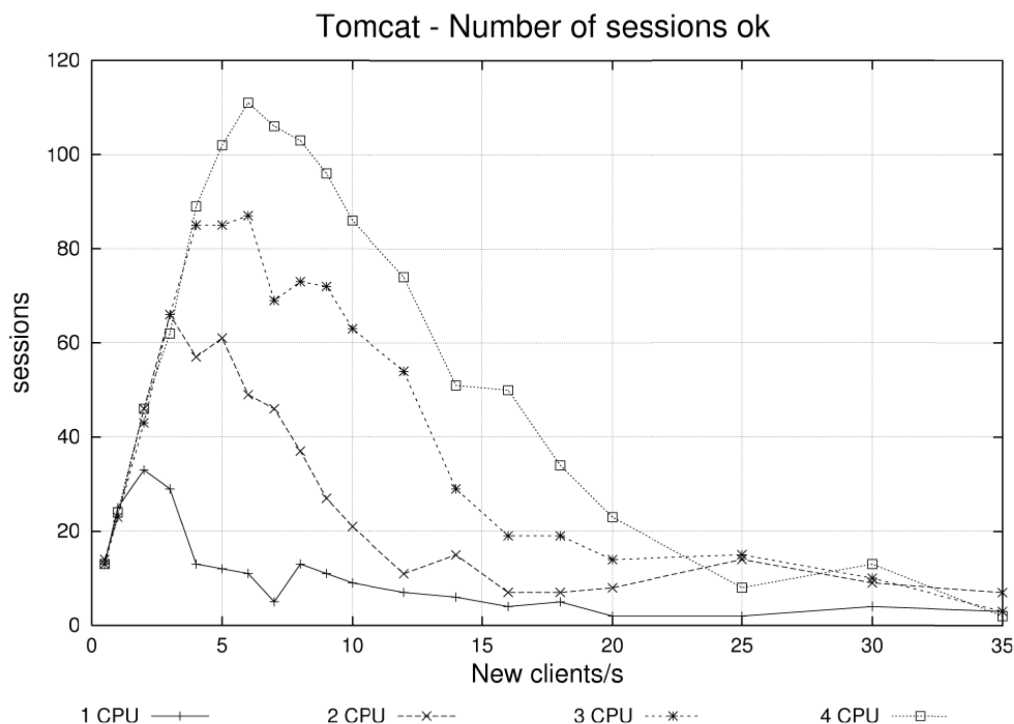


Figure 4.5 Completed sessions by the original Tomcat with different numbers of processors

Server overload has another undesirable effect, especially in e-commerce environments where session completion is a key factor. As shown in Figure 4.5, which shows the number of sessions completed successfully when running with different numbers of processors, when the server is overloaded only a few sessions can finalize completely. Consider the great revenue loss that this fact can provoke for example in an online store, where only a few clients can finalize the acquisition of a product.

4.3.3.2 Analyzing scalability limits

The analysis methodology consists of comparing the server's behavior when it is overloaded with that when it is not, using the performance analysis framework described in Chapter 3. We calculate a series of metrics representing the server's behavior, and we determine which of them are affected while increasing the number of clients. From these metrics, an in-depth analysis is performed looking for the causes of their dependence on the server load.

First of all, using the performance analysis framework, we calculate the average time spent by the server processing a persistent client connection, distinguishing the

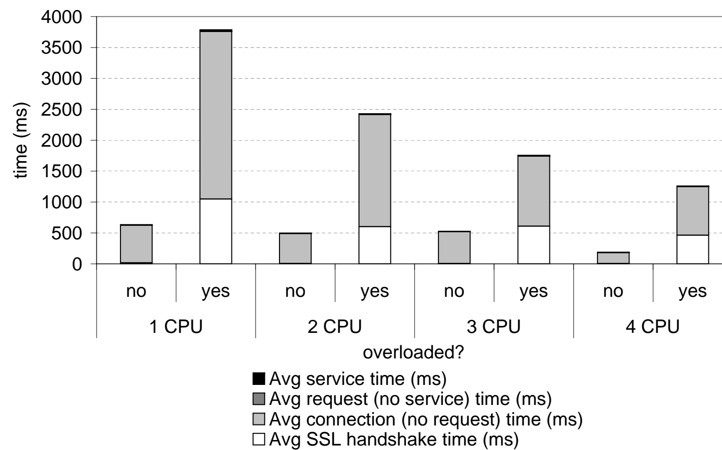


Figure 4.6 Average time spent by the server processing a persistent client connection

time devoted to each phase of the connection (connection phases have been described in Section 4.3.2) when running with different numbers of processors. This information is displayed in Figure 4.6. Notice that when the server is not overloaded, the majority of the time spent to process a client connection is devoted to the *connection (no request)* phase. During this phase the connection remains established waiting for additional requests from the client (i.e. maintaining connection persistence) and for this reason CPU consumption is low.

On the other side, when the server overloads, the average time required to handle a connection increases considerably, mainly at the *SSL handshake* and the *connection (no request)* phases. The proportionally greater increment occurs in the *SSL handshake* phase. In particular, the time spent in this phase increases from 18 ms to 1050 ms when running with one processor, from 7 ms to 602 ms with two processors, from 6 ms to 610 ms with three processors and from 6 ms to 464 ms with four processors. The increment that occurs in the *connection (no request)* phase is proportionally lower, but also noticeable. In particular, the time spent in this phase has increased from 605 ms to 2711 ms when running with one processor, from 485 ms to 1815 ms with two processors, from 515 ms to 1134 ms with three processors and from 176 ms to 785 ms with four processors.

To determine the causes of the great increase in the time spent in the *SSL handshake* phase of the connection, we calculate the percentage of connections that perform a resumed SSL handshake (reusing the SSL Session ID) versus the percentage of connections that perform a full SSL handshake when running with different numbers of processors. This information is shown in Figure 4.7. Notice that when the server runs with one processor without overloading, 97.3% of SSL handshakes reuse the SSL connection, but when it overloads, only 32.9% reuse it. The rest negotiate a full SSL handshake,

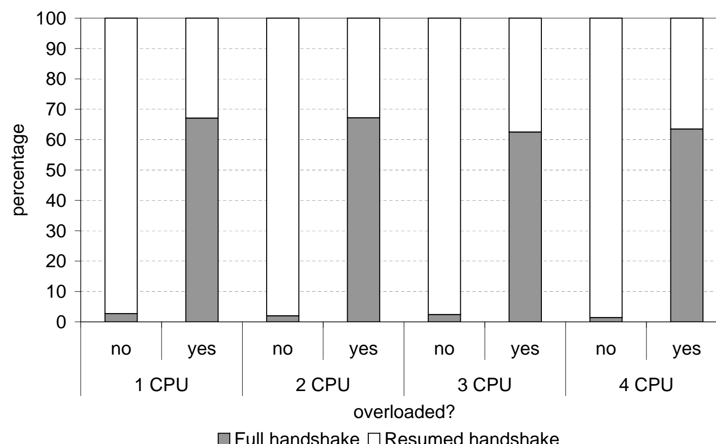


Figure 4.7 Incoming server connections classification depending on the SSL handshake type performed

overloading the server because it cannot supply the computational demand of these full SSL handshakes. Remember the big difference between the computational demand of a resumed SSL handshake (2 ms) and a full SSL handshake (175 ms). The same situation is produced when running with two processors (the percentage of full SSL handshakes increases from 2% to 67.2%), when running with three processors (from 2.4% to 62.5%), and when running with four processors (from 1.4% to 63.5%).

This lack of computational power also explains the increase in the time spent in the *connection (no request)* phase. The connection remains open waiting for additional requests from a given client, but when these requests arrive to the server machine, since there is no available CPU to accept them, they must wait longer in the operating system's internal network structures before being accepted.

We have determined that when running with any number of processors the server overloads when most of the incoming client connections need to negotiate a full SSL handshake instead of resuming an existing SSL connection, requiring a computing capacity that the available processors are unable to supply. Nevertheless, why does this occur from a given number of clients? In other words, why do incoming connections negotiate a full SSL handshake instead of a resumed SSL handshake when attending to a given number of clients? Remember that we have configured the client with a timeout of 10 seconds. This means that if no reply is received in this time, the client's connection will be discarded. When the server is overloaded, it cannot handle the incoming requests before the client timeouts expire. For this reason, in the long run most of the resumed connections with the server will be discarded, and only new clients will arrive at the server. Remember that the initiation of a new client requires the establishment of a new

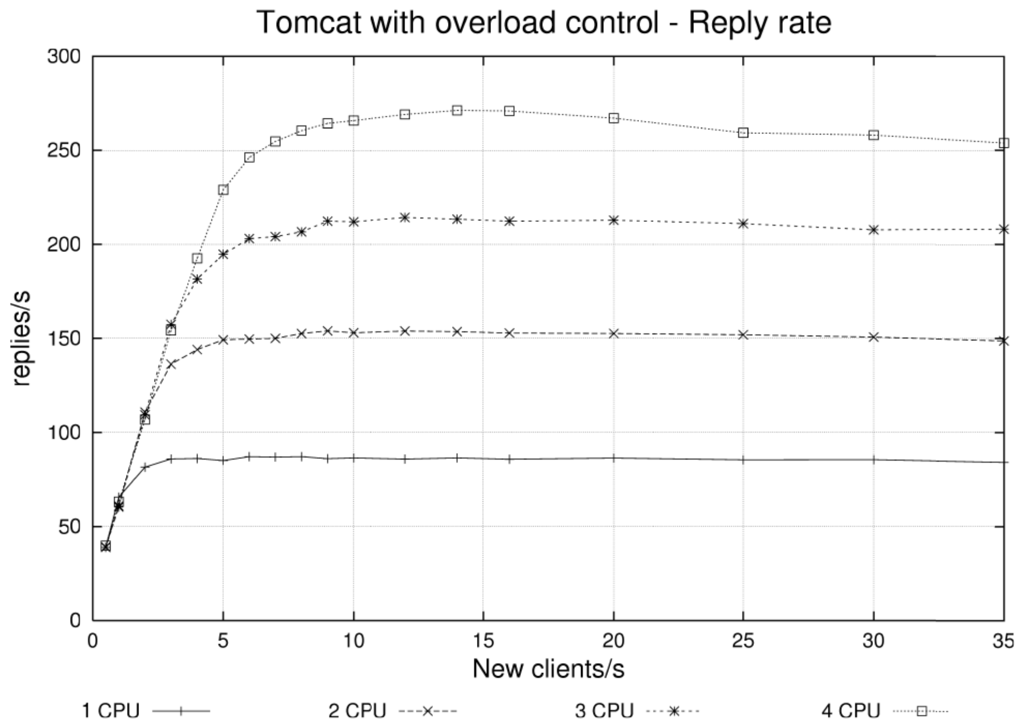


Figure 4.8 Throughput of Tomcat with overload control with different numbers of processors

SSL connection, and therefore the negotiation of a full SSL handshake. Therefore, if the server is overloaded and it cannot handle the incoming requests before the client timeouts expire, this provokes the arrival of a great amount of new client connections that need the negotiation of a full SSL handshake, requiring a computing capacity that the available processors are unable to supply.

This shows that client timeouts have an important effect on the server's performance. One could think about raising client timeouts as the server load increases in order to avoid the degradation of server's performance. However, this is not an appropriate solution for two reasons. Firstly, client timeouts cannot be modified because they are out of the scope of the server administrator. Secondly, even if this modification were feasible, the server will overload anyway, although this will occur when attending to a higher number of clients. In addition, since this solution does not allow differentiating resumed SSL connections from new ones, the prioritization of resumed connections that our approach supports cannot be accomplished, and for this reason, the number of sessions completed successfully will be lower, losing one of the added values of our approach.

The conclusions of this study led to a derived work, described in detail in [45] and out of the scope of this thesis, in which an overload control mechanism was proposed in order

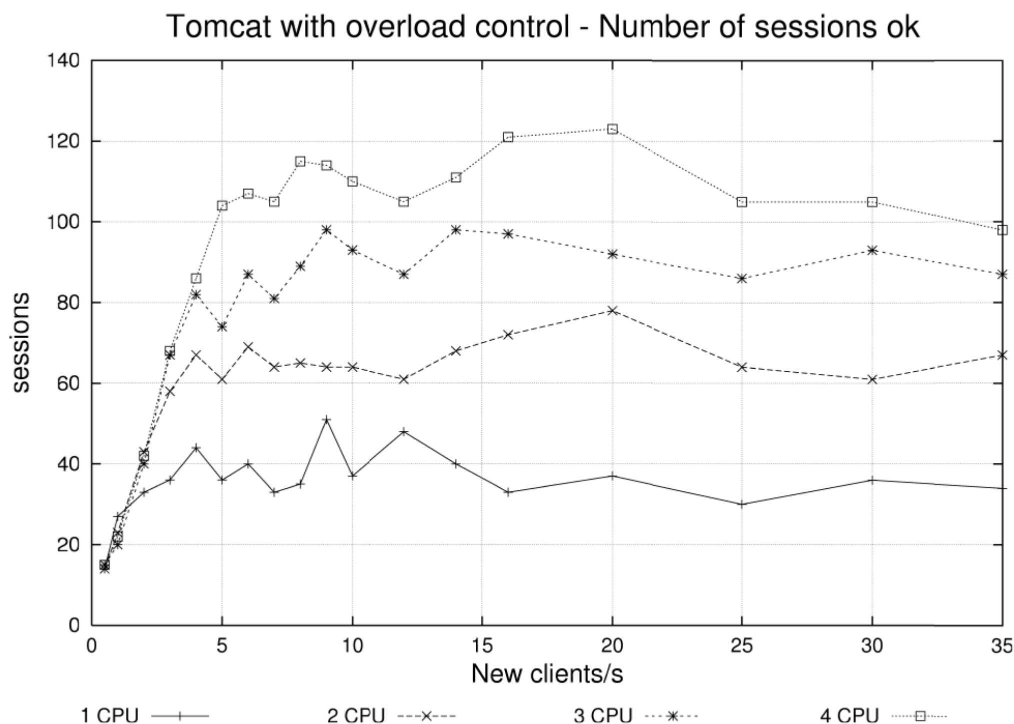


Figure 4.9 Completed sessions by Tomcat with overload control with different numbers of processors

to prioritize resumed connections ahead of new connections requiring a full-handshake to be negotiated. This solution provides better performance for the connected clients, and is specially suited for overloaded commodity scenarios. Figures 4.8 and 4.9 show the sustained throughput delivered by the server under unmodified workload conditions in replies per second, and user session completions respectively.

The performance characterization of secure application servers for commodity environments shown that different strategies can be followed to improve the performance of such an scenario, of which overload control is an example. But in more general terms it shown that the effectiveness of the connection management performed by an application server can result in a big overall performance impact, whenever it uses secure connections or not.This topic is the focus of study of next section.

4.4 Hybrid Architecture

In this section we describe a new hybrid web server architecture that exploits the best of each one of the server architectures discussed in section 4.2. With this hybrid architecture, an event-driven model is applied to receive the incoming client requests. When a request is received, it is serviced following a multithreaded programming model, with the resulting simplification of the web container development associated to the multithreading paradigm. When the request processing is completed, the event-driven model is applied again to wait for the client to issue new requests. This architecture can be used to decouple the management of active connections from the request processing and servicing activity of the worker threads. With this, the web container logic can be implemented following the multithreaded natural programming model and the management of connections can be done with the highest possible performance, without blocking I/O operations and reaching a maximum overlapping of the client think times with the processing of requests. In consequence, the hybrid architecture makes a better use of the characteristics introduced to the HTTP protocol in the 1.1 version, such as connection persistence, with the corresponding reduction in the number of client re-connections (and the corresponding bandwidth save). Additionally some kind of admission control policy must be implemented in the server in order to maintain the system in an acceptable load level, since this architecture does not benefit from the natural overload protection present in the multithread model, intrinsically defined by the thread pool size. A brief operation diagram for this architecture can be seen on figure 4.10.

4.4.1 Implementation on top of Tomcat container

To validate the proposed hybrid architecture we have implemented it inside of Tomcat 5.5, a widely extended web container. Tomcat 5 is built in the top of the Java platform, which provides non blocking I/O facilities across different operating systems in its NIO [13] (Non Blocking I/O) API. The implementation has been carried on by modifying the HTTP connector.

In the scope of this work, two major components of Tomcat are considered: Coyote and Catalina. Coyote is the default HTTP connector and deals with client connection, request parsing and thread pooling. Catalina is the servlet container, and implements most of the web container logic. The implementation of the hybrid server architecture in Tomcat only affects Coyote.

Coyote avoids the use of a dedicated thread to accept new connections, and follows a connection service schema in which worker threads (namely *HttpProcessors*) accept TCP

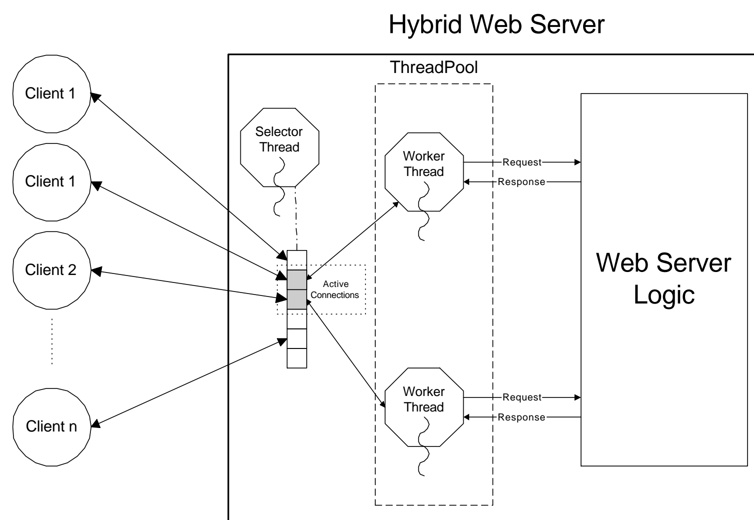


Figure 4.10 Operation of the hybrid architecture

connections and parse and process HTTP requests. *HttpProcessors* are commonly chosen from a pool of threads in order to avoid thread creation overheads. A connection timeout is programmed to close the connection if no more requests are received in a period of time. When a request is parsed, Coyote requires Catalina to process the request and to send the corresponding response to the client.

The implementation of the hybrid architecture modifiesd the original Coyote threading and I/O model. One thread is in charge of accepting and registering, through a channel selector, new incoming connections. When a registered connection becomes active (i.e. it has data available to read so a read operation over the socket will not be blocking), it is dispatched by the selector thread to small pool of *HttpProcessor* threads. Each *HttpProcessor* services only one request for each assigned active connection. The request is read and parsed, always without blocking the thread, and it is send to Catalina who processes the request. When the request is finished the connection is re-registered on the selector and the thread is sent back to the pool until a new active connection is assigned to it.

This implementation presents a major drawback when the system becomes overloaded because in this situation the acceptor thread allows new connections to enter the server faster than Catalina can service them, what results in a quick growth of the number of concurrent connections, which in turn causes a severe response time degradation. In consequence the number of client timeouts grows an the throughput decreases. To avoid this problem we have introduced a simple but effective admission control mechanism (similar to the backpressure technique described in [118]), that prevents the acceptor thread from accepting new connections while all *HttpProcessors* are busy.

4.4.2 Performance evaluation

In our experiments we evaluate how the hybrid architecture, implemented on top of the Tomcat container compares to the original multithread Tomcat architecture. We use two different applications: one offering static content (Surge) and one offering dynamic (Servlets) content (RUBiS). Notice that although the results presented in section 4.3 were for secure workloads, the evaluation presented in this section is based on plain workloads. The use of secure connections combined with non-blocking I/O in Java is a challengig task by itself, and is a problem that has been addressed in the derived work described in [12].

4.4.2.1 Testing platform

For the experiments, we configured Httperf setting the client timeout value to 10 seconds. Each individual benchmark execution had a fixed duration of 30 minutes for the dynamic content tests and 10 minutes for the static content experiments.

We used a 4-way Intel Xeon 1.4 GHz with 2GB RAM to run the web servers and a 2-way Intel XEON 2.4 GHz with 2 GB RAM to run the benchmark clients. For the benchmark applications that require the use of a database server, a 2-way Intel XEON 2.4 GHz with 2 GB RAM was used to run MySQL v4.0.18, with the MM.MySQL v3.0.8 JDBC driver. All the machines were running a Linux 2.6 kernel, and were connected through a switched Gbit network. The SDK 1.5 from Sun was used to develop and run the web servers.

The servers were tested in two different scenarios, one to evaluate the server performance for a static content application and another for a dynamic content environment. The requests issued by httperf were extracted from the Surge workload generator for the static content scenario and from the RUBiS application for the dynamic content environment. Section 2.2.1 describes in detail these applications.

It can be stated that static content applications are usually characterized by the short length of the user sessions as well as by the low computational cost of each request to be serviced. In opposite, dynamic content applications tend to show long length user sessions (an average of 300 requests per session in front of the 6 requests per session for the static workload in ourt experiments) as well as a high computational cost associated to each processed request (including embedded requests to external servers, such as databases).

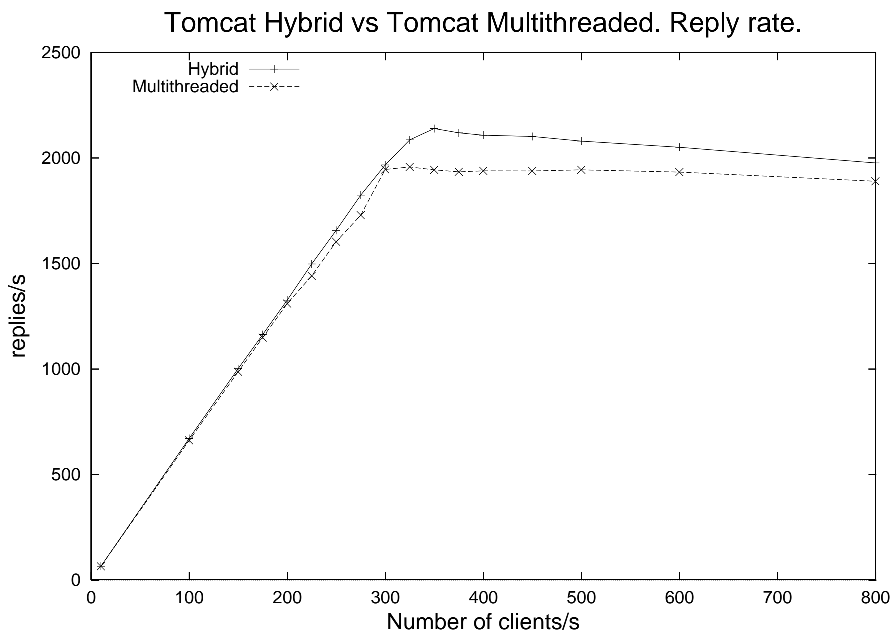


Figure 4.11 Throughput comparison under an static content workload

4.4.2.2 Static content

The first performance metric evaluated for this scenario is the throughput obtained for each architectural design, measured in replies per second. The results for both architectures are shown in figure 4.11. It can be observed that the multithreaded architecture obtains a slightly lower performance than the hybrid one in terms of throughput, yet still very close. It is remarkable that the same throughput is obtained by the hybrid architecture with a thread pool size of only 10 threads, while in the multithreaded architecture requires 500 threads to obtain this result.

If we move from the throughput to the response time observed for each implementation, we can see that the hybrid architecture offers a clearly better result than the multithreaded one, as it can be seen in figure 4.12. When the system is not saturated (under a load equivalent to 300 new clients per second), the response time for the multithreaded server is slightly better than for the hybrid design, possibly because of the overhead introduced by the extra operations that the hybrid server must do to register the sockets in the selector and to switch them between blocking and non-blocking mode when moving from multithreaded to the event-driven and reverse. This effect would be dispelled in a WAN environment, where the latencies are much higher than in the Gbit LAN used for the experiments. When the system is saturated, beyond 300 new clients/s, the benefits of the hybrid architecture turn up and the response time reduction with respect to the multithreaded version of the server is of about a 50%, moving from a 300 ms average

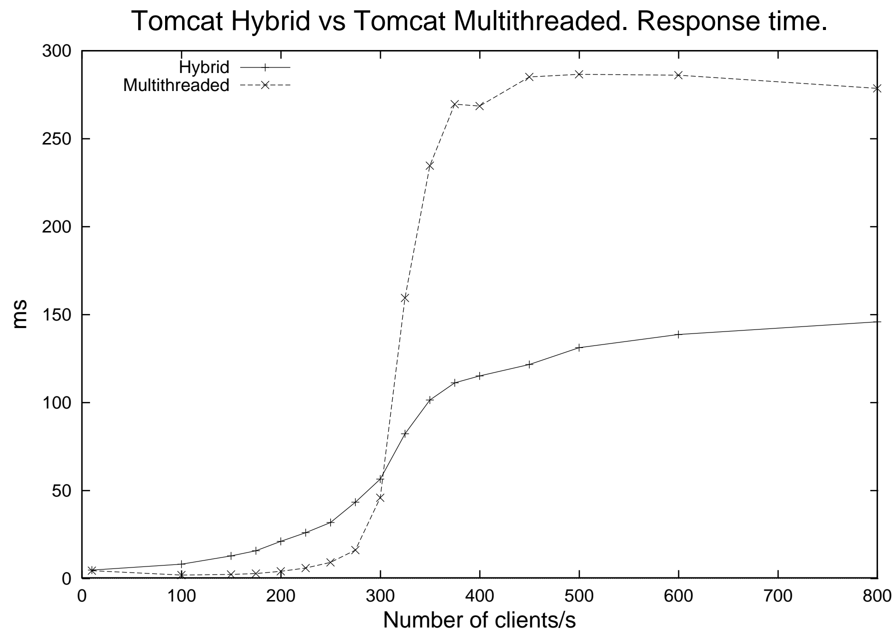


Figure 4.12 Response time under an static content workload

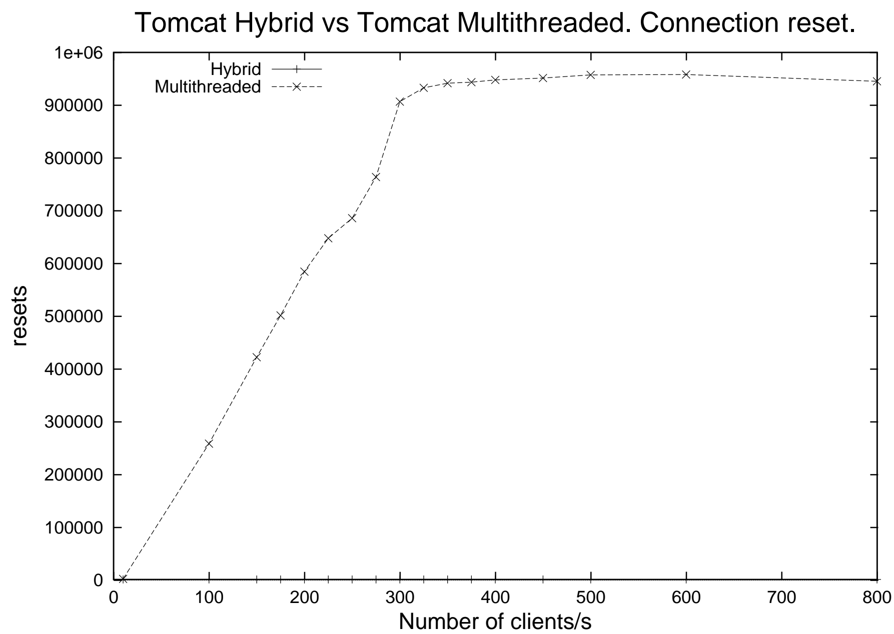


Figure 4.13 Number of connections closed by the server by a timeout expiration

response time for the multithreaded architecture to a 150 ms response time for the hybrid design.

What is making this big difference in response time between the two architectures is shown in figure 4.13. This figure shows the amount of connections that have been closed by the server because of a too long client inactivity period, causing the server timeout to expire and obligating the client to be reconnected to resume its session. As it can be seen, while the hybrid architecture is not producing this kind of situation (zero errors are detected by the benchmark client), a high number of errors are detected for the multithreaded architecture. This situation can be explained by the need of the multithreaded design to free worker threads to make them available for new incoming connections. This causes that the client inactivity periods must be avoided by closing the connection and requiring the client to resume its session with a new connection establishment when necessary. In the hybrid architecture, that assigns client requests as work units to the worker threads instead of client connections, this situation is naturally avoided and the cost of keeping a client connection event in periods of inactivity is equivalent to the cost of keeping the connection socket opened. The effect of this characteristic for the hybrid architecture is that all the re-connections are eliminated.

4.4.2.3 Dynamic content

Dynamic content applications implement a higher complexity and more developed semantics than static ones, which is usually translated into longer user sessions and involves that the common performance metrics are partially redefined in terms of business concepts. This means that e-commerce applications are more concerned about sales or business transactions than about more technical metrics such as the throughput or the response time offered by the server.

For this experiment, we consider that one of the most important metrics for an auction website (the scenario reproduced by RUBiS, see subsection 4.4.2.1 for more details) is the number of user sessions that are completed successfully. Each user that can complete its navigation session represents a potential bid for an item and in consequence a higher profit for the auction company.

Looking at figure 4.14, it can be seen that the throughput offered by both architectural designs is very similar, although the multithreaded architecture shows a slightly better performance when the server is saturated.

Looking at the number of sessions completed per second, in figure 4.15, it can be seen that the amount of successfully finished sessions reached by the hybrid architecture is clearly higher than the amount reached by the multithreaded design, especially beyond

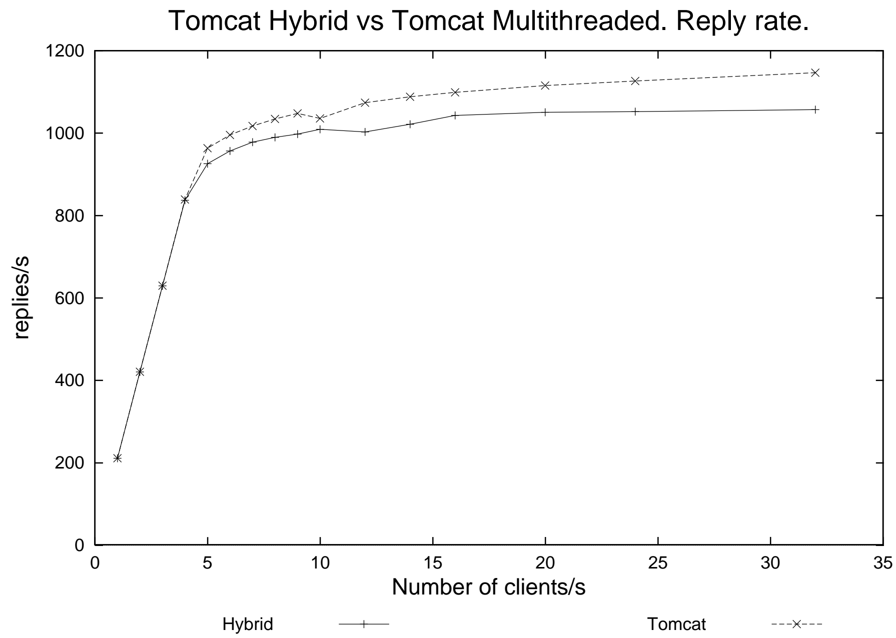


Figure 4.14 Reply throughput comparison under a dynamic content workload

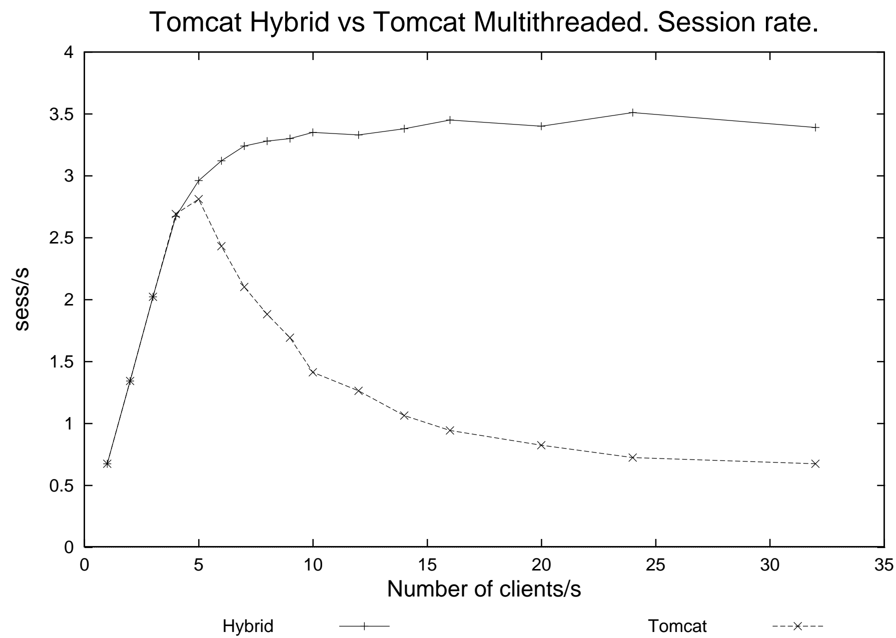


Figure 4.15 Successfully completed session rate under a dynamic content workload

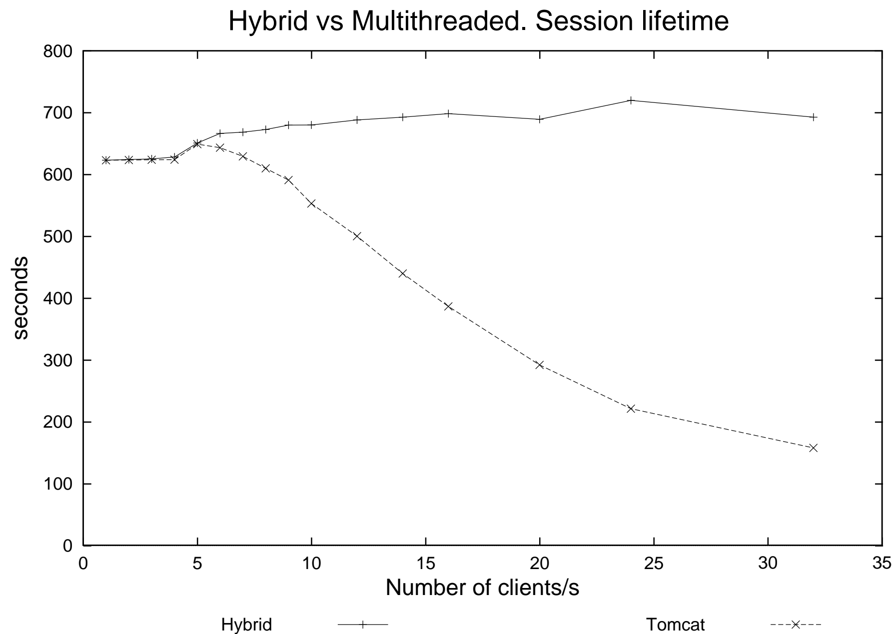


Figure 4.16 Lifetime comparison for the sessions completed successfully under a dynamic content workload

saturation. As it can be observed, the multithreaded architecture tends to decrease the number of completed sessions per second as the workload intensity is increased. This can be explained because under a pure multithreaded design the worker threads are obligated to close the client connections in order to be freed and rest available for new incoming connection requests. This situation, under high loads, leads to a scenario where the clients whose connections have been closed by the server, start experiencing problems to be reconnected because the amount of active clients trying to establish a connection is remarkably higher than the amount of worker threads disposed in the server. If this is extended to the amount of connections required to complete a long user session, characteristic of dynamic applications, the probability of being able to finish the session successfully is reduced to near zero.

This explanation to the difference of performance between the two architectural designs in terms of session completion is supported by the results shown in figure 4.16 that indicate that the sessions completed by the multithreaded server are significantly shorter than the sessions completed by the hybrid server. This explains that the reply rate for the multithreaded server can be sustained even when the session rate is remarkably reduced because it proves that the sessions completed by the multithreaded server are those ones with less requests (the shorter ones). Completing only short sessions means that many active clients finishes their sessions unsuccessfully, which in turn may result in an important amount of unsatisfied clients that have received a very poor quality of

service.

4.5 Related Work

The use of event-driven architectures for web servers is an already explored area. Flash[83] is an asymmetric multi-process event-driven web server which exploits the creation of a set of helper processes to avoid thread-blocking in the main servicing process. Haboob[118] is also an event-driven web server, based on the concepts of staged event-driven servers introduced by SEDA[118]. JAWS[52] web server uses the Proactor[51] pattern to easily construct an event-driven concurrency mechanism. In [119], the authors propose a design framework to construct concurrent systems based on the combination of threading and event-driven mechanisms to achieve both high throughput and good parallelism. Some advanced topics about performance issues involving event-driven architectures have been studied in [67], [127] and [50]. The introduction of a non-blocking I/O API in the J2SE was preceded by the development of an extensively used API called NBIO[117] (Non-Blocking I/O), which was used to create the standard API NIO. In [94] the authors propose a light-weight threading model for java applications designed to overcome the limitations found in massively multi-threaded Java applications. None of the previously commented articles evaluate the causes of performance scalability, with respect to the workload intensity and the number of processors, of the event-driven architectures for Java web servers in comparison to the more commonly used multithreaded servers.

Related with the vertical scalability covered in this chapter, some works have evaluated this scalability on web servers or application servers. For example, [47] only consider static web content and the evaluation is limited to a numerical study without performing an analysis to justify the scalability results obtained. [3] and [26] provide a quantitative analysis based on general metrics of application server execution collecting system utilization statistics (CPU, memory, network bandwidth, etc.). These statistics may allow the detection of some application server bottlenecks, but this coarse-grain analysis is often not enough when dealing with more sophisticated performance problems. The influence of security on application server scalability has been covered in some works. For example, the performance and architectural impact of SSL on the servers in terms of various parameters such as throughput, utilization, cache sizes and cache miss ratios has been analyzed in [57], concluding that SSL increases computational cost of transactions by a factor of 5-7. The impact of each individual operation of TLS protocol in the context of web servers has been studied in [31], showing that key exchange is the slowest operation

in the protocol. [43] analyzes the impact of full handshake in connection establishment and proposes caching sessions to reduce it.

4.6 Summary

In this chapter we have presented the second contribution of this thesis: a hybrid web container architecture that combines the best characteristics of both a multithreaded design and an event-driven model. The proposed implementation into the Tomcat 5.5 code offers a slightly better performance than the original multithreaded Tomcat server when it is tested for a static content application, and a remarkable performance increase when it is compared for a dynamic content scenario, where each user session failure can be put into relation with business revenue losses. Additionally, the natural way of programming introduced by the multithreading paradigm can be maintained for most of the web container code. But even more important, we have shown how the hybrid architecture naturally adapts to dynamically changing workload conditions without need to be reconfigured. This desired adaptability property reduces the need of human interaction in order to keep the server properly configured.

A preliminary study, that motivated this contribution, consisted of measuring Tomcat's vertical scalability (i.e. adding more processors) when using SSL and analyzing the effect of this addition on the server's scalability. The results confirmed that, since secure workloads are CPU-intensive, running with more processors makes the server able to handle more clients before overloading, with the maximum achieved throughput improvement ranging from 1.7 to 2.8 for 2 and 4 processors respectively. In addition, even when the server has reached an overloaded state, a linear improvement on throughput can be obtained by using more processors. The second part involved the analysis of the causes of server overload when running with different numbers of processors by using a performance analysis framework. The analysis revealed that the server can be easily overloaded if connections are not properly managed, demonstrating the convenience of developing advanced connection management strategies to overcome such a complicated scenario.

The work performed in this area is described in the following publications:

[19] D. Carrera, V. Beltran, J. Torres, E. Ayguade. **A Hybrid Web Server Architecture for e-Commerce Applications**. 11th International Conference on Parallel and Distributed Systems (ICPADS'05), 2005

[20] D. Carrera, V. Beltran, J. Torres, E. Ayguade. **A Hybrid Connector for Efficient Web Servers**. Special Issue on High Performance Computing in Parallel and Distributed Systems of the International Journal of High Performance Computing and Networking (IJHPCN). Issue 5/6 of 2007, Vol. 5. ISSN: 1740-0562

has been motivated by the work described in:

[45] J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. **Designing an overload control strategy for secure e-commerce applications**. Computer Networks 51, 15 (Oct. 2007), 4492-4510

and has resulted in the following derived work:

[13] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. **Evaluating the Scalability of Java Event-Driven Web Servers**. In 2004 International Conference on Parallel Processing (ICPP'04), 2004

[12] V. Beltran , D. Carrera, J. Guitart, J. Torres and E. Ayguadé. **A Hybrid Web Server Architecture for Secure e-Business Web Applications**. 1st International Conference on High Performance Computing and Communications (HPCC'05), 2005 Lecture Notes in Computer Science, pp. 366-377, vol. 3726, no. 3726, September 2005. ISSN: 0302-9743 ISBN: 978-3-540-29031-5.

[14] V. Beltran , J. Torres and E. Ayguadé. **Understanding Tuning Complexity in Multithreaded and Hybrid Web Servers**. 22nd International Parallel and Distributed Symposium (IPDPS'08), 2008

Chapter 5

Integrated management of heterogeneous workloads

5.1 Introduction

Nowadays, many organizations rely on a heterogeneous set of applications to deliver critical services to their customers and partners. For example, in financial institutions, transactional web workloads are used to trade stocks and query indices, while computationally intensive non-interactive workloads are used to analyze portfolios or model stock performance. Due to intrinsic differences among these workloads, today they are typically run on separate dedicated hardware and managed using workload specific management software.

A traditional approach to resource management for heterogeneous workloads is to configure resource allocation policies that govern the division of computing power among web and non-interactive workloads based on temporal or resource utilization conditions. With a temporal policy, the resource reservation for web workloads varies between peak and off-peak hours. Resource utilization policies allow non-interactive workload to be executed when resource consumption by web workload falls below a certain threshold. Typically, resource allocation is performed with a granularity of a full server machine, as it is difficult to configure and enforce policies that allow server machines to be shared among workloads. Coarse-grained resource management based on temporal or resource utilization policies has previously been automated [7, 48].

Once server machines are assigned to either the web or the non-interactive workload, existing resource management policies can be used to manage individual web and non-interactive applications. In the case of web workloads, data centers use admission control, flow control, load balancing, and application placement mechanisms, which are controlled using a variety of policies. Due to the increasing size and heterogeneity of datacenters, customers demand that these policies be fine grained, automatic, and dynamic. In the case of non-interactive workloads, the techniques involve job scheduling, which may be performed based on various existing scheduling disciplines [39].

To efficiently utilize the computing power of their datacenters, organizations demand management solutions that permit these kinds of workloads to run on the same physical hardware and be managed using a resource management technology to determine the most effective allocation of resources to particular workloads. To effectively manage heterogeneous workloads, we need a solution that combines flow control and dynamic placement techniques with job scheduling.

Integrated automated management of heterogeneous workloads is a challenging problem for several reasons. First, performance goals for different workloads tend to be of different types. For interactive workloads, goals are typically defined in terms of average or percentile response time or throughput over a certain time interval, while

performance goals for non-interactive workloads concern the performance of individual jobs. Second, the time scale of management is different. Due to the nature of their performance goals and short duration of individual requests, interactive workloads lend themselves to automation at short control cycles. Non-interactive workloads typically require calculation of a schedule for an extended period of time. Extending the time scale of management requires long-term forecasting of workload intensity and job arrivals, which is a difficult if not impossible problem to solve. Server virtualization helps us avoid this issue by providing automation mechanisms by which resource allocation may be continuously adjusted to the changing environment. Third, to collocate applications on a physical resource, one must know the applications' behavior with respect to resource usage and be able to enforce a particular resource allocation decision. For web applications, with the help of an L7 gateway provided by modern application server middleware [116], one can rather easily observe workload characteristics and, taking advantage of similarity of web requests and their large number, derive reasonably accurate short-time predictions regarding the behavior of future requests. Non-interactive jobs do not exhibit the same self-similarity and abundance properties, hence predicting their behavior is much harder. Enforcing a resource allocation decision for web workloads can also be achieved relatively easily by using flow control mechanisms [70, 81]. Server virtualization gives us similar enforcement mechanisms for non-interactive applications.

While server virtualization allows us to better manage workloads to their respective SLA goals, it also introduces considerable challenges in order to use it effectively. They concern the configuration and maintenance of virtual images, infrastructure requirements to make an effective use of the available automation mechanisms, and the development of algorithmic techniques capable of utilizing the larger number of degrees of freedom introduced by virtualization technologies.

In the third contribution of this thesis we address the problem of managing heterogeneous workloads in a virtualized data center. We consider two different workloads: transactional applications and long running jobs. We present a technique that permits collocation of these workload types on the same physical hardware. Our technique dynamically modifies workload placement by leveraging control mechanisms such as suspension and migration, and strives to optimally trade off resource allocation among these workloads in spite of their differing characteristics and performance objectives. We achieve this goal by using utility functions, which permit us to compare the performance of various workloads, and which are used to drive allocation decisions. We demonstrate that our technique maximizes heterogeneous workload performance while providing service differentiation based on high-level performance goals. Notice that the importance

of this point is that this is the first proposal that uses utility functions to make the performance of long running workloads directly comparable to the performance of transactional workloads, overcoming their intrinsic differences.

The system has been implemented and integrated with a commercial application server middleware [116] that provides transparent application replication via clustering and request routing, session and transaction state management, application server quiesce mechanisms and takes advantage of recent advances in fields of resource usage profiling for web applications [80] and performance modeling and overload protection [81]. Thanks of the existence of these services and their resilience to dynamic configuration changes, we can concentrate on the problem of integrated management of heterogeneous workloads, which implies deciding application placement and particular resource allocation dynamically, without having to explicitly address these critical issues. Our system is capable of taking advantage of a series of virtualization features, that were already introduced and enumerated in section 2.3.2.

5.2 System architecture

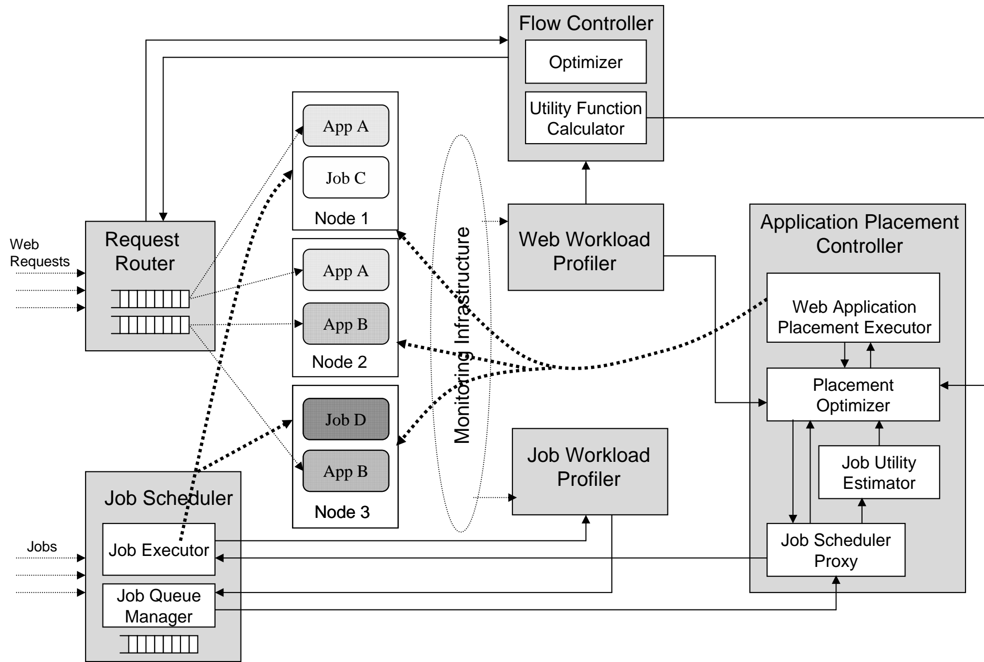


Figure 5.1 Architecture of the system

Before describing the integrated management technique proposed in this thesis, we present the environment on top of which our work relies. This environment is provided by a state-of the art application server middleware [116], which is composed of a large number of cooperative components. Thanks of the existence of this environment, in our work we can concentrate on the problem of managing workloads. From the point of view of this work, the *application placement controller* (APC) is the most important component of the system. It provides the decision-making logic that affects placement of workloads.

Figure 5.1 shows a simple example of a system we consider in our work. The managed system includes a set of heterogeneous server machines, referred to henceforth as *nodes*. Web applications, which are served by application servers, are replicated across nodes to form application server clusters. Requests to these applications arrive at an entry router which may be either an L4 or L7 gateway that distributes requests to clustered applications according to a load balancing mechanism. Long-running jobs are submitted to the job scheduler, placed in its queue, and dispatched from the queue based on the resource allocation decisions of the management system.

The system takes advantage of an overload protection mechanism that can prevent a web application from utilizing more than the allocated amount of resources. Such

overload protection may be achieved using various mechanisms including admission control [37], flow control [70, 81], or OS scheduling techniques [71]. Server virtualization mechanisms could also be applied to enforce resource allocation decisions on interactive applications.

In the system considered in our work, overload protection for interactive workloads is provided by an L7 request router which implements a flow control technique. The router classifies incoming requests into flows depending on their target application and service class, and places them in per-flow queues. Requests are dispatched from the queues based on weighted-fair scheduling discipline, which observes a system-wide concurrency limit. The concurrency limit ensures that all the flows combined do not use more than their allocated resource share. The weights further divide the allocated resource share among applications and flows.

Both the concurrency limit and scheduling weights are dynamically adjusted by the *flow controller* in response to changing workload intensity and system configuration. The flow controller builds a model of the system that allows it to predict the performance of the flow for any choice of concurrency limit and weights. This model may also be used to predict workload performance for a particular allocation of CPU power. In our work, we use this functionality of the flow controller to come up with a utility function for each web application, which gives a measure of application happiness with a particular allocation of CPU power given its current workload intensity and performance goal. The flow control technique implemented by the flow controller and request router are described in [81] and further enhanced in [79], and will not be further discussed in our work.

Long-running jobs are submitted to the system via the *job scheduler*, which, unlike traditional schedulers, does not make job execution and placement decisions. In our system, the job scheduler only manages dependencies among jobs and performs resource matchmaking. Once dependencies are resolved and a set of eligible nodes is determined, jobs are submitted to the *application placement controller* (APC).

Each job has an associated performance goal. Currently we only support completion time goals, but we plan to extend the system to handle other performance objectives. From this completion time goal we derive an objective function which is a function of actual job completion time. When job completes exactly on schedule, the value of the objective function is zero. Otherwise, the value increases or decreases linearly depending on the distance of completion time from the goal.

The job scheduler uses APC as an adviser to where and when a job should be executed. When APC makes a placement decision, actions pertaining to long-running jobs are returned to the scheduler and put into effect via its *job executor* component. The job

executor monitors job status and makes it available to APC for use in subsequent control cycles.

From the point of view of this work, APC is the most important component of the system. It provides the decision-making logic that affects placement of both web and non-interactive workloads. To learn about jobs in the system and their current status, APC interacts with the job scheduler via its *job scheduler proxy*. The *placement optimizer* calculates the placement that maximizes the minimum utility across all applications. In [58], we have introduced a technique that provides such dynamic placement for web applications. It is able to allocate CPU and memory to applications based on their CPU and memory requirements, where memory requirement of an application instance is assumed not to depend on the intensity of workload that reaches the instance. APC used in this system is a version of the controller presented in [58] that has been enhanced in several ways. We modified the algorithm inputs from application CPU demand to a per-application utility function of allocated CPU speed. Permitting resource requirements to be represented by non-linear utility functions allows us to better deal with heterogeneous workloads which may differ in their sensitivity to a particular resource allocation. The attention to workload sensitivity to resource allocation is important when system is overloaded and resource requirements of some applications cannot be satisfied in full. We also changed the optimization objective from maximizing the total satisfied CPU demand to maximizing the minimum utility across all applications, which focuses the algorithm on ensuring fairness and, in particular, prevents it from starving some applications. In addition, we have improved the heuristics used be the algorithm, which resulted in a significant reduction of its computational complexity.

Since APC is driven by utility functions of allocated CPU demand and (for non-interactive workloads) we are only given objective functions of achieved completion times, we need a way to map completion time into CPU demand, and vice versa. Recall that for web traffic we already have a similar mechanism, provided by the flow controller. The required mapping is very hard to obtain for non-interactive workloads, because the performance of a given job is not independent of CPU allocation to other jobs. After all, when not all jobs can simultaneously run in the system, the completion time of a job that is waiting in the queue for other jobs to complete before it may be started depends on how quickly the jobs that were started ahead of it complete, hence it depends on the CPU allocation to other jobs. In our system, we have implemented heuristics that allow us to estimate CPU requirements for long-running jobs for a given value of utility function. We use this estimation to obtain a set of data-points from which we extrapolate the utility function. The utility function allows us to evaluate a placement of long-running jobs with

respect to how well it is likely to satisfy their SLAs. The process of calculating the utility function is rather involved, and due to space limitations it will not be described in our work.

To manage web and non-interactive workloads, APC relies on the knowledge of resource consumption by individual requests and jobs. Our system includes profilers for both kinds of workloads. The *web workload profiler*, which was introduced in [80], obtains profiles for web requests in the form of the average number of CPU cycles consumed by requests of a given flow. The *job workload profiler*, which is a subject of our ongoing research, obtains profiles for jobs in the form of the number of CPU cycles required to complete the job, the number of threads used by the job, and the maximum CPU speed at which the job may progress.

5.3 The placement problem

In this section we describe the heuristic we use to solve the placement problem, that is, to decide where applications, either long running jobs or application server instances, are placed in the system. In addition, it decides the load distribution, that is, the particular resource allocation for each application.

Finding an optimal solution to the placement problem is a non-linear optimization objective when non-linear continuos utility functions are used to represent the satisfaction of applications. The problem is known to be NP-hard and requires the use of either non-deterministic algorithms or heuristics to be solved. While other approaches [114] use non-deterministic algorithms to solve a similar problem (but restricted to transactional workloads only), we use a heuristic to find a solution.

The basic algorithm, as described in following sections, is surrounded by the *Placement control loop*, which resides within the Executor in Figure 5.1. This is designed to have the Application Placement Controller periodically inspect the system to determine if placement changes are now required to better satisfy the changing extant load. The period of this loop is configurable—however, this loop is interrupted when the configuration of the system is changed, thus ensuring that the system is responsive to administrative changes.

5.3.1 Problem statement

Each time the algorithm runs, we are given a set of nodes, $\mathcal{N} = \{1, \dots, N\}$ and a set of applications $\mathcal{M} = \{1, \dots, M\}$. We use n and m to index into the sets of machines and applications, respectively. With each machine n we associate its memory and CPU capacities, Γ_n and Ω_n. Both values measure only the capacity available to workload controlled by placement controller. Capacity used by other workloads is subtracted prior to invoking the algorithm. With each application, we associate its load independent demand, γ_m that represents the amount of memory consumed by this application whenever it is started on a machine. CPU requirements of applications are given in the generic form of utility functions, briefly discussed in section 2.3.1. More details about the particular utility functions for transactional and long running workloads will be provided in sections 5.4.1 and 5.4.2 correspondingly.

We use symbol $\mathcal{I}$ to denote a placement matrix of applications on machines. Cell $\mathcal{I}_{m,n}$ represents the number of instances of application m on machine n. We use symbol L to represent a load placement matrix. Cell $L_{m,n}$ denotes the amount of CPU speed consumed by all instances of application m on machine n.

We define load placement utility function $U(L) = (u_{m_1}(L), \ldots, u_{m_M}(L))$, where applications inside the vector are ordered according to increasing $u_{m_k}(L)$. Utility $U(L) = (u_{m_1}(L), \ldots, u_{m_M}(L))$ is greater than utility $U(L') = (u_{m'_1}(L'), \ldots, u_{m'_M}(L'))$ if there exists k such that $u_{m_k}(L) > u_{m'_k}(L')$ and for all $l < k$, $u_{m_l}(L) = u_{m'_l}(L')$. This induces a lexicographic order of utility vectors.

Given an instance placement, all controllers in the system try to find the best possible load distribution. Hence, utility of instance placement is $U(\mathcal{I}) = \max_L U(L)$, where load distribution defined by any considered L does not overload CPU capacity of any node and for any application m and node n, $L_{m,n}$ is not greater than the amount of CPU demand that may actually be satisfied using $\mathcal{I}_{m,n}$.

We define current placement $\mathcal{I}^{old}$, for which each cell $\mathcal{I}^{\text{old}}_{m,n}$ represents the number of instances of application m that are currently placed on machine n. Notice we want to calculate a new placement $\mathcal{I}$ that may differ from current placement $\mathcal{I}^{old}$. Each application to be stopped or started as a result of a new proposed placement, namely a *placement change*, results in an overload for the system, and is something to be avoided if possible. To measure the number of placement changes given the current placement and a suggested new placement, we define the distance δ between two placement matrices $\mathcal{I}^1$, with applications $\mathcal{M}_1$ and nodes $\mathcal{N}_1$, and $\mathcal{I}^2$, with applications $\mathcal{M}_2$ and nodes $\mathcal{N}_2$, as:

$$\delta(\mathcal{I}^1, \mathcal{I}^2) = \sum_{m \in \mathcal{M}_1 \bigcup \mathcal{M}_2, n \in \mathcal{N}_1 \bigcup \mathcal{N}_2} |\mathcal{I}^1_{m,n} - \mathcal{I}^2_{m,n}| \tag{5.1}$$

The objective of placement controller is to find $\mathcal{I}$ and L that maximize $U(\mathcal{I})$. In addition, the algorithm tries to minimize the number of placement changes, which are time-consuming and CPU-intensive. Remember that a lexicographic order is defined for utility vectors, and that $U(\mathcal{I}) = \max_L U(L)$.

The objective function for the placement problem can more formally be defined as finding $\mathcal{I}$, subject to the constraints defined later in this section, such that:

$$\max U(\mathcal{I}) \tag{5.2}$$

$$\min \delta(\mathcal{I}, \mathcal{I}^{\text{old}}) \tag{5.3}$$

Considering the form taken by the utility function, our problem formulation is an extension of $\max\min$ criterion and differs from it by explicitly stating that after $\max\min$ objective can no longer be improved (because the lowest utility application cannot be allocated any more resources), the system should continue improving the utility of other applications.

The placement constraints considered by the algorithm can be summarized as:

1. The sum of the memory consumed by instances of all applications on a node may not exceed the memory capacity of this node.
2. For each application and each node, there may exist a limit on the amount of throughput (and hence CPU demand) that may be served by a single instance of the application. A limit of this type exists if an application has some internal bottleneck or scalability problem. For such applications, it may be beneficial to start more than one instance on the same node.
3. To satisfy high-availability requirement, with each application one may associate the minimum and maximum limits on the number of nodes where application is placed. Similarly, one can configure the minimum and maximum number of instances that should be started for an application.
4. A user may flag some application instances as pinned, which means that they cannot be stopped by the controller. Pinning instances is useful when an application includes some singleton components that run in only one instance in the application cluster and may be costly or impossible to migrate to other instances.
5. A placement of an application in manual mode cannot be changed.
6. For each application a user may configure a set of nodes where the application may be execute. The controller may not start an instance of the application on a node that is not included in this list.
7. A user may configure collocation constraints for any pair of applications.

5.3.2 Algorithm outline

The placement algorithm proceeds in three phases: demand capping, placement calculation, and maximizing load distribution. Placement change phase is where actual placement optimization is done, and the most important part of the algorithm in the scope of this thesis. Based on a prior study focusing on the placement problem with a linear optimization objective [108], we identified several heuristics that are applicable also in the placement problem with non-linear optimization objective. Demand capping constraints the amount of CPU capacity that may be allocated to an application, which is used by placement calculation. The phase of maximizing load distribution takes placement obtained by placement calculation phase and calculates the best corresponding load distribution.

5.3.2.1 Placement change method

The placement change phase is executed several times, each time being referred to as a ‘round’. Each round first calculates the load distribution that maximizes the utility of the initial placement. It then invokes the placement change method, which makes a single new placement suggestion based on the provided initial placement. In the first round, the initial placement is the current placement. In subsequent rounds, the initial placement is the placement calculated in the previous round. Additionally, in the first round, the method may be invoked multiple times with various additional instance pinning constraints. For example, in one invocation, all instances are pinned (thus only new instances may be started). In another invocation, all instances that receive load are pinned. We perform up to 10 rounds. We break out of the round loop earlier if no improvement in placement utility is observed at the end of a round.

The placement change method iterates over nodes in a so called *outer* loop. For each node, we invoke an *intermediate* loop, which iterates over all instances placed on this node and attempts to remove them one by one, thus generating a set of configurations whose cardinality is linear in the number of instances placed on the node. For each such configuration, an *inner* loop is invoked, which iterates over all applications whose satisfied demand is less than the limit calculated in the capping phase, attempting to place new instances on the node as permitted by the constraints.

For each application and for each node we calculate *utility-of-stopping* as the application utility that would be obtained if this instance alone was stopped. For each node we obtain its *utility-of-stopping* as the maximum *utility-of-stopping* among all application currently hosted on it. In the *outer* loop we order nodes according to the decreasing utility of stopping. Nodes with sufficient memory to host one more instance of some unsatisfied application have a *utility-of-stopping* equal to 1. Among nodes with equal *utility-of-stopping* we select the one that has the most CPU capacity available, which helps us maximize placement utility without making unnecessary changes. It is also used in a shortcut: node iteration can stop once a node is reached whose *utility-of-stopping* is less than or equal to the lowest utility of an unsatisfied application.

In the *intermediate loop*, we visit instances in decreasing order of *utility-of-stopping*. We break out of the loop when the *utility-of-stopping* becomes lower than or equal to the lowest utility of an unsatisfied application.

In the *inner loop*, we visit application in the increasing order of their utility in the current placement. While the placement change method calculates a load distribution matrix along with the placement matrix, due to the heuristic nature of algorithm, it does not necessarily find the best load distribution for the calculated placement. This step is

solved in the external *maximizing load distribution* method described in 5.3.2.3.

5.3.2.2 Capping application demand

At the beginning of the placement algorithms we cap the demand of each application to a value that corresponds to a maximizing load distribution in a perfect placement, which is a placement that is not constrained by memory, minimum and maximum constraints, and collocation constraints. In other words, we try to calculate an upper bound on the achievable placement utility. In the main method of the algorithm, we observe this capping while deciding which applications are unsatisfied and how much CPU capacity should be allocated to an application instance. This aids the heuristic of the inner loop, which otherwise allocates the maximum available CPU power to an application instance it creates. Since the algorithm is driven by non-decreasing utility function over a continuous range, without some upper bound, the inner loop would always allocate the entire available CPU power of a box to a single application without giving other applications a chance to use the node. This would result in coarse and unfair allocation of resources, and possibly in starving some applications. When capping is given, we can constrain the CPU allocation to any application to what we believe it could obtain in an optimal, unconstrained placement. Naturally, it is possible that no other application will be able to use the node, which seems to result in wasted node capacity. However, this under-allocation will be fixed by the maximizing load distribution phase, which will give the unallocated capacity of a node back to the previously capped application.

To calculate the capping limits, we solve the maximizing load distribution problem by providing a *complete* placement as input, where complete placement includes the maximum number of instances of each application on every node as long as allocation restrictions are not violated.

5.3.2.3 Maximizing load distribution

To find a maximizing load distribution we need to solve a non-linear optimization problem $\max_L U(L)$ subject to linear constraints, which were outlined before. We use standard approximation techniques (see [33, 2]) to solve this optimization problem achieving an approximation that is within a configurable ΔU of the optimum.

5.4 Characterization of heterogeneous workloads

In this section we describe the workload characterization that is required by the proposed placement algorithm to produce optimized placements. We use a different workload characterization technique for transactional and long running workloads. Transactional workloads usually show short duration of individual requests and their actual performance can be continuously compared to their performance goals. Long running workloads tipically require calculation of a long-term schedule to evaluate their performance. In addition, transactional workloads usually present similarity properties that allows performance forecasting, while these properties have not been observed for long running workloads.

First, we describe how we calculate the utility of transactional applications. This is mostly done by the flow controller provided by the application server middleware and previously described in section 5.2. Later describe how we characterize the performance of long running applications. We need a mechanism to calculate their actual utility, but even most important, we need a performance model that allows reliable performance prediction. The proposed technique does not need to calculate an optimal schedule of the jobs. Such a study, considering a system that is also subject to varying transactional workload, would be unfeasible. To our knowledge, this is the first proposal that uses utility functions to make the performance of long runnnig workloads directly comparable to the performance of transactinal workloads, overcoming their intrisic differences.

5.4.1 Transactional workloads

5.4.1.1 Calculating application utility

We assume that in the system described in section 5.1, a user can associate a response time goal, τ_f and an importance level, i_f with each application. The importance level is an integer value which is greater than or equal to 1 and controls the system behavior when the response time goal cannot be met.

Based on the observed response time for an application, t_f we evaluate the system performance with respect to the application satisfaction using utility function u_f, which is defined as follows.

$$u_f(t_f) = \begin{cases} \frac{\tau_f - t_f}{\tau_f} & \text{if } t_f \leq \tau_f \\ \frac{\tau_f - t_f}{i_f \tau_f} & \text{otherwise} \end{cases} \tag{5.4}$$

The importance level decides the slope of the utility function degradation when the

response time exceeds its goal. For less important applications (those with a lower importance level value) the utility function degrades slower with the increasing distance between the response time and its goal.

For the purpose of resource allocation we need to formulate the utility function as a function of allocated CPU capacity, ω_f, that is $u_f(\omega_f) = u_f(t_f(\omega_f))$. Hence, we need to be able to express response time as a function of allocated CPU capacity. The flow controller [79] component of the application server middleware (see figure 5.1 in section 5.2 for more details) performs this calculation. Although this process is out of the scope of this work, some details about this process are provided below for the reader's interest.

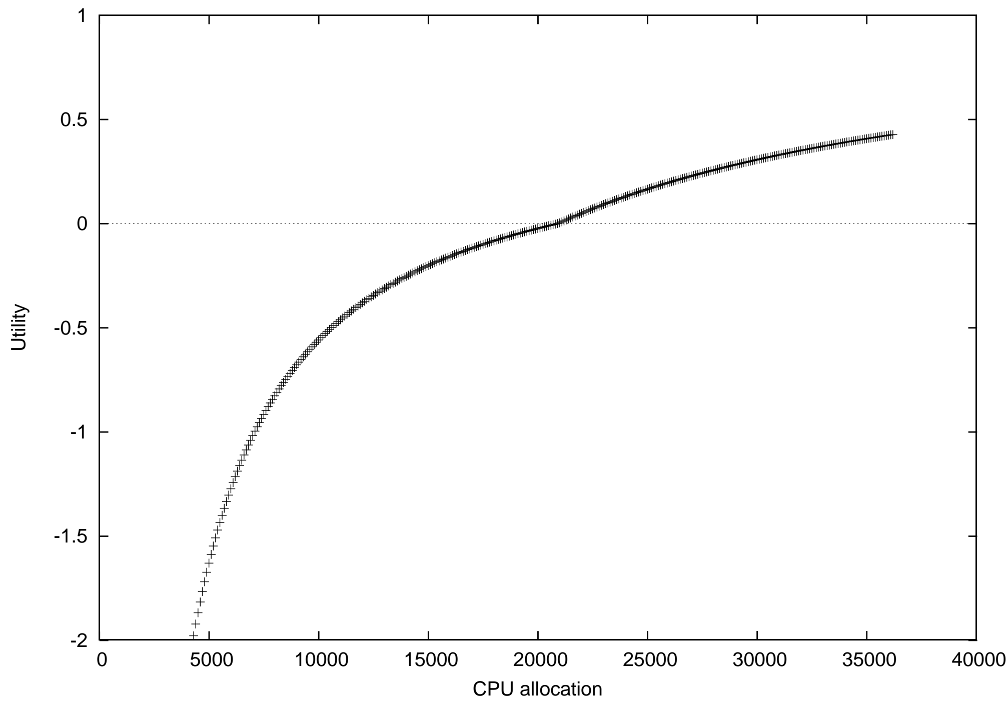


Figure 5.2 Real utility function that corresponds to a transactionl application

It is mathematically easier to express the opposite relationship: the amount of CPU power needed to achieve a particular utility. For a value of the utility function u_f^*, its corresponding response time is expressed as $t_f(u_f^*)$, which is defined by inversing Eq. 5.4. Given client population C_f and client think time T_f, we can obtain the throughput corresponding to u_f^* as follows.

$$\lambda_f(u_f^*) = \frac{C_f}{t_f(u_f^*) + T_f} \tag{5.5}$$

In a clustered environment such as proposed in section 5.2, it is supposed that requests

can reach the system through different request routers. Requests targetting the same application, that belong to the same traffic class and that arrive through the same request router are considered to be a request flow. Given the average number of CPU cycles consumed by each request of a particular flow, α_f, the amount of CPU power needed to achieve u_f^* can be found by multiplying α_f and $\lambda_f(u_f^*)$.

$$\omega_f(u_f^*) = \alpha_f \lambda_f(u_f^*) = \frac{\alpha_f C_f}{t_f(u_f^*) + T_f} = \begin{cases} \frac{\alpha_f C_f}{\tau_f(1-u_f^*)+T_f} & \text{if } u_f^* > 0 \\ \frac{\alpha_f C_f}{\tau_f(1-i_f u_f^*)+T_f} & \text{otherwise} \end{cases} \tag{5.6}$$

Assuming that the flow controller equalizes the utility among flows belonging to the same application, as it is the case with the controller described in [79] and assumed in this work, the CPU requirement of an application can be now expressed as a function of the utility for this application, u_m^*, as a sum over all application flows.

$$\omega_m(u_m^*) = \sum_f \omega_f(u_m^*) \tag{5.7}$$

To obtain $u_m(\omega_m)$, $\omega_m(u_m^*)$ can be sampled for various values of u_m^* and from the obtained datapoints, $u_m(\omega_m)$ can be extrapolated. Figure 5.2 shows an example of a utility function that corresponds to a transactionl application as it is generated by the flow controller discussed in [79], and provided to the placement algorithm.

5.4.2 Long running workloads

In this section, we focus on applying our placement technique to manage long-running jobs. We start by making an observation that long-running jobs cannot be treated as individual management entities for the purpose of performance management as their completion times are not independent. For example, if jobs that are currently running complete faster, this permits jobs currently in the queue (not running) to complete faster as well. Thus, performance predictions for long-running jobs must be done in relation other other long-running jobs.

Another challenge is to provide performance predictions with respect to job completion time on a control cycle which may be much lower than job execution time. Typically, such a prediction would require us to calculate an optimal schedule for the jobs. To trade off resources among transactional and long-running workloads we would have to evaluate a number of such schedules calculated over a number of possible divisions of resources among the two kinds of workloads. The number of combinations would be exponential in the number of machines in the cluster.

We avoid this complexity by proposing an approximate technique, which is presented in this section.

5.4.2.1 Job characteristics

With each job m that is submitted to run, we associate the following information.

Resource usage profile. A resource usage profile describes resource requirements of a job and is given at job submission time. In an actual system, the profile is estimated based on historical data analysis. Each job m is a sequence of N_m stages, $s_1, \ldots, s_{N_m}$, where each stage s_k is described by the following parameters.

- The amount of CPU cycles consumed in this stage, $\alpha_{k,m}$
- The maximum speed with which the stage may run, $\omega_{k,m}^{\max}$. A CPU allocation higher by $\omega_{k,m}^{\max}$ would not be consumed in this stage.
- The minimum speed with which the stage must run, whenever it runs, $\omega_{k,m}^{\min}$. An allocation lower than $\omega_{k,m}^{\min}$ would not permit a correct execution of the stage.
- Memory requirement $\gamma_{k,m}$

Performance objectives. An SLA objective for a job is expressed in terms of its desired completion time, τ_m, which is clock time when the job must have completed. Clearly, τ_m should be greater than job desired start time, τ_m^{start}, which itself is greater than or equal to the time when the job was submitted. A difference between the completion time goal and the desired start time, $\tau_m - \tau_m^{\text{start}}$ is called a relative goal.

We are also given a utility function that maps actual job completion time t_m to a measure of satisfaction from achieving it, $u_m(t_m)$. Many utility function forms may be used. In our implementation, we use the following form.

$$u_m(t_m) = \frac{\tau_m - t_m}{\tau_m - \tau_m^{\text{start}}} \tag{5.8}$$

Runtime state. At runtime, for each job we monitor and estimate the following properties.

- Current status, which may be either running, not-started, suspended, or paused.
- CPU time consumed thus far, α_m^*

5.4.2.2 Stage aggregation in a control cycle

We now focus on reasoning that the APC must apply in order to decide which jobs should be scheduled for execution, on which physical resources they should execute, and how much CPU power they should be allocated. We presume that the APC operates with a control cycle of duration T. Thus, when making decisions at time t_{now}, the APC must consider job progress between time t_{now} and time $t_{\text{now}} + T$.

Depending on stage duration and the value of T, one or more stages can be executed in a control cycle. Since resource allocation will not change for the duration of the control cycle, the resource allocation must be such so as to accommodate all stages that will execute in this cycle.

Considering this, in addition to the job characteristics introduced in section 5.4.2.1, we must now define some additional parameters.

First, we define the cumulative work that a job must complete, $\alpha^c_{D,m} = \sum_{i=1}^{D,m} \alpha_{i,m}$. Since α^*_m cycles have already been completed, the remaining work to complete D stages is $\alpha^{cr}_{D,m} = \max(0, \alpha^c_{D,m} - \alpha^*_m)$. The remaining work to complete the entire job is simply $\alpha^{cr}_{N_m,m}$. At t_{now}, the job must have already completed D^{done}_m stages which may be obtained by taking $D^{\text{done}}_m = \max_D \alpha^c_{D,m} \leq \alpha^*_m$. For each job stage, we can now obtain the work remaining in this stage, which is given as follows:

$$\alpha^r_{D,m} = \begin{cases} 0 & \text{if } D \leq D^{\text{done}}_m \\ \alpha^c_{D,m} - \alpha^*_m & \text{if } D = D^{\text{done}}_m + 1 \\ \alpha_{D,m} & \text{otherwise} \end{cases} \tag{5.9}$$

Let us assume that in each state a job is allocated the maximum usable speed. Then, the time remaining to complete stage D is $t^r_{D,m} = \frac{\alpha^r_{D,m}}{\omega^{max}_{D,m}}$. In time T, the job cannot progress to complete more than D^{last}_m stages where D^{last}_m is the maximum D such that $\sum_{i=D^{\text{done}}_m+1}^{D} t^r_{i,m} \leq T$.

We can now make a conservative assumption, that throughout the cycle that starts at t_{now} and lasts for time T, the job will require the minimum CPU speed, maximum CPU speed, and memory of $\omega^{\min}_m$, ω^{req}_m, and γ_m, respectively, which are defined as follows.

$$\omega^{\min}_m = \max_{D^{\text{done}}_m+1 \leq i \leq D^{\text{last}}_m} \omega^{\min}_i \tag{5.10}$$

$$\omega^{\text{req}}_m = \max_{D^{\text{done}}_m+1 \leq i \leq D^{\text{last}}_m} \omega^{\max}_i \tag{5.11}$$

$$\gamma_m = \max_{D^{\text{done}}_m+1 \leq i \leq D^{\text{last}}_m} \gamma_i \tag{5.12}$$

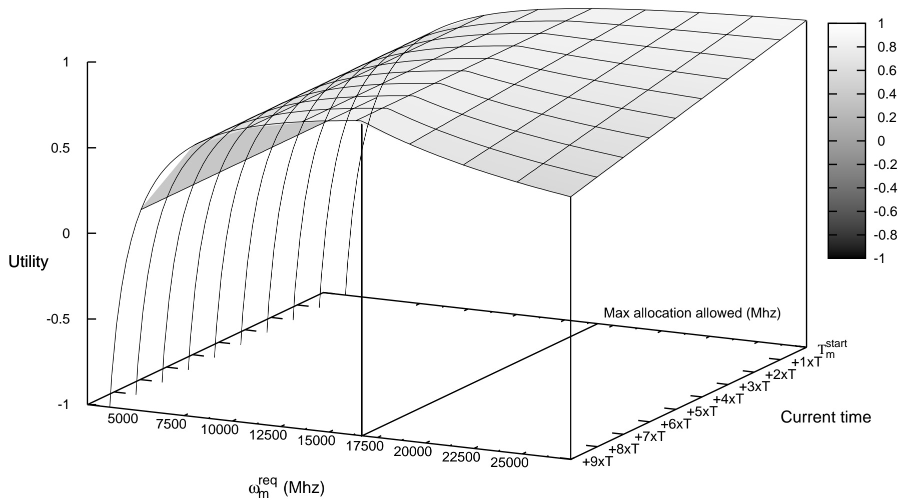


Figure 5.3 Evolution of maximum achievable utility for long running jobs

5.4.2.3 Maximum achievable utility

There is an inherent upper bound to the utility that a job may achieve considering its progress thus far. When progressing with the maximum possible speed, a job will complete in time $t_m^{\text{best}} = \sum_{i=D_m^{done}}^{D} t_{i,m}^r$ thus its completion time will be $t_{\text{now}} + t_m^{\text{best}}$. The maximum utility that the job can achieve given its progress thus far is $u_m^{max} = u_m(t_{\text{now}} + t_m^{\text{best}})$.

At any time, a job may be either running with the maximum speed, running with less than the maximum speed, or stopped or suspended (not running).

In general, in each control cycle in which the CPU allocation of a job is less than the maximum it can use, the value of maximum achievable utility decreases linearly. We illustrate this property with the following example.

Consider a single stage job and suppose that its amount of work and an SLA goal is such that the job can achieve the utility of 0.8 when executing with speed 16,000 MHz. In Figure 5.3, we show the evolution of the maximum achievable utility over 10 control cycles. We presume that the system cannot allocate more than 16,000 MHz to the job and vary the job maximum speed, ω_m^{req} (plotted on the X axis). When $\omega_m^{req} < 16000$, u_m^{max} is lower than 0.8, and it decreases as ω_m^{req} decreases, but it stays constant over time (plotted on the Y axis), as the system is always able to allocate ω_m^{req}. When $\omega_m^{req} > 16000$, then u_m^{max} may be higher than 0.8, as the job may complete its work faster. However, since the system is not able to provide this much CPU power, and the job only executes with the speed of 16,000 MHz, u_m^{max} decreases over time.

The maximum achievable utility is used to order long-running jobs in the queue. When allocating resources to long-running workload, the APC will first consider jobs with a

lower maximum achievable utility.

5.4.2.4 Hypothetical utility

To calculate job placement, we need to define a utility function which the APC can use to evaluate its placement decisions. While the actual utility achieved by a job can only be calculated at completion time, the algorithm needs a mechanism to predict (at each control cycle) the utility that each job in the system will achieve given a particular allocation. And this is still true even for jobs that are not yet started, for which the expected completion time is still undefined. To help answer questions that APC is asking of the utility function for each application we introduce the concept of *hypothetical utility*.

Estimating application utility given aggregate CPU allocation Suppose that we deal with a system in which all jobs can be placed simultaneously and in which the available CPU power may be arbitrarily finely allocated among the jobs. We require a function that maps the system's CPU power to the utility function achievable by jobs when placed on it.

Let us consider job m. Based on its properties, we can estimate the completion time needed to achieve utility u, $t_m(u) = \tau_m - u(\tau_m - \tau_m^{\text{start}})$. From this number, we can calculate the average speed with which the job must proceed over its remaining lifetime to achieve u, which is given as follows.

$$\omega_m(u) = \frac{\alpha^{cr}_{N_m,m}}{t_m(u) - t_{\text{now}}} \tag{5.13}$$

To achieve the utility of u for all jobs, the aggregate allocation to long-running workload must be $\omega_g = \sum_m \omega_m(u)$. To create the utility function, we sample $\omega_m(u)$ for various values of u and interpolate values that fall between the sampling points.

Let $u_1 = -\infty, u_2, \ldots, u_R = 1$, where R is a small constant, be a set of sampling points (target utilities from now on). We define matrices W and V as follows:

$$W_{i,m} = \begin{cases} \omega_m(u_i) & \text{if } u_i < u_m^{\max} \\ \omega_m(u_m^{\max}) & \text{otherwise} \end{cases} \tag{5.14}$$

$$V_{i,m} = \begin{cases} u_i & \text{if } u_i < u_m^{\max} \\ u_m^{\max} & \text{otherwise} \end{cases} \tag{5.15}$$

Cells $W_{i,m}$ and $V_{i,m}$ contain the average speed with which application m should execute starting from t_{now} to achieve utility u_i and value u_i, respectively, if it is possible for application m to achieve utility u_i. If utility u_i is not achievable by applicaion m, these

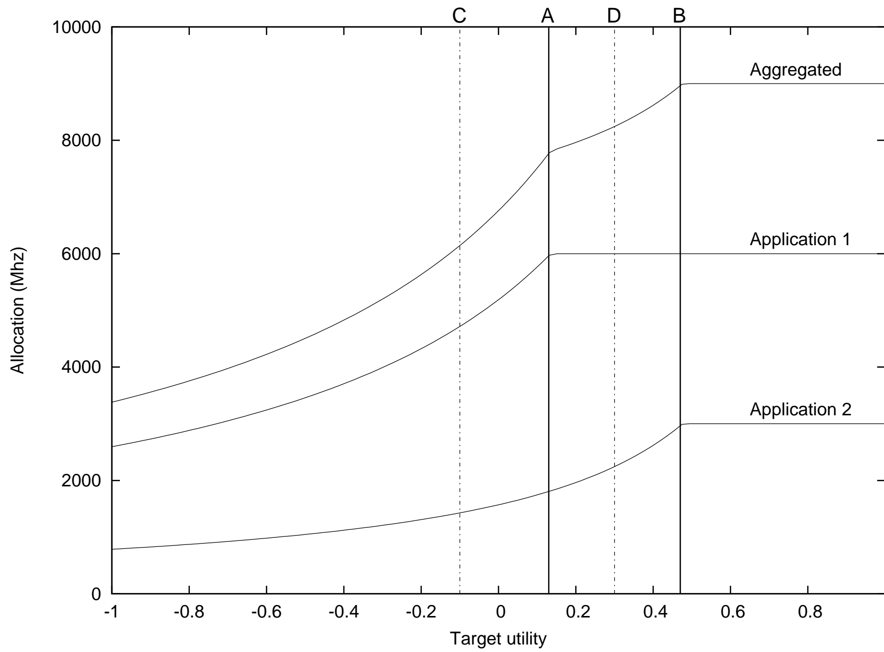


Figure 5.4 Allocation as a function of target utilities as used to estimate application utility given aggregate CPU allocation (used to fill vectors V and W). Notice that the maximum achievable utility for some jobs may be lower than the target utility (i.e. target utilities beyond point A for application 1, and beyond point B for application 2). Cells of vectors W and V for application 1 that correspond to target utilities beyond point A are filled with values 6000 and 0.14 (maximum achievable utility of 0.14 for allocation 6000Mhz). In the case of application 2 for target utilities beyond point B, vector cells are filled with values 3000 and 0.44

cells instead contain the average speed with which the application should execute starting from t_{now} to achieve its maximum achievable utility, and the value of the maximum utility, respectively.

For a given ω_g, there exist two values k and $k+1$ such that:

$$\sum_m W_{k,m} \leq \omega_g \leq \sum_m W_{k+1,m} \tag{5.16}$$

Allocating a CPU power of ω_g to the group will result in having a utility u_m for each application m in the group in the range:

$$V_{k,m} \leq u_m \leq V_{k+1,m} \tag{5.17}$$

That corresponds to a hypothetical CPU allocation in the range:

$$W_{k,m} \leq \omega_m \leq W_{k+1,m} \tag{5.18}$$

Figure 5.4 shows, for two different applications, the allocation required to achieve a range of target utilities, as well as the aggregated demand for the group. Vectors V and W can be constructed by sampling some of the points shown in this figure. Note that, for utilities above the maximum achievable utility for a particular application (points A and B), we take the allocation that corresponds to that maximum achievable utility.

Interpolating u_m and ω_m given ω_g. At some point the algorithm needs to know the utility that each application will achieve (u_m) if it decides to allocate a CPU power of ω_g to the group. We must find values ω_m and u_m for each application m such that equations 5.16, 5.17, and 5.18 are satisfied, while also satisfying $\sum_m \omega_m = \omega_g$. As finding a solution for this final requirement implies solving a system of linear equations, which is too costly to perform in an online placement algorithm, we use an approximation based on the interpolation of value ω_m from cells $W_{k,m}$ and $W_{k+1,m}$, where k and $k+1$ follow equation 5.16, and deriving u_m from ω_m. Notice from equations 5.14 and 5.15 that the value of a cell $V_{i,m}$ does not necessarily corresponds to the target utility u_i, and thus a cell $W_{i,m}$ does not necessarily correspond to $\omega_m(u_i)$.

To interpolate ω_m, we first consider the case for which cells $W_{k,m}$ and $W_{k+1,m}$ correspond to the allocations required to make application m achieve utilities u_k and u_{k+1} correspondingly, i.e., the case for which the calculation of those cells was not constrained by the maximum achievable speed for application m. In this situation, ω_m can be interpolated by calculating first a value $ratio_g$ that corresponds to the position of ω_g relative to the distance between $\sum_m W_{k,m}$ and $\sum_m W_{k+1,m}$. We define $ratio_g$ as:

$$ratio_g = \frac{\omega_g - \sum_m W_{k,m}}{\sum_m W_{k+1,m} - \sum_m W_{k,m}} \tag{5.19}$$

Once $ratio_g$ is calculated, we can interpolate ω_m as $(W_{k+1,m} - W_{k,m}) * ratio_g + W_{k,m}$.

Figure 5.5 shows an example of this interpolation. We consider an scenario similar to that shown in Figure 5.4 in the region indicated by point C. For simplicity, we consider vectors V and W to be filled with values obtained from functions x^3 for job 1, x^2 for job 2 and $x^3 + x^2$ for the aggregated values. We are given a group allocation $\omega_g = 150$ and utility sampling points 2 and 8; and we need to interpolate values $\tilde{\omega}_{job1}$ and $\tilde{\omega}_{job2}$ that satisfy equation 5.18 and that are as similar as possible. We have the following available data: $W_{2,job1} = 8$, $W_{2,job2} = 4$, and so $\sum_m W_{2,m} = 12$; and $W_{8,job1} = 512$, $W_{8,job2} = 64$, and so $\sum_m W_{8,m} = 576$. We calculate $ratio_g$ to be $(150 - 12)/(576 - 12) = 0.24$. With this value we esttimate $\tilde{\omega}_{job1} = 131$ and $\tilde{\omega}_{job2} = 18$. The corresponding utilities for these job allocations are 4.2 for job 1 and 5.08 for job 2. Obviously, solving the linear system of equations would have produced a more precise solution at a much higher computational

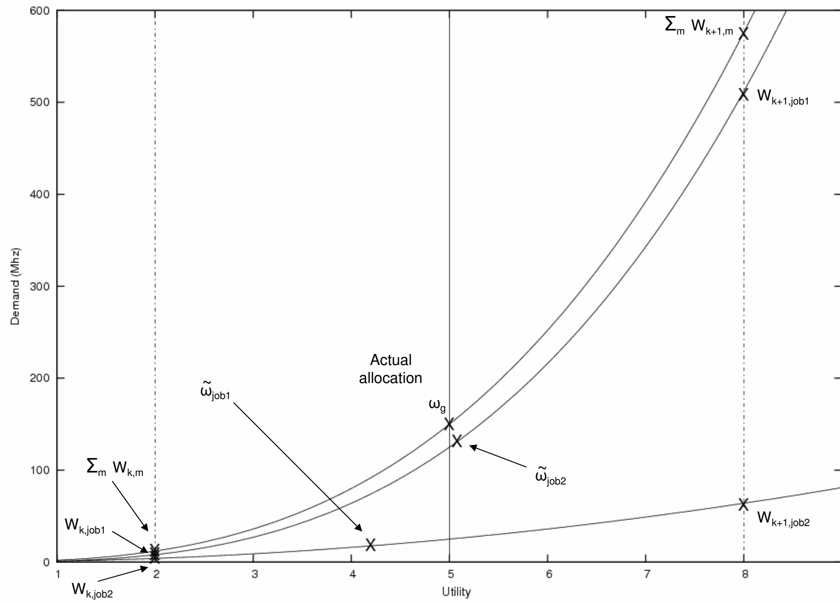


Figure 5.5 Estimating $\tilde{\omega}_m$ using $ratio_g$

cost.

This technique works well when all applications are unconstrained (they have not reached their maximum speed). Notice that the region around point D in Figure 5.4 shows a different scenario, in which application 1 is running at maximum speed. In this situation, interpolating $\tilde{\omega}_{job1}$ and $\tilde{\omega}_{job2}$ requires some additional effort. Figure 5.6 shows an scenario similar to that indicated by point D in Figure 5.4. We proceed analogously as we did before, with the only difference that 1) we consider the function for application 1 to be $min(1000, x^3)$ and obviously the aggregated function becomes $min(1000, x^3)+x^2$; and 2) we are given a group allocation $\omega_g = 1180$ and utility sampling points 5 and 30;. We have the following available data: $W_{5,job1} = 25$, $W_{5,job2} = 125$, and so $\sum_m W_{5,m} = 150$; and $W_{30,job1} = 1000$ (constrained), $W_{30,job2} = 900$, and so $\sum_m W_{30,m} = 1900$. We calculate $ratio_g$ to be $(1180 - 150)/(1900 - 150) = 0.58$. With this value we estimate $\tilde{\omega}_{job1} = 640$ and $\tilde{\omega}_{job2} = 540$. This time the corresponding utilities for these job allocations are 8.6 for job 1 and 23.0 for job 2. As it can be observed, $\tilde{\omega}_{job1}$ and $\tilde{\omega}_{job2}$ satisfy equation 5.18, but they are far from each other. This is caused by the fact that application 1 is contributing differently to the aggregate function in the range of utilities 5 - 30, and under these circumstances $ratio_g$ is not accurate enough for our purpose of equalizing utilities if possible.

To overcome this problem we define app_ratio_m as:

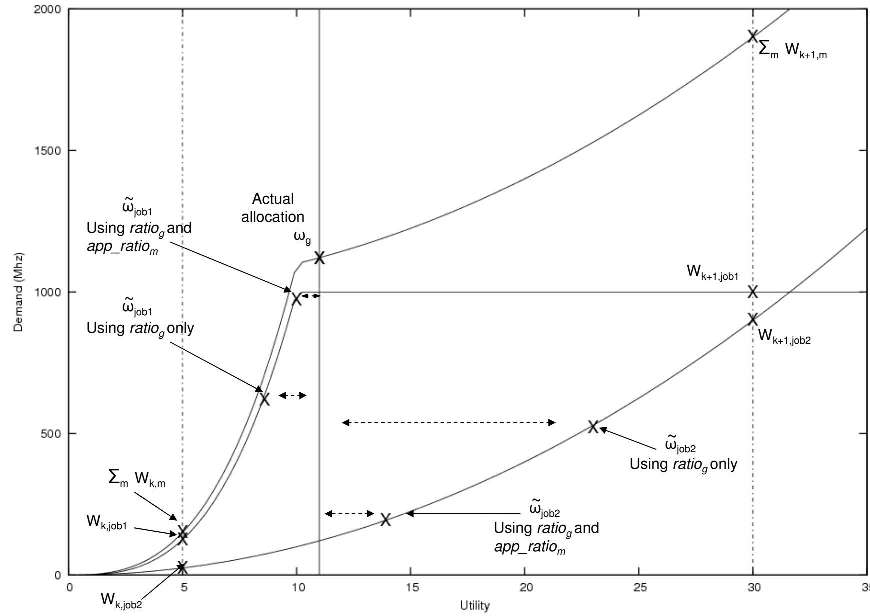


Figure 5.6 Estimating $\tilde{\omega}_m$ using $ratio_g$ and app_ratio_m

$$app_ratio_m = \frac{\frac{\sum_m W_{k+1,m}}{\sum_m W_{k,m}}}{\frac{W_{k+1,m}}{W_{k,m}}} \tag{5.20}$$

Notice that app_ratio_m observes how the difference between cells $W_{k,m}$ and $W_{k+1,m}$ is related to the overall system allocation that corresponds to $\sum_m W_{k,m}$ and $\sum_m W_{k,m}$. In the case that both $W_{k,m}$ and $W_{k+1,m}$ are constrained by the maximum achievable speed for application m, then we have that $app_ratio_m = 0$.

We now define:

$$ratio_m = \begin{cases} ratio_g & W_{i,m} = u_{i,m}, i = k, k+1 \\ ratio_g * app_ratio_m & \text{otherwise} \end{cases} \tag{5.21}$$

and interpolate the allocation $\tilde{\omega}_m$ that corresponds to application m given ω_g as $(W_{k+1,m} - W_{k,m}) * app_ratio_m + W_{k,m}$.

Looking again to the example proposed in Figure 5.6, we get $app_ratio_{job1} = (1900/150)/(1000 - 125) = 1.65$ and $app_ratio_{job2} = (1900/150)/(900 - 25) = 0.33$. With these values, as well as the previously calculated $ratio_g$, we can calculate values $\tilde{\omega}_{job1} = 974$ and $\tilde{\omega}_{job2} = 195$, that still satisfy equation 5.18 but are closer in terms of utility. This technique has proven to work as expected for our input functions.

Once $\tilde{\omega}_m$ is correctly estimated, we can easily estimate the expected utility of

application m that belongs to group g as:

$$\tilde{u}_m(\omega_g) = \begin{cases} -\infty & \text{if } \omega_g = 0 \\ u_m(t_{\text{now}} + \frac{\alpha^{cr}_{N_m,m}}{\tilde{\omega}_m(\omega_g)}) & \text{otherwise} \end{cases} \tag{5.22}$$

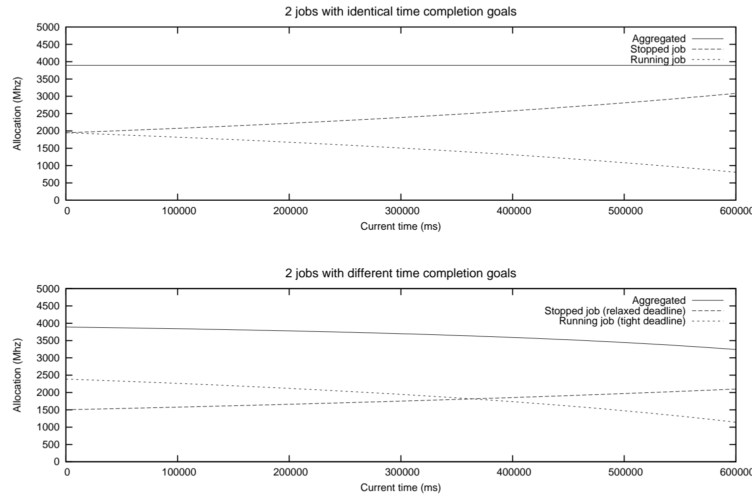


Figure 5.7 Hypothetical utility: effect of resource competition on the required allocation to obtain utility -0.1. Only one job can be placed (running) while the other is stopped. When both jobs have identical characteristics and same deadline (upper chart), the hypothetical utility for both jobs can be equalized over time by hypothetically allocating more CPU power to the job that is stopped to compensate the time it is stopped, resulting in a constant aggregate allocation. When both jobs are different (lower chart), their hypothetical utility still can be equalized but the aggregate allocation is not constant

Evolution of hypothetical utility over time. The hypothetical utility function assumes all jobs can be placed at once, which is usually not the case. This means that real placements differ from what is assumed by the hypothetical utility. In this case, when after time T the hypothetical utility is calculated again it may have a different form than the utility calculated at time t_{now}. Figure 5.7 illustrates two scenarios in which two applications compete for resources (only one of the applications can be placed at a time). In the upper chart, both applications have the same characteristics (maximum speed, importance level, submission time and completion time goal). In the upper chart, each application has different characteristics. The charts show how much CPU must be allocated to each application according to the calculated hypothetical utility as well as the aggregated allocation ($\omega_{m_1} + \omega_{m_2}$) necessary to achieve the utility of -0.1. Notice that when both applications have identical characteristics, the evolution of the required allocation for both applications to get the same hypothetical utility is complementary: the application that is placed and running presents a decreasing demand to get the

same utility while the application that is stopped presents an increasing demand. The aggregated allocation keeps constant. In the second chart, where applications present different characteristics, the evolution of the allocation required by each applications differ. When the application placed has a tighter completion time goal, its requested allocation decreases more quickly than the demand for the other application increases. In this case, the aggregated required allocation for both applications to obtain identical hypothetical utility decreases slightly over time as the more constrained job progresses in execution.

Evaluating placement decisions Let P be a given placement. Let ω_m be the amount of CPU power allocated to application m in placement P. For jobs that are not placed, $\omega_m = 0$.

To calculate utility of application m given placement P that is calculated at time t_{now} for a control cycle that lasts time T, we calculate a hypothetical utility function at time $t_{\text{now}} + T$. For each job, we increase its α^* by the amount of work that will be done over T with allocation ω_m. We use this obtained hypothetical utility to extrapolate u_m from matrices W and V for $\omega_g = \sum_m \omega_m$.

Thus, we use the knowledge of placement in the next cycle to predict job progress over its duration, and use hypothetical function to predict job performance in the following cycles. We also assume that the total allocation to long-running workload in the following cycles will be the same as in the next cycle. This assumption helps us balance long-running work execution over time.

5.5 Prototype implementation

In this section we present the details of the implementation of a prototype system that is able to explicitly manage heterogeneous workloads. The system takes full advantage of virtualization technology (Xen [10] in our case) to enforce resource allocation decisions made by the *Application Placmenet Controller* (APC). The prototype was build by extending the state-of-the-art application server middleware presented in section 5.2. Thus, in following sections we present the details of the integration of virtualization technology into that middleware.

First, we discuss how our system makes use of virtualization technologies to manage heterogeneous workloads. Recall from subsection 5.2 that, to manage web workloads, our system relies on an entry gateway that provides flow control for web requests. The entry gateway provides a type of high-level virtualization for web requests by dividing CPU capacity of managed nodes among competing flows. Together with an overload protection mechanism, the entry gateway facilitates performance isolation for web applications. Such virtualization technology exists in some application server middleware systems, of which WebSphere Extended Deployment [116] is the example most familiar to us.

Server virtualization could also be used to provide performance isolation for web applications. This would come with a memory overhead caused by additional copies of the OS that would have to be present on the node. Hence, we believe that middeware virtualization technology is a better choice for managing the performance of web workloads.

Since middeware virtualization technology can only work for applications whose request-flow it can control, a lower level mechanism must be used to provide performance isolation for other types of applications. As outlined in the previous subsection, server virtualization provides us with powerful mechanisms to control resource allocation of non-web applications.

Several server virtualization technologies have been developed thus far [76] and they all could be used by our system, although possibly with a limited set automation mechanisms. Our implementation makes use of Xen [10], and takes full advantage of the virtualization operations enumerated in section 2.3.2.

5.5.1 VM management

To manage VMs inside a physical Xen-enabled node, we have implemented a component, called the *machine agent* (Figure 5.8), which resides in domain 0 so as to have access to the Xen domain controls. The machine agent provides a Java-based interface

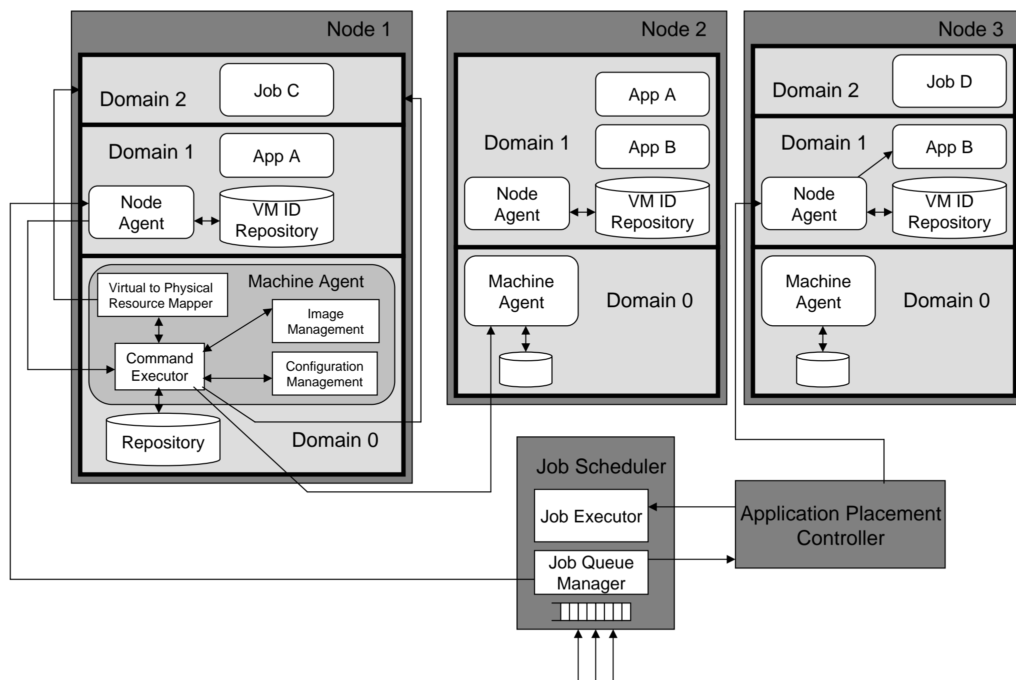


Figure 5.8 Management architecture for Xen machines.

to create and configure a VM image for a new domain, copy files from domain 0 to another domain, start a process in another domain, and to control the mapping of physical resources to virtual resources. During its life-cycle, a domain can transition between various states in accordance with the transitions shown in Figure 5.9.

We use Xen to provide on-line automation for resource management, hence we want to make management actions light-weight and efficient. This consideration concerns the process of creating virtual images, which may be quite time consuming. We avoid substantial delays, which would otherwise be incurred each time we intend to start a job, by pre-creating a set of images for use during runtime. The dispensing of these pre-created images is performed by the *image management* subsystem. Images once used to run a process are scrubbed of that process data and may be reused by future processes. In our small-scale testing thus far, we found it sufficient to pre-create a small number of images, however, we plan to extend the *image management* subsystem to dynamically extend the pool of available images, if needed.

Inside a created image, we can create new processes. This is done by populating the image with the files necessary to run that new process. In our system, we assume that the files required for of all processes that may run on the node are placed in its domain 0 in advance. Hence, we only need to copy them from domain 0 to the created image. Clearly, there are mechanisms that would allow us to transfer files from an external repository to

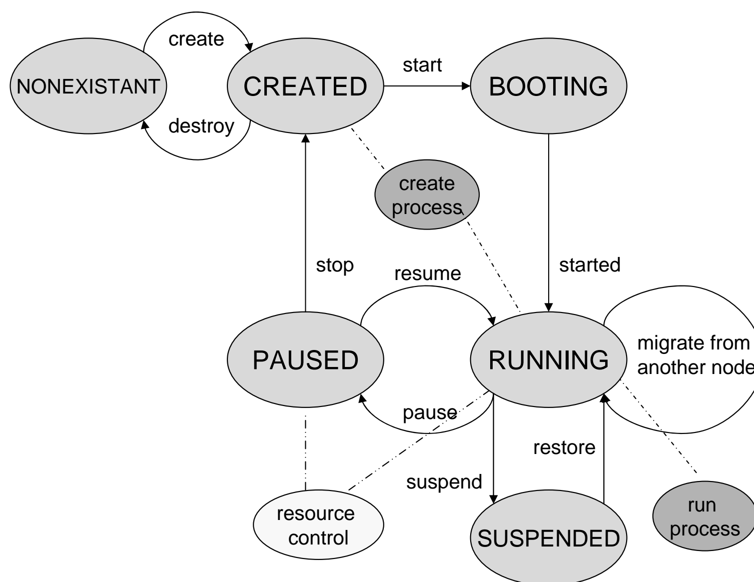


Figure 5.9 Life-cycle of a Xen domain.

a node where the process is intended to run, but we have not used them in our prototype.

Before it may be booted, an image must be provided with configuration files to set up its devices and networking. This functionality is encapsulated by the *configuration management* subsystem. To assign an IP address and DNS name, a DHCP server can be used, although in our system we have implemented a simpler, more restrictive, module that selects configuration settings from a pool of available values.

An image, once configured, may then be booted. Once in the running state, it may be suspended or paused. New processes may be created and run inside it. An image that is either running or paused may also be resource controlled. Migration may be used to transfer the image to another node. Since at the time of writing this dissertation we do not have the shared storage infrastructure required to use migration, we have implemented a suspend-move-and-restore mechanism by which the domain is suspended on one machine, the checkpoint and image files are copied to another node, and the domain is restored on the new host node. This is obviously quite inefficient mechanism, which nevertheless allows us to study the benefits of migration.

Xen provides resource control mechanisms to manage memory and CPU usage by its domains. We set memory for a domain based on configured or profiled memory requirements. We set CPU allocation for a domain based on the decisions of APC, which result from its optimization technique. The CPU allocation to a domain may be lower that the amount of CPU power actually required by a process running inside a domain. Both memory and CPU allocations to a domain may change while the domain is running based

on changing process requirements and decisions of APC.

CPU allocation to domains may be controlled in Xen using three mechanisms. First, the number of virtual CPUs (vCPUs) can be selected for any VM. Second, vCPUs may be mapped to physical CPUs. By 'pinning' vCPUs of a domain to different physical CPUs we can improve the performance of the domain. Finally, CPU time slices may be configured for each domain. When all vCPUs of a domain are mapped to different physical CPUs, allocation of 50 out of 100 time slices to the domain implies that each vCPU of the domain wil receive 50% of the physical CPU to which it is mapped. Xen also permits borrowing, by which CPU slices allocated to a domain that does not need them can instead be used by other domains.

In a default configuration provided by Xen, each domain receives the same number of vCPUs as there are physical CPUs on a machine. Each of those vCPUs will be mapped to a different physical CPU and receives 0 time slices with CPU borrowing turned on. In the process of managing the system, we modify this allocation inside the *virtual-to-physical resource mapper*. When a domain is first started, we allow Xen to create the default number of vCPUs and map them to different physical CPUs. We only set the number of time slices to obtain the CPU allocation requested by placement controller. While domain is running, we observe its actual CPU usage. If it turns out that the domain is not able to utilize all vCPUs it has been given, we can conclude that the job is not multi-threaded. Hence, to receive its allocated CPU share, its vCPUs must be appropriately reduced and remapped. The *virtual-to-physical resource mapper* must attempt to find a mapping that provides the domain with the required amount of CPU power spread across the number of vCPUs that the job in the domain can use—clearly, this is not always possible.

All the VM actions provided by the machine agent are asynchronous JMX calls followed by JMX notifications.

5.5.2 Job management

To hide the usage of VMs from a user, we have implemented a higher-layer of abstraction, embedded inside the *node agent*, which provides the job management functionality. It provides operations to start, pause, resume, suspend, restore, and resource control a job. To implement these operations, the node agent interacts with the machine agent in domain 0 using its VM management interfaces. When a job is first started, the node agent creates (or obtains a pre-created) image in which to run the job. It records the mapping between the job ID and VM ID. Then it asks the machine agent to copy corresponding process binaries to the new image and to boot the image. Once domain is running, the job is started inside it.

Observe that we always place a job in its own domain. This gives us performance isolation among jobs such that we can control their individual resource usage, but it comes at the expense of added memory overhead. We plan to extend our system such that it allows collocation of multiple jobs inside a single domain based on some policies.

The node agent process is placed in domain 1, which is the domain we use for all web applications. There are two reasons for placing the node agent in a separate domain than domain 0. First, our application server middleware already provides a node agent process with all required management support, thus adding new functionality is a matter of a simple plugin. Second, domain 0 is intended to remain small and light-weight. Hence, we avoid using it to run functionality that does not directly invoke VM management tools. Like the machine agent, the node agent exposes its API using JMX.

5.5.3 Xen machine organization

In Figure 5.8 we show the organization of a Xen-enabled server machine we use in our system. We always run at least two domains, domain 0 with the machine agent, and domain 1 with the node agent and all web applications. Since resource control for web applications is provided by request router and flow controller, such collocation of web applications does not affect our ability to provide performance isolation for them. Domains for jobs are created and started on-demand.

5.6 Evaluation in a simulator

In this section we use a simulator for the environment presented in section 5.2 to evaluate the proposed technique in several different scenarios. The behavior of simulator has been validated against the prototype system described in section 5.5. The simulator has been also used to prepare the experiments described in section 5.7, and that were later run in the real system.

We use the simulator to run large scale systems (comprising hundreds of nodes and applications) to demonstrate the effectiveness of our technique as well as evaluate under which circumstances making utility-driven placement decisions makes significant difference compared to other techniques. With our experiments we demonstrate the benefits of using our techniques to make placement decisions under certain circumstances, where state-of-the-art techniques can incur severe unfairness in terms of application performance.

The evaluation is done in three parts. First, a system subject to transactional-only workloads is studied in section 5.6.1. Later, we present some more experiments involving long-running only workloads in section 5.6.2. And finally we present an experiment in which long running jobs and transactional applications are mixed to produce a heterogeneous workload in section 5.6.3.

5.6.1 Transactional-only workloads

In this section, we evaluate the effectiveness of our placement algorithm when subject to a number of different workload conditions and when managing a large number of nodes and applications. For each application we generate a realistic and randomized workload across 300 placement control cycles, which results in a varying CPU demand. The memory demand is uniform across applications for each simulation run.

We focus our study on three different evaluation criteria for the algorithm: evaluation of the algorithm's ability to achieve its objectives (maximization of the minimum utility across applications and minimization of placement changes), evaluation of the sensitivity of the algorithm to the different parameters present in the placement problem, and evaluation of the quality of the placement decisions made by the algorithm. At each step we compare our algorithm with a state-of-the-art dynamic application placement algorithm, described in [58], that differs with respect to our approach in that it tries to maximize the satisfied demand in the system instead of equalizing application satisfaction.

5.6.1.1 Generation of utility functions in the simulator

In the real system implementation, utility functions for transactional applications are generated online by the Flow Controller (as discussed in section 5.2). The shape of the utility functions generated by the Flow Controller is derived from the observed workload conditions as well as from the characterization of the workload properties for each application, such as the CPU allocation required to process particular requests present in the workload. The maximum utility value that can be reach by an application, is calculated from its performance goal and from the observed performance at each control cycle. Notice that the maximum utility value is bounded to 1 (when service time is zero) and is directly constrained by the minimum response time achievable, which in turn is defined by the resource-dependent fraction of the response time. In actuality, the service time is never equal to zero, and the maximum utility always less than 1. Figure 5.10(a) shows an example of two utility functions generated by the Flow Controller, both with a maximum utility of 0.52. One reaches its maximum utility value when 4800Mhz are allocated for it and the other requires 36200Mhz to reach the maximum utility point.

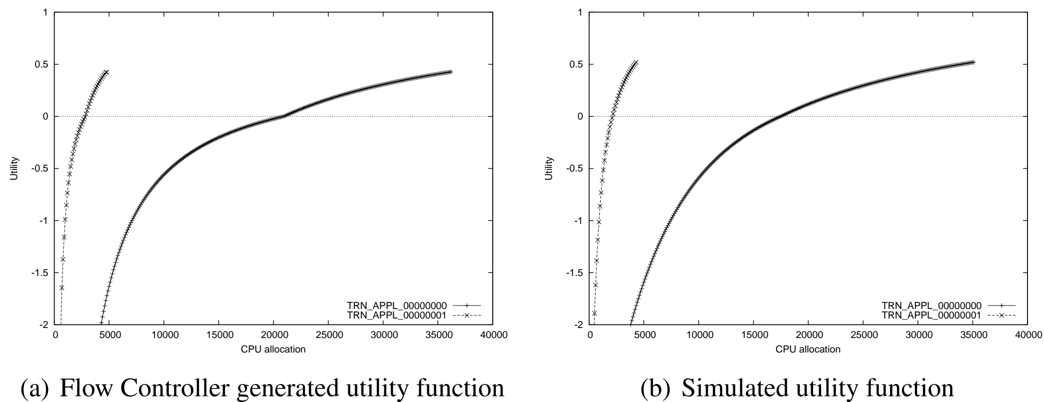
To provide utility functions for the simulation, we have implemented a utility function generator that produces a realistic curve whose shape is controlled by maximum utility value and CPU allocation required to reach this maximum point. Figure 5.10(b) shows a utility function as generated by the simulation environment. Notice that it is directly comparable to the utility functions produced by the Flow Controller (shown in figure 5.10(a)) in the real system. In our simulations, a new maximum CPU allocation is generated for each application at each control cycle and the corresponding utility function updated. These utility functions are directly fed into the utility-based algorithm.

To provide application demand values for the demand-based algorithm, we need to emulate the behavior of Flow Controller, which in the demand-based system, calculates the optimal load distribution across applications based on utility functions. This problem is analogous to the problem of capping application demand described in section 5.3 and we use the same technique to solve it. We use the same utility functions as in the utility-based technique.

5.6.1.2 Evaluation criterion: minimum utility

First, we evaluate the capability of the algorithm to maximize the minimum utility across applications. We simulate a slighlty overloaded system, composed of 100 nodes and 20 applications. A system is overloaded if the total amount of demand needed to maximize the utility of all applications is greater than the total CPU capacity of the system. Each application, on average, requires an allocation equivalent to 5.5 nodes to

be fully satisfied. Given this scenario, we run three simulations, each producing the same per-application CPU demand, but using different application memory demands each time. In the first simulation, we use applications that require little memory, resulting in a configuration where up to 6 application instances can be placed on the same node. For the second simulation, we use medium applications, resulting in a configuration where 2 instances at maximum can be placed on the same node. Finally, in our third simulation we simulate applications large enough to ensure that only one instance can be placed on each node. Increasing the memory demand of the application instances also increases the hardness of the problem. The summary of the results obtained in this experiment is shown in Figure 5.11, where our algorithm is referred to as 'Utility-based' and the algorithm described in [58] is referred to as 'Demand-based'. The utility values shown in the figure correspond to the lowest utility observed across applications at each control cycle.



(a) Flow Controller generated utility function (b) Simulated utility function

Figure 5.10 Example of utility functions

This experiment shows that our utility-driven algorithm consistently achieves an overall minimum utility higher than the value obtained by the demand-based algorithm. In particular, the harder the problem input becomes, the bigger the difference between the two algorithms is. The results obtained for this experiment also indicate that our algorithm has low sensitivity to the problem hardness, shown by the fact that the minimum utility achieved across applications is very similar given three different memory fragmentation scenarios. Notice here that the minimum utility across applications is a direct representation of the QoS obtained by the applications, and that maximizing the minimum utility is a guarantee of fairness in terms of application satisfaction. These results confirm that our utility-driven algorithm is achieving its primary objective of maximizing the minimum utility across applications in a wide range of workload and system configurations.

5.6.1.3 Evaluation criterion: number of placement changes

In our second experiment we evaluate the capability of our algorithm to minimize the number of placement changes over time. A placement change incurs a significant cost in terms of resources and time and it is thus desirable that placement algorithm keep the number of placement changes to a minimum. Such changes should occur only when the benefit that they introduce in terms of utility improvement and fairness is significant. For this experiment, we compare our utility-driven algorithm with the demand-based placement algorithm described in [58], when subjected to a particularly hard scenario. We simulate a system composed of 100 nodes and 200 applications. Each application instance requires half of the memory capacity of a node to be placed, so we can place a only 200 instances. With respect to the CPU demands, we consider three different scenarios: first, when no overload is present in the system; second, when overload is only present in some stages of the test; and third, when the system is always overloaded.

Figure 5.13 shows that the minimum utility achieved by the utility-driven algorithm is slightly worse than the obtained by the demand-driven algorithm when the system is not overloaded or only partially overloaded, but clearly better when the system is completely overloaded. This is because our algorithm is driven by utilities, while the demand-based algorithm tries to maximize the satisfied demand for all applications. This tight scenario forces the demand-based algorithm to make many placement changes because the severe memory constraints make the problem challenging. Our algorithm, instead, decides that because the utility improvement from making any changes is so low, no changes should be made after the initial placement. Notice that, at some points, the demand-based algorithm makes up to 400 placement changes, which means that it is effectively stopping all instances and starting them in different places, chasing a better one-to-one combination of applications sharing nodes that helps it to improve the overall satisfied demand. In addition, the average utility charts demonstrate that although the minimum utility achieved by our algorithm is lower than the result obtained for the demand-based algorithm, the average utility value achieved across applications is very close for the two algorithms. These results confirm that our utility-driven algorithm is achieving its second objective of minimizing placement changes even in hard placement problems.

5.6.1.4 Evaluation criterion: optimality

Ideally, we would like to compare our technique to an optimal, even if very complex, algorithm. Unfortunately, implementing such an algorithm is extremely difficult, and the execution is extremely slow, preventing us from running any useful experiments. Therefore, we implemented a heuristic algorithm which ignores all but CPU and memory

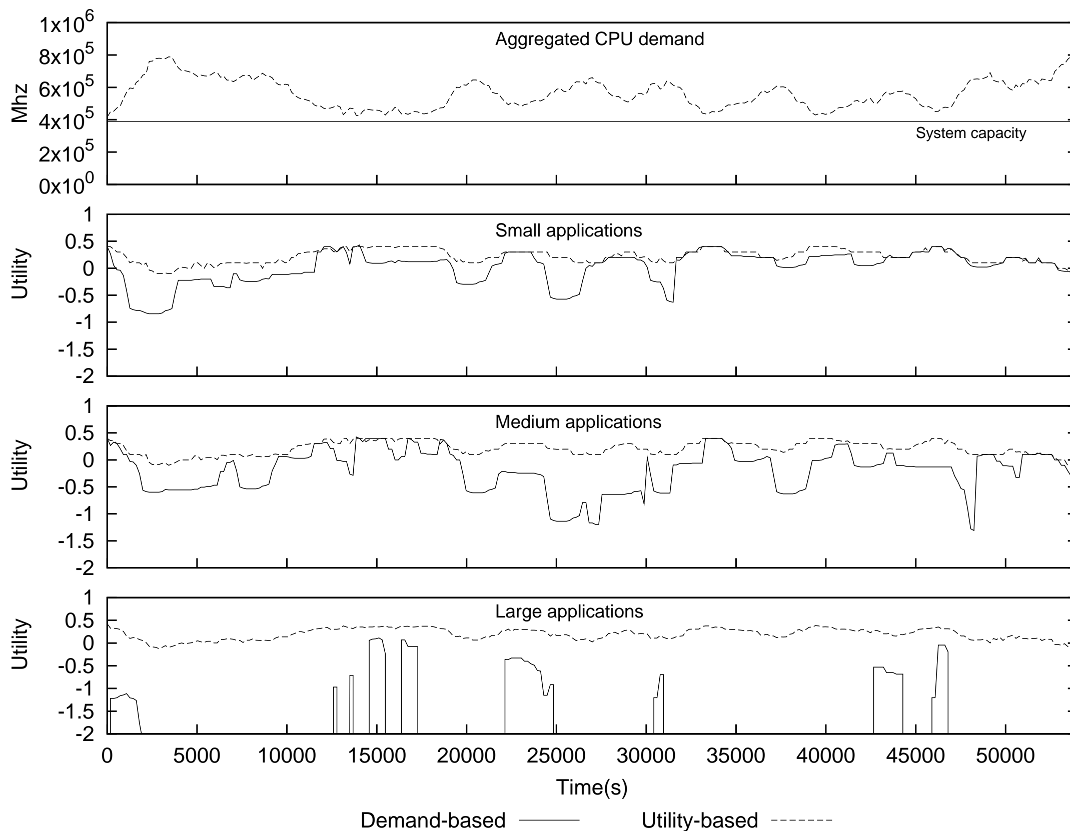


Figure 5.11 Maximizing minimum utility across applications

capacity constraints, and does not aim to minimize the number of placement changes. Consequently, we can design a heuristic that achieves a better result in terms of maximizing the minumum utility.

The heuristic proceeds incrementally following a utility-driven criteria: starting from an empty placement, a new instance is given to the application with the lowest actual satisfaction; once the application instance is placed, the actual utilities are recalculated, and a new application is picked from a set of eliglible applications with lowest actual utility and placed in a randomly selected node with enough free memory available to host an instance of that application; the process is repeated iteratively until no more instances can be placed.

We compare this heuristic with our algorithm in 13 different scenarios, including those described in Figures 5.11 and 5.13. The selected scenarios represented a wide representation of the possible configurations. The results obtained after these tests is that the minimum utility achieved by our algorithm is, in average, in the range of the 95% to the 110% of the minimum utility achieved by the heuristic discussed above.

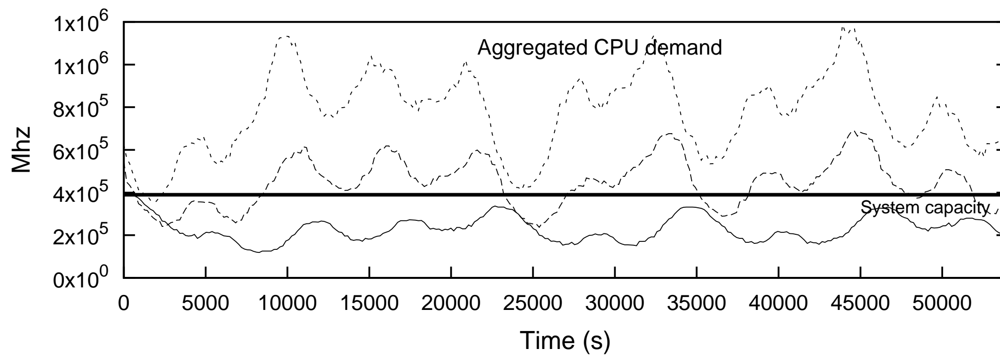


Figure 5.12 Minimizing placement changes: generated workload

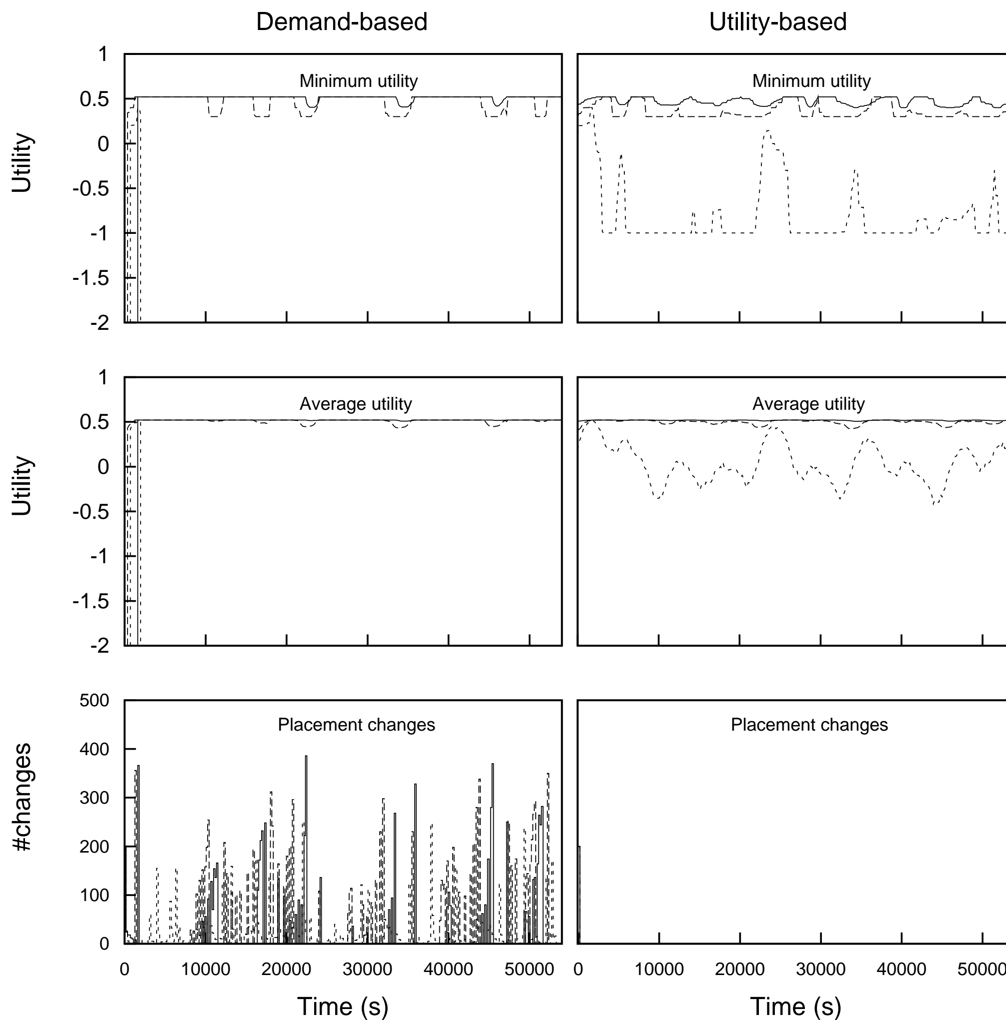


Figure 5.13 Minimizing placement changes. The solid, dashed, and dotted lines represent non-overloaded, partially overloaded, and overloaded scenarios, respectively. In the overloaded system test, the minimum and average utility achieved by the demand-based algorithm is below the range of the chart and the corresponding curve is not shown in the figure.

5.6.2 Long running-only workloads

In this subsection we present 4 experiments performed using the simulator. The focus of the experiments is to show how our technique allows an explicit and integrated management of heterogeneous workloads (exposed in section 5.6.3), but still performs well when managing only long running job workloads.

In Experiment One, we illustrate using a simple example how the hyphothetical utility discussed in subsection 5.4.2.4 guides the algorithm. In Experiment Two, we simulate a 25 node cluster to which a large number of identical jobs with identical deadline factors are submitted. In Experiment Three, we modify Experiment Two by randomly selecting (from three options) the deadline factor of each submitted job. Finally, in Experiment Four, we modify Experiment Two by randomly selecting (from three options) the execution time, maximum speed, and deadline factor for each submitted job. We then compare our algorithm against both FCFS and EDF.

Operation	Cost
Start VM	3.6s
Suspend VM	Memory demand * 0.0353s
Resume VM	Memory demand * 0.0333s
Live Migrate VM	Memory demand * 0.0132s

Table 5.1 Cost of virtualization operations

For the purpose of easily controlling the tightness of SLA goals, we introduce a relative goal factor which is define as a ratio of the relative goal of the job to its execution time at the maximum speed, $\frac{\tau_m - \tau_m^{\text{start}}}{t_m^{\text{best}}}$.

In all experiments, except for one (Experiment Four), the cost of virtualization operations (start, suspend, resume, migrate and move_and_resume) are considered. These costs were modelled using performance data obtained on our test systems running one of the most widely encountered virtualization products for Intel-compatible systems. These models show simple linear relationships between the VM memory footprint and the cost of the operation, as it can be seen in Table 5.1. Notice that the boot time observed for all our virtual machines was constant.

5.6.2.1 Experiment One: Hypothetical utility

In this experiment we illustrate how hypothetical utility (see section 5.4.2.4) guides our algorithm to make placement decisions. We use three jobs, J1, J2, and J3 with properties shown in Table 5.2. We also use a single node with resource capacities shown

in Table 5.2. The memory characteristics of the jobs and the node mean that the node can host only two jobs at a time. J1 can completely consume the node's CPU capacity, whereas J2 and J3, at maximum speed, can each consume only half of the node's CPU capacity.

We execute two scenarios, S1 and S2,which differ in the setting of the completion time factor for J2, which in turn affects the completion time goal for J2, as illustrated in Table 5.2. Note that J3 has a completion time factor of 1, which means that in order to meet its goal it must be started immediately after submission and that it must extecute with the maximum speed throughout its life.

To improve the clarity of mathematical calculations, we also use an unrealistic control cycle $T = 1s$.

Node	**Memory**	**CPU speed**
Capacity	2,000MB	1,000MHz

Job characteristics	**J1**	**J2**	**J3**
Start time [s]	0	1	2
Maximum speed [MHz]	1,000	500	500
Memory requirement [MB]	750	750	750
Work [Mcycles]	4,000	2,000	4,000
Minimum execution time [s]	4	4	8
Scenario 1			
Relative goal factor	5	**4**	1
Relative goal [s]	20	**16**	8
Completion time goal [s]	20	**17**	10
Scenario 2			
Relative goal factor	5	**3**	1
Relative goal [s]	20	**12**	8
Completion time goal [s]	20	**13**	10

Table 5.2 Properties of Experiment One

Figure 5.14 show cycle-by-cycle executions of the algorithm for S1 and S2, respectively. Rectangular boxes show the outstanding work, $\alpha_m - \alpha^*_m$, work done, α^*_m, value of hypothetical utility and corresponding CPU allocation for each job and various considered placement alternatives in subsequent control cycles. In most cycles in S1, only one placement is considered as the algorithm efficiently prunes the search space. Two alternative placements are considered in cycles 2 and 3. In cycle 2, we consider a placement, P1, that halves CPU allocation to J1 and starts J2 and a placement, P2, that leaves J1 running at full speed without starting J2. The same placement alternatives, P1

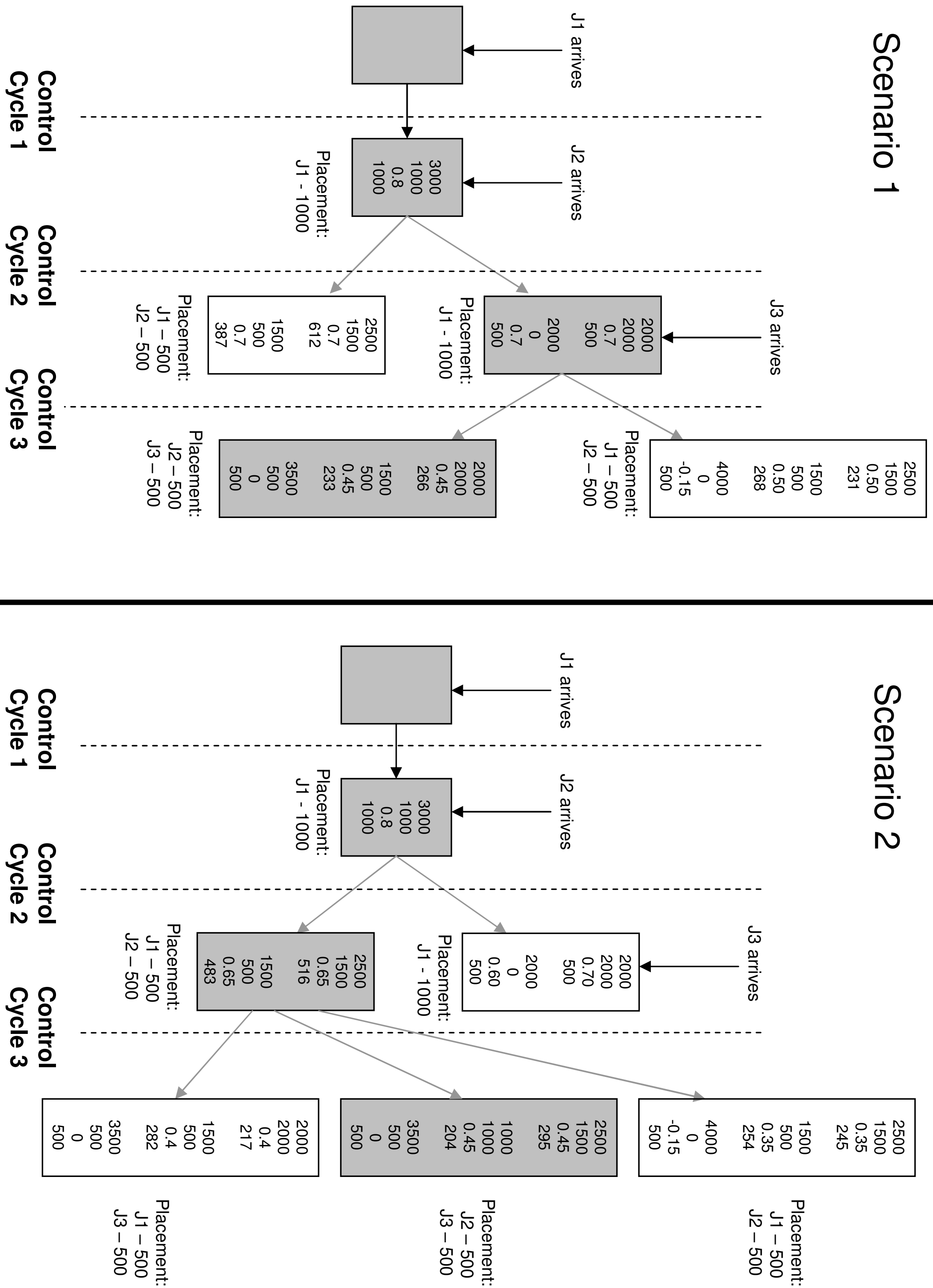


Figure 5.14 Experiment One: Description

and P2, are considered in cycle 2 of S2. In S1, these two placements have the same hypothetical utility of 0.7 for both jobs. Since P1 and P2 hve equal utilities, the algorithm opts to not make any changes and selects P2. In S2, due to the tighter completion time goal for J2, P2 has hypothetical utilities of 0.7 and 0.6 for J1 and J2 respectively, while P1 results in hypothetical utilities of 0.65 for both J1 and J2. Clearly, P1 is a better choice for S2.

The difference in hypothetical utilities of J2 in control cycle 2 between the two scenarios can be explained by looking at the maximum achievable utility of J2. If J2 is not started in cycle 2, and hence is started in cycle 3 or later, its earliest possible completion time is 19. In S1, this results in maximum achievable utility of 0.69 ($= (16-5)/16$), whereas in S2, it is only 0.58 ($= (12-5)/12$).

5.6.2.2 Experiment Two: Baseline

In this experiment, we examine the basic correctness of our algorithm by stressing it with a sequence of identical jobs, i.e., jobs with the same profiles and SLA goals. When jobs are identical, in the best scheduling strategy no placement changes (suspend, resume, migrate) should happen. This is the best possible behaviour in this case, as no benefit to job completion times (when looked on as a vector) would be gained by interrupting the execution of a currently placed job in order to place another job.

We consider a system of 25 nodes, each of which has four processors with properties shown in Table 5.3. To the system we submit 800 identical jobs with properties shown in Table 5.3. Jobs are submitted to the system using an exponential inter-arrival time distribution with an average inter-arrival time of 260s. This arrival rate is sufficient to cause queuing at some points during the experiment. The control cycle length is 600 s.

Observe that each job's maximum speed permits it to use a single processor, and so four jobs could run at full speed on a single node. However, the memory characteristics of the system mean that only three jobs will fit on a node at once. Consequently, no more than 75 jobs can run concurrently in the system. Each job, running at maximum speed, takes 17,600s to complete. The relative goal factor for each job is 2.7, resulting in a completion time goal of 47,520s ($2.7 * 17,600$), which is measured from the submission time.

The maximum achievable utility for a job described in Table 5.3 is 0.63. This utility will be achieved for a job that is started immediately upon submission and runs at full speed for 17,600s. In that case, the job will complete 29,920s before its completion time goal and thus will need a 37% of the time between the submission time and the completion time goal to complete. This utility is an upper bound for the job, and will be decreased if

Nodes	Memory	CPU Speed
Capacity	16,000MB	4x 3,900MHz

Job characteristics	Job
Maximum speed [MHz]	3,900 (1 CPU)
Memory requirement [MB]	4,320
Work [Mcycles]	68,640,000
Minimum execution time [s]	17,600
Relative goal factor	2.7
Relative goal [s]	**47,520**

Table 5.3 Properties of Experiment Two

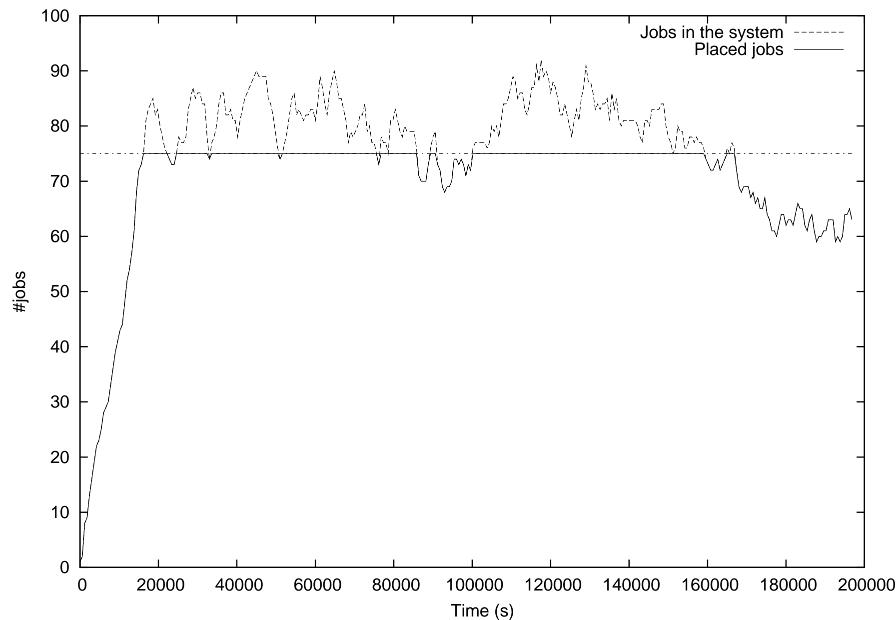


Figure 5.15 Experiment Two: Jobs in the system and jobs placed

queuing occurs.

Figure 5.15 shows the number of jobs in the system (already submitted), and the number of jobs that are placed at each moment in time. Note that the number of placed jobs never exceeds 75. In Figure 5.16, we show the average hypothetical utility over time as well as the actual utility achieved by jobs at completion time. When no jobs are queued, the hypothetical utility is 0.63 and it decreases as more jobs are delayed in the queue. Notice how the the utility achieved by jobs at completion time follows the shape of the hypothetical utility. As expected, there is a delay before the changes observed for the hypotethical are noticeable in the actual utility of completing jobs, caused by the fact that the hypothetical utility is predicting the actual utility that jobs will obtain at the time

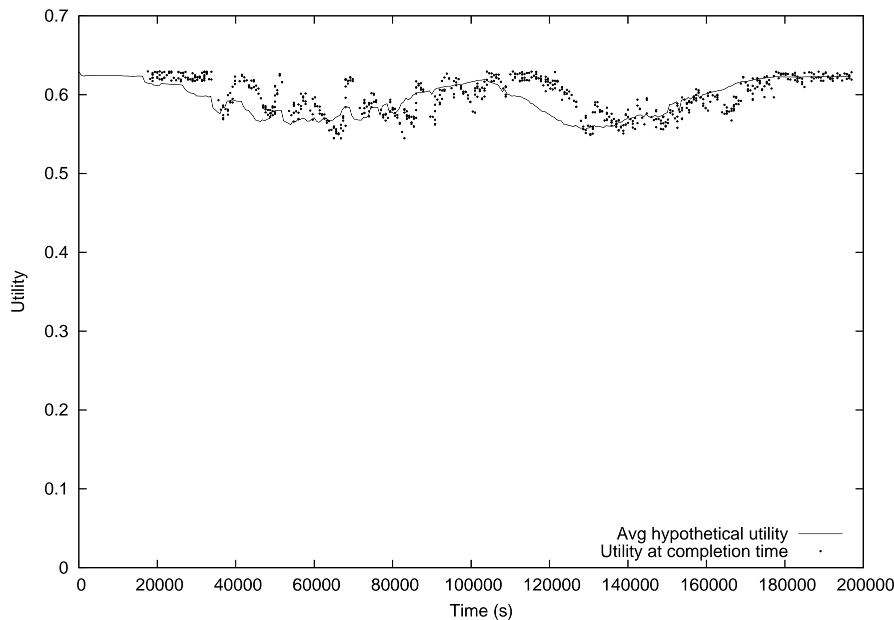


Figure 5.16 Experiment Two: Average hypothetical utility over time and actual utility achieved at completion time

they complete. The algorithm does not elect to suspend or migrate any jobs during this experiment, hence we do not include a figure showing the number of placement changes done by the algorithm. Finally, Figure 5.17 shows the execution time for the algorithm at each control cycle when running on a 3.2GHz Intel Xeon node. It can be observed that when all submitted jobs can be placed concurrently, the algorithm is able to take internal shortcuts, resulting in a significant reduction in execution time. In normal conditions, the algorithm produces a placement for this system in about 1.5s.

5.6.2.3 Experiment Three: Variable deadlines

In this experiment, we modify the conditions of Experiment Two (section 5.6.2.2) by introducing three different relative goal factors. For each job, a relative goal factor is randomly chosen from three different possibilities – 1.9 (with a probability of 30%), 2.5 (with a probability of 40%), and 10 (with a probability of 40%). All jobs have the same execution characteristics as in Experiment Two.

Mixing jobs with different relative goal factors introduces a new range of options for improvement for managing the workload. Jobs with more relaxed goals can be suspended to permit newly submitted jobs with tighter goals to be started in their place. However, the longer a job with a relaxed goal is suspended, the more difficult its goal becomes to satisfy, making it comparable to a newly submitted job with a tight goal. section 5.4.2.4 discusses in detail how the hypothetical utility guides the algorithm in the prediction of

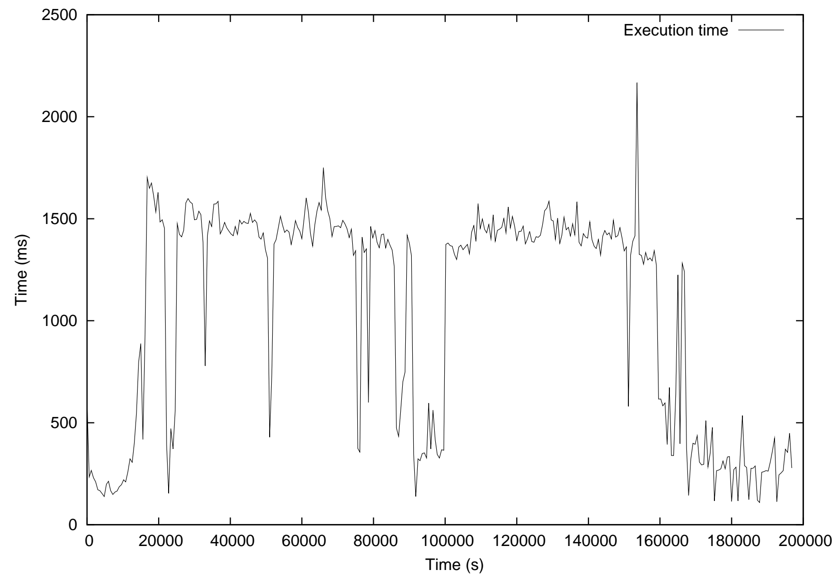


Figure 5.17 Experiment Two: Algorithm execution time

the achievable satisfaction for a job even when that job is currently not running.

Figure 5.18 shows the number of jobs in the system (already submitted), and the number of jobs that are placed at each moment in time. As in Experiment Two, we can never start more than 75 jobs simultaneously, owing to the memory constraints. Figure 5.19 shows the average hypothetical utility at each control cycle as well as the actual utility achieved by jobs at completion time, and the maximum achievable utility for jobs with relative goal factors 1.9, 2.5 and 10. Remember that all jobs have identical characteristics so their maximum achievable utility at the time they are submitted is the same for all jobs with the same relative goal factor. Note that the average hypothetical utility is no longer less than or equal to 0.63, as was the case in Experiment Two, as different deadline factors change the maximum achievable utility. However, the average hypothetical utility is still governed by the number of jobs in the system, and (in particular) the number of submitted jobs that are not currently placed (the job queue). The actual utility obtained by jobs at completion time is close to the maximum achievable utilities calculated for each relative goal factor. Our technique aims to equalize the utility at completion time for all jobs in the system, but in this scenario the presence of three different relative goal factors prevents it from achieving it – jobs with relative goal factor of 10 can achieve higher utility than the jobs with relative goal factor 1.9 without interfering. But notice that when the hypothetical utility decreases because some queueing is happening, less resources are allocated to the jobs with relative goal factor 10 in order to

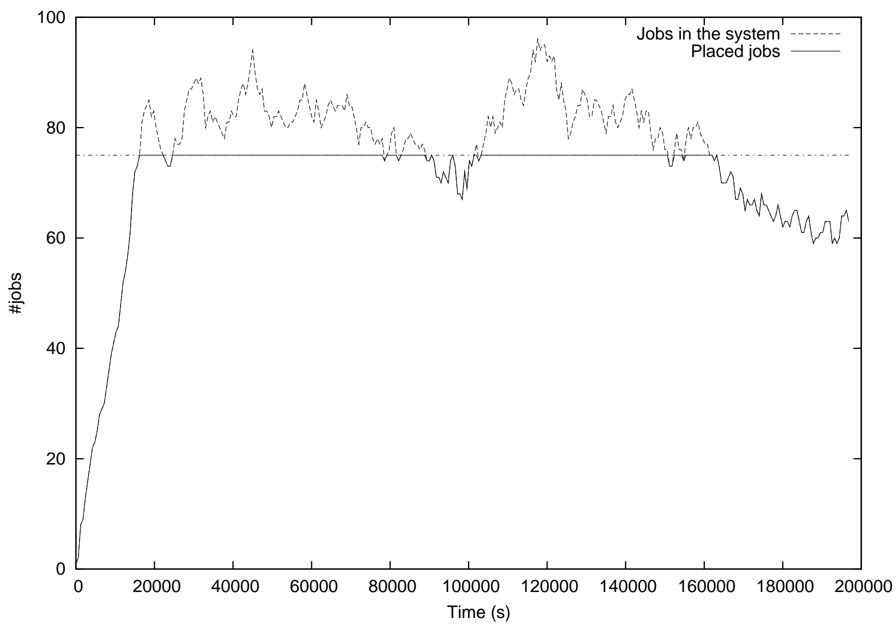


Figure 5.18 Experiment Three: jobs in the system and jobs placed

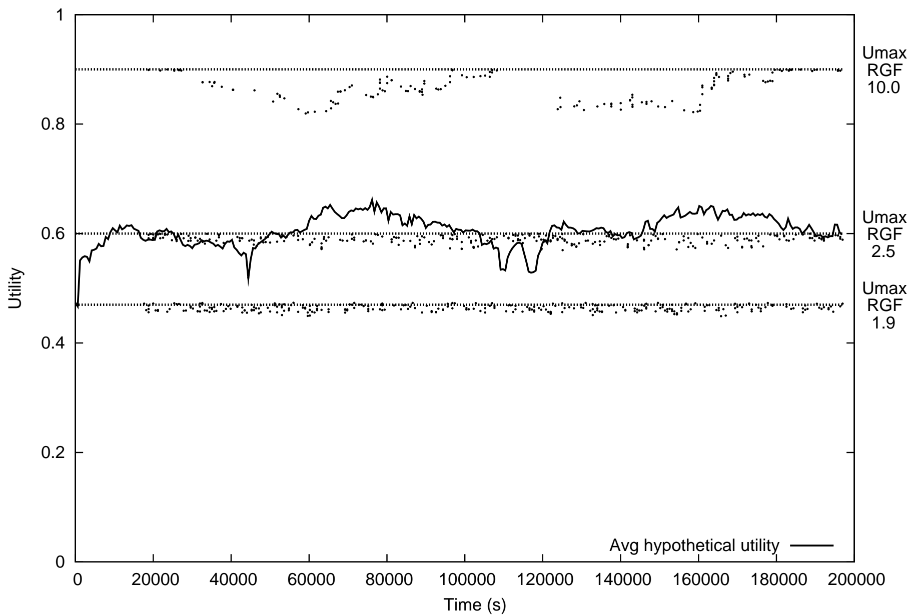


Figure 5.19 Experiment Three: average hypothetical utility over time and actual utility achieved at completion time

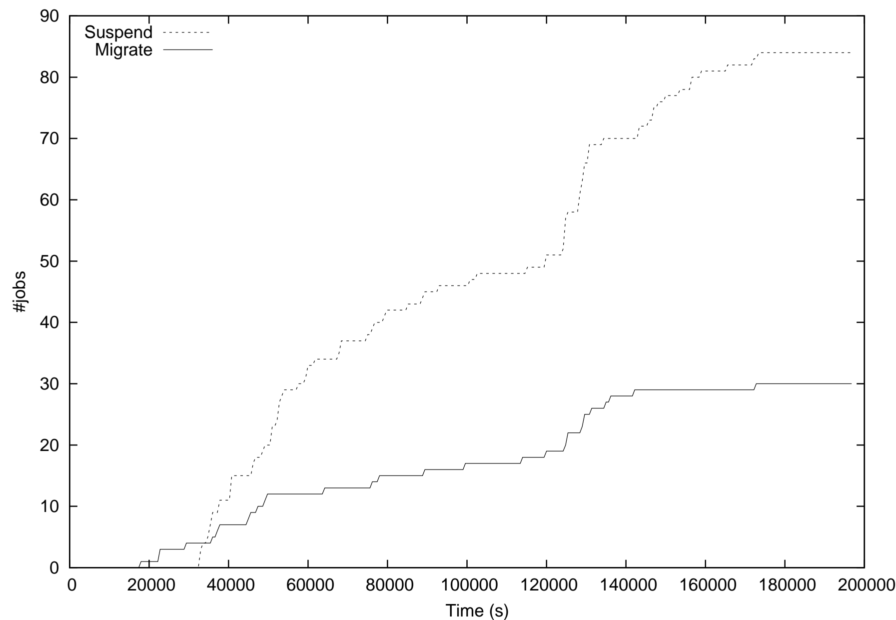


Figure 5.20 Experiment Three: total number of virtualization operations over time

maintain as high as possible the utility achieved by jobs with tighter relative goal factors at completion time. This fact can be observed short after times 40,000s and 100,000s – the algorithm decides to sacrifice the utility of jobs with relative goal factor 10 to keep jobs with relative goal factors 2.5 and 10 close to their maximum achievable utility values. Even the number of jobs with relaxed relative goal factor completing is reduced at some of these periods, allowing other jobs to be run instead.

Figure 5.20 shows that the algorithm elects to both suspend and migrate jobs during the course of this experiment. While the load on the system is the same in this experiment and the previous one, in this case the multiple deadline factors mean that making placement changes after a job has been started is a useful way to improve the utility of the system (as can be seen by comparing Figures 5.16 and 5.19).

5.6.2.4 Experiment Four: Randomized jobs

In this section, we simulate the system exercised with jobs of various profiles and SLA goals. The relative goal factors for jobs are randomly varied among values 1.3, 2.5, and 4 with probabilities 10%, 30%, and 60%, respectively. The job minimum execution times and maximum speeds are also randomly chosen from three possibilities – 9,000s at 3,900MHz, 17,600s at 1,560MHz, and 600s at 2,340MHz which are selected with probabilities 10%, 40%, and 50%, respectively.

We compare our algorithm (referred to as APC) with simple, effective, and well-known scheduling algorithms: Earliest Deadline First (EDF) and First-Come, First-

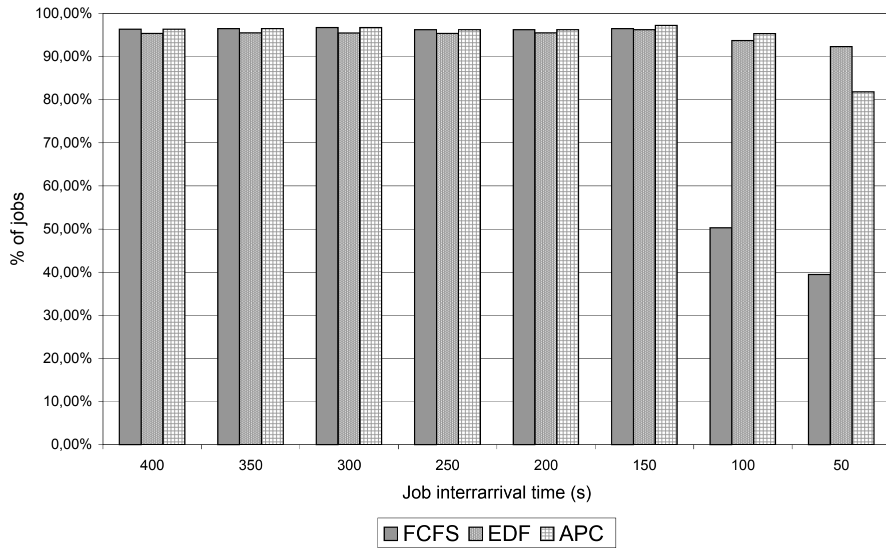


Figure 5.21 Experiment Four: Percentage of jobs that met the deadline

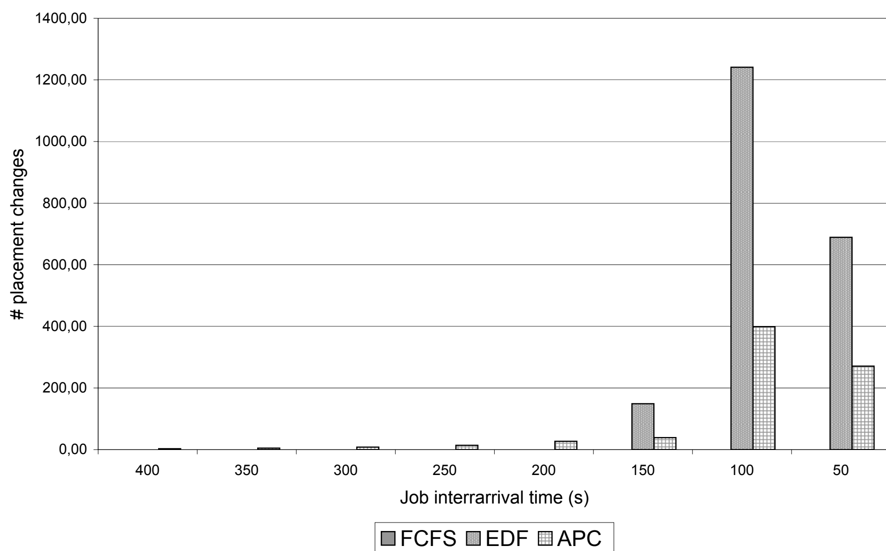


Figure 5.22 Experiment Four: Number of jobs migrated, suspended, and moved_and_resumed

Served (FCFS). Note that while EDF is a preemptive scheduling algorithm, FCFS does not preempt jobs. In both cases, a first-fit strategy was followed to place the jobs.

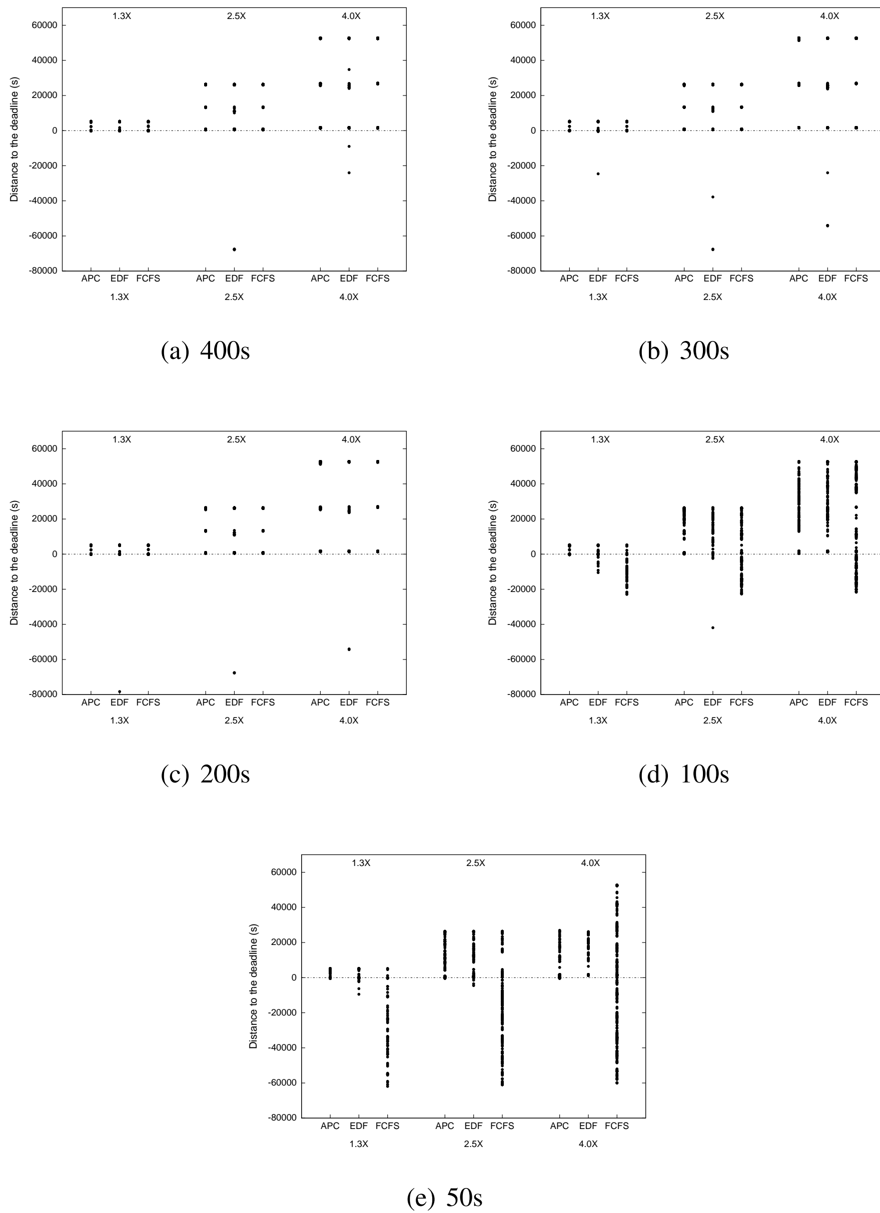
In this experiment, we use eight different inter-arrival times, ranging in increments of 50s from 50s to 400s, and continue to submit jobs until 800 have completed. The experiment is repeated for the three mentioned algorithms: our algorithm (APC), EDF, and FCFS.

Figure 5.21 shows the percentage of jobs that met their completion time goal. There is no significant difference between the algorithms when inter-arrival times are greater than 100s – this is expected, as the system is underloaded in this configuration. However, with an inter-arrival period of 100s or less, FCFS starts struggling to make even 50% of the jobs meet their goals. EDF and APC have a significantly higher, and comparable, ratio of jobs that met their goals. At a 50s inter-arrival time, the goal satisfaction rate for FCFS has dropped to 40%, and the goal satisfaction rate is actually higher for EDF than for APC.

Figure 5.22 shows the penalty for EDF's higher on-time completion rate at low inter-arrival times – EDF makes considerably more placement changes than does the APC once the inter-arrival time is 150s or less. Recall that FCFS is non-preemptive, and so makes no changes. Note that in this experiment, we did not consider the cost of the various types of placement changes – this does not change the conclusions, as our technique is making many fewer changes that EDF under heavy load. This figure, coupled with Figure 5.21, shows our algorithm's ability to making few changes to the system whilst still achieving a high on-time rate.

Figure 5.23 shows the distribution of distance to the deadline at job completion time for the three different relative goal factors (1.3, 2.5 and 4.0). We show these results for inter-arrival times of 400, 300, 200, 100, and 50 seconds, in Figure 5.23 (a), (b), (c), (d), and (e), respectively. Points with distance to the goal greater than zero, indicate jobs that completed before their goal. Observe that for inter-arrival times of 200s or greater, all three algorithms are capable of making the majority of jobs meet their goal, and the points for each algorithm are concentrated – for each algorithm and each relative goal factor, the distance points form three clusters, one for each job length.

However, as the inter-arrival time becomes 100s or less, the algorithms produce different distributions of distance to the goal. In particular, observe that for APC the data points are closer together than for EDF (this is most easily observed for the relative goal factor of 1.3). This illustrates that APC outperforms EDF in equalizing the satisfaction of all jobs in the system.



(a) 400s

(b) 300s

(c) 200s

(d) 100s

(e) 50s

Figure 5.23 Experiment Four: distribution of distance to the goal at job completion time, for five different mean interarrival times (50s to 400s)

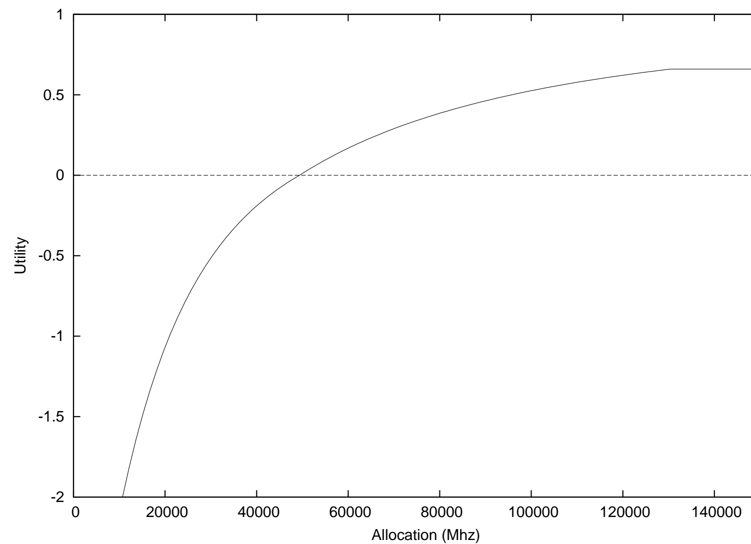


Figure 5.24 Heterogenenous workload: utility function for the transactional workload (utility as a function of allocated CPU power)

5.6.3 Heterogeneous workloads

In this experiment, we examine the behavior of our algorithm in a system presented with heterogeneous workloads. We demonstrate how our integrated management technique is applicable to combined management of transactional and long-running workloads. The experiment will show how our algorithm allocates resources to both workloads in a way that equalizes their satisfaction in terms of distance between their performance and performance goals. We compare our dynamic resource sharing technique to a static approach in which resources are not shared, and are pre-allocated to one type of work. This static approach is widely used today to run mixed workloads in datacenters.

We extend Experiment Two presented in section 5.6.2 by adding transactional workload to the system, and compare three different system configurations subject to the same mixed workload. In the first configuration we use our technique to perform dynamic application placement with resource sharing between transactional and long-running workloads. In the second and third configurations we consider a system that has been partitioned into two groups of machines, each group dedicated to either the transactional or the long-running workload. In both configurations, we use a First-Come First-Served (FCFS) to place jobs—FCFS was chosen because it is a widely adopted policy in commercial job schedulers. Notice that creating static system partitions is a common practice in many datacenters. In the second configuration, we dedicate 9 nodes to the transactional workload (9 nodes offer enough CPU power to fully satisfy this workload), and 16 nodes to the long-running workload. In the third configuration, we

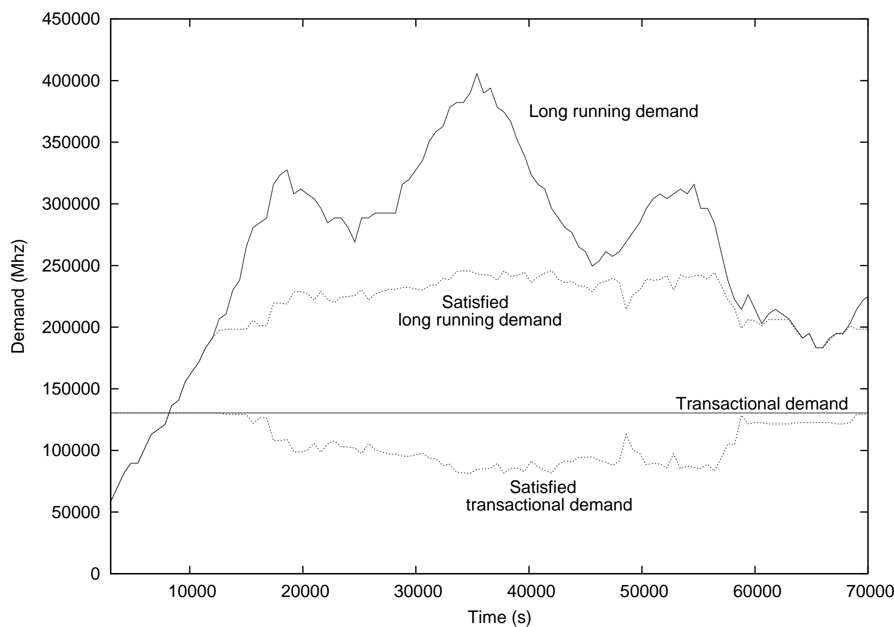


Figure 5.25 Heterogenenous workload: CPU power allocated to each workload and CPU demands to achieve maximum utility

dedicate 6 nodes to the transactional application and 19 to the long-running workload.

To simplify the experiment, the transactional workload is handled by a single application, and is kept constant throughout. Note that the long-running workload is exactly the same as that presented in Section 5.6.2.2. The memory demand of a single instance of the transactional application was set to a sufficiently low value that one instance could be placed on each node alongside the three long-running instances that fit on each node in Experiment One. This was done to ensure that the two different types of workload compete only for CPU resources (notice from Experiment One that a maximum of 3 long-running instances can be placed in the same node because of memory constraints).

Figure 5.24 shows the utility function used for the transactional workload. It shows how much CPU power must be allocated to this application for it to achieve a certain level of utility. The utility of transactional workloads is calculated as described in subsection 5.4.1. A utility of zero means that the actual response time exactly meets the response time goal: lower utility values indicate that the response time is greater than the goal (the requests are being serviced too slowly), and higher utility values indicate that the response time is less than the goal (the requests are being serviced quickly). The maximum achievable utility is around 0.66 in this case, at an approximate allocation of 130,000MHz. Allocating CPU power in excess of 130,000MHz to this application will not further increase its satisfaction: that is, it will not decrease the response time. This is normal behaviour for transactional applications – the response time cannot be reduced to

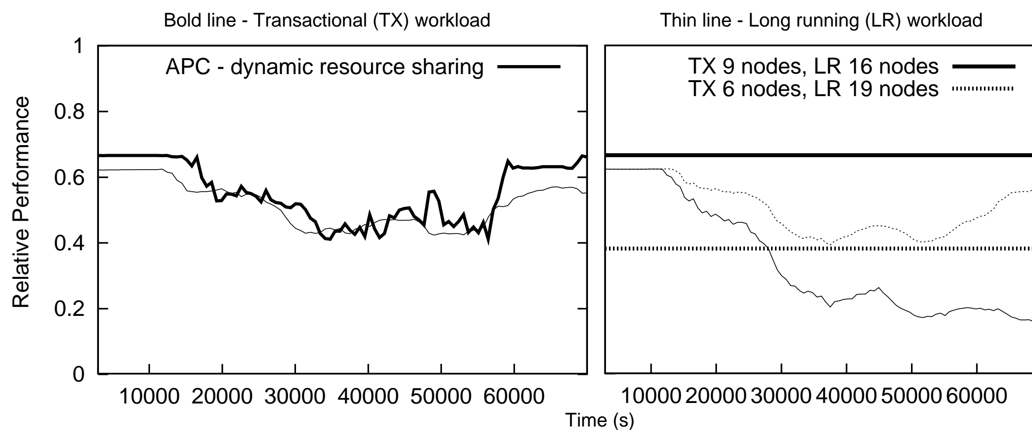


Figure 5.26 Heterogenenous workload: actual relative performance for the transactional workload and average calculated hypothetical relative performance for the long-running workload

zero by continually increasing the CPU power assigned.

The experiment starts with a system subject to the constant transactional workload used throughout, in addition to a small (insignificant) number of long-running jobs already placed. In this state, the transactional application gets as much CPU power as it can consume, as there is little or no contention with long-running jobs. As more long-running jobs are submitted, following the workload properties described in subsection 5.6.2.2, the hypothetical utility for those long-running jobs starts to decrease as the system becomes increasingly crowded. As soon as the hypothetical utility calculated for the long-running jobs becomes lower that the utility observed for the transactional workload (that is to say, no more resources can be allocated to the long-running workload without taking CPU power away from the transactional workload), our algorithm starts to reduce the allocation for the transactional workload and give that CPU power instead to the long-running workload, until the utility each achieves is equalized. At the end of the experiment the job submission rate is slightly decreased, what results in more CPU power being returned to the transactional workload again.

Figure 5.25 shows the utility for both of the workloads during the experiment. The utility for both workloads is continuously adjusted by dynamically allocating resources over time. Figure 5.25 shows the particular allocation at each moment of the experiment, as well as the CPU demand that would make each workload achieve its maximum utility. Notice how, as it was pursued, our technique makes an uneven distribution of resources in terms of CPU capacity, but it results in an even level of utility across the workloads.

Comparing these results with the results obtained for the static system configurations, shown in figures 5.26 and 5.26, reveal that the overall performance they deliver is lower than the performance observed for our dynamic resource sharing technique, and

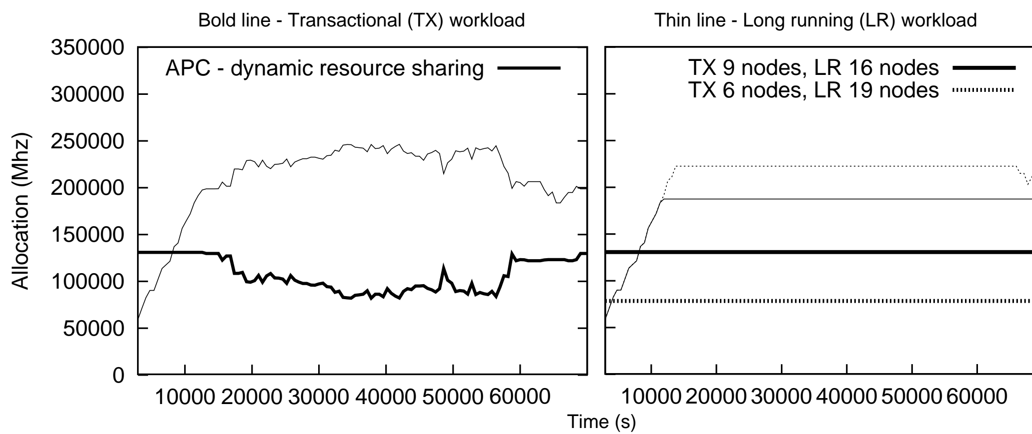


Figure 5.27 Heterogenenous workload: CPU power allocated to each workload for the three system configurations

the performance of both static and dynamic approaches is only comparable when the size of each machines partition exactly matches the resource allocation decided by our technique. Notice that when 9 nodes are dedicated to the transactional workload (offering more than the CPU power required to fully satisfy it), the utility achieved by the transactional workload is, as expected, 0.66—the maximum achievable. In this configuration, while transactional workload obtains good performance, long-running jobs struggle to meet their completion time goals, as shown by the low achieved utility values. When only 6 nodes are dedicated to the transactional workload, the utility that it achieves is consistently lower than that achieved with our dynamic resource sharing technique, while the performance benefits observed for the long-running jobs are not obvious when compared to the results obtained with our technique. Recall also that utility represents the relative distance to the goal achieved by each particular workload—distance to the response time goal for the transactional application and distance to the completion time goal for long-running jobs. Thus, utility is a direct measurement of the performance obtained by each workload.

5.7 Evaluation in the prototype

In this section we show three different experiments that stress the system prototype described in section 5.5. The experiments are here presented following the same order used for the simulation experiments. First, the system subject is to transactional-only workloads in section 5.7.1. Later, we present another experiment involving long-running only workloads in section 5.7.2. And finally we present an experiment in which long running jobs and transactional applications are mixed to produce a heterogeneous workload in section 5.7.3.

5.7.1 Transactional-only workloads

We deploy three applications, A1, A2, and A3 in a system composed of four homogeneous nodes. Applications are identical with respect to their per-request CPU requirements and their each request involves the same amount of computation interleaved with sleep time that simulates applications backend activity. We configure only one flow in each application, thereby making an application the smallest unit of management for the purpose of this experiment. Neither allocation restrictions nor collocation restrictions are defined, but placements are still subject to resource constraints, such as available memory of the nodes.

Property	**Node 1**	**Node 2**	**Node 3**	**Node 4**
Effective total CPU capacity[MHz]	3800	3800	3800	3800
Effective Memory capacity[MB]	2500	2500	2500	2500

Table 5.4 Node properties

We run the experiments on a cluster of IBM xSeries 335 servers, each containing 2 2.4GHz Intel Xeon processors with hyperthreading enabled. All the servers are connected through a switched gigagbit Ethernet network and run Linux 2.6.

The properties for the nodes and applications used in our experiments are shown in tables 5.4 and 5.5, respectively. Each node is able to support roughly 38 concurrent sessions of either application at a time, before overload protection becomes necessary. The base service time of each application is about 240 ms, which makes the response time goal for A3 rather aggressive. Also, notice that A1 and A2 use 40% of the memory capacity of a node each, while A3 uses 75%. Hence, A1 and A2 can both fit on a node, but neither of them can be placed together with A3. We configure such memory requirements by configuring a corresponding *maxHeapSize* value on application server JVM.

Property	Application 1	Application 2	Application 3
Memory demand[MB]	1200	1200	1800
Response time goal [ms]	1200	1200	350
Importance	50	50	50

Table 5.5 Application properties

5.7.1.1 Baseline experiment

Before experimenting with placement algorithm we baseline the system to observe the amount of CPU demand imposed by each application. We set all applications in manual mode, thus preventing any placement changes. We also configure memory requirements of applications such that all applications can be placed together on a node. Then we start an instance of each application on every node.

We choose total workload intensity so as not to overload the system. Then we vary the number of client sessions for applications within this limit.

Figure 5.28 shows the amount of CPU demand imposed on the system throughout the experiment. After a warm-up period, we start 95, 10, and 35 client sessions for A1, A2, A3, respectively (point A in Figure 5.28). This setting gives us a total number of client sessions of 140, which is slightly below the total that may be satisfied by our four-node system, 152. At point B, we increase the number of client sessions for A3 by 10 and correspondingly decrease the number of session for A1 by 10. At point D, we further increase the load for A3 by 20 clients and decrease load for A1 by the same amount. Finally, at point E, we further increase load for A3 by 19 clients, and correspondingly decrease the load for A1. Since throughout the experiment the system is never overloaded, the CPU usage observed across all nodes for each application gives us the CPU demand of this application.

5.7.1.2 Benefits of a utility-based placement

In the second experiment, we enable dynamic placement of applications and configure the applications as specified in Table 5.5. We configure initial placement such that A1 and A2 are both placed on three nodes, and A3 occupies one node. We run the same workload scenario as in the baseline experiment.

Figures 5.29 and 5.30 show observed response time for applications and their corresponding utility value. Figure 5.31 shows CPU capacity allocated to each application.

In phase A-B of the experiment, workload distribution is such that with the existing placement, all application CPU demands are satisfied and response time goals are met.

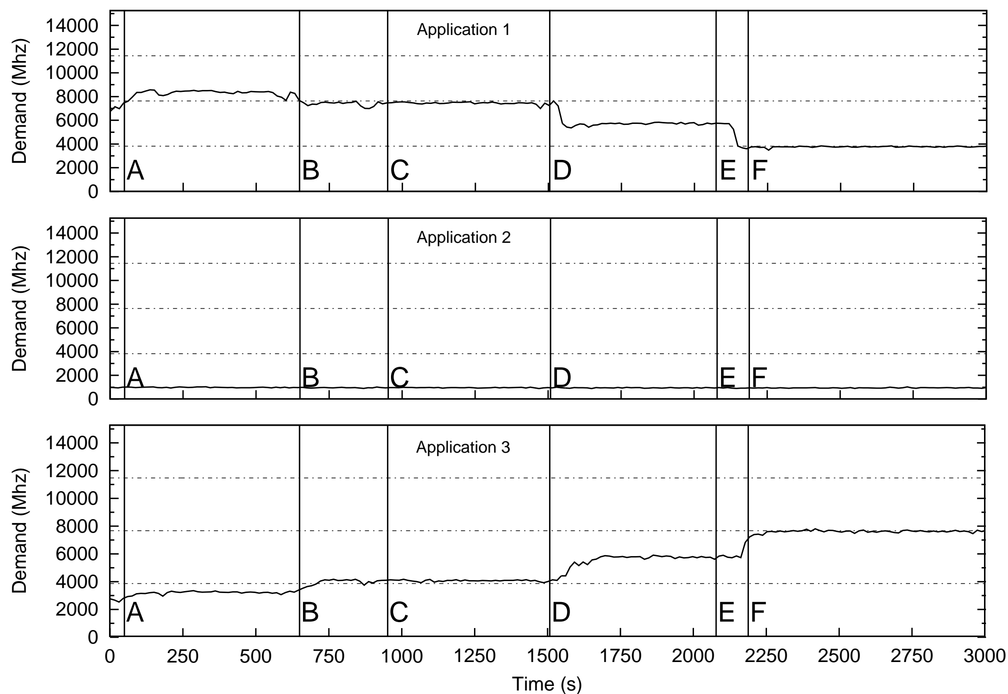


Figure 5.28 Demand

Utility values of applications are different as a result of them having different goals.

When workload changes at point B, A3 cannot be satisfied by a single node on which it is running. Its response time increases and it starts to violate the goal. At the same time, A1 experiences very good performance relative to its goal. At this point, it is reasonable to consider making a placement change that would remove A1 and A2 from one node and give this node to A3. The decision is made at point C. There is a rather long delay between the time workload has changed to the time placement changes. This time is used bue the controllers to accumulate enough statistics to build performance models for the new workload conditions. After the change is executed, within 2-3 minutes, response time for A1 returns to normal, while response time for A1 and A2 increases. Given the high goal for the latter applications, their utility is very moderately affected by this change.

At point D, we further increase load for A3 and decrease it for A1. We end up in well balanced state where all applications have almost identical performance.

At point E, we change the load again. We experience a similar change in performance as at point B. This time placement decision is done earlier, at point F. However, due to imprecise models, the decision is quickly reversed back, and remade again in the consecutive cycles of placement algorithm. This is clearly the evidence of instability, which in real deployments of our controller is avoided by introducing a stabilization delay after each placement change. In this experiment, we have not exercised this stabilization

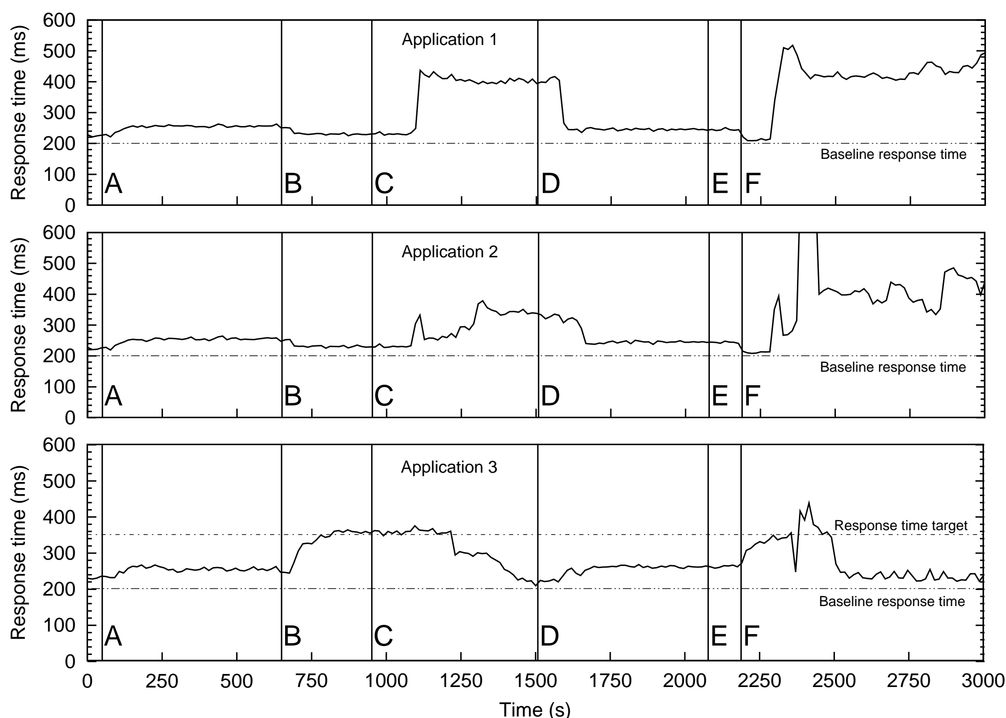


Figure 5.29 Response time

interval. In the final placement, A3 is allocated three nodes, and A1 and A2 share a single node. After the last placement change is completed, the response time for all applications is within the configured goal.

Let us examine the demand values obtained in the baseline experiment (Figure 5.28) and consider what demand-based algorithm would do when presented with these inputs.

In phase A-B, the demand of all applications can clearly be satisfied using the initial placement, hence no changes would happen.

In phase B-D, the demand of A3 cannot be satisfied by a single node. The demand-based algorithm should now look at the offered demand of the applications, which is about 1.05 nodes (4000MHz) for A3 and 2.23 nodes (8500MHz) for A1 and A2 combined. Clearly, when 4 nodes are available, satisfied demand is maximized (at 3.23 nodes) with the current placement, even though in the current placement A3 is missing the goal.

In phase D-E, the demand of A3 is 1.58 nodes (6000MHz) and the total for A1 and A2 is 1.82 nodes (7000MHz). At this time, the demand-based algorithm transfers a node from A1 and A2 to A3.

In the last phase of the test, the load for A3 is 2.1 nodes (8000MHz) and for A1 and A2 it is 1.31 nodes (5000MHz). Again, to maximize satisfied demand, it is better to leave placement unchanged, as this will result in satisfied demand of 3.31 nodes as opposed to

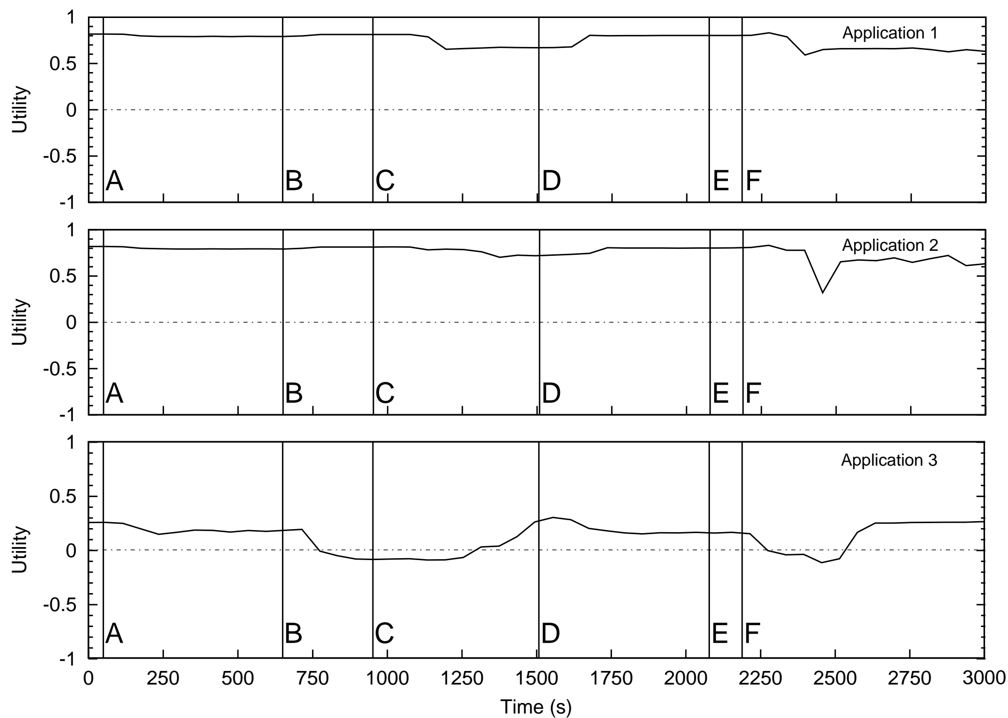


Figure 5.30 Utility

3.1 nodes if a change happened. Clearly, from the performance perspective, this is not the right decision.

5.7.2 Long running-only workloads

In this experiment we show the usefulness of server virtualization technology in the management of non-interactive workloads. We run a testcase that involves only long-running jobs shown in Table 5.6. In particular, we use BLAST [11], Lucene [5], ImageMagick [54] and POV-Ray [85] as representative applications for bioinformatics, document indexing, image processing and 3D rendering scenarios respectively. BLAST (Basic Local Alignment Search Tool) is a set of similarity search programs designed to explore all of the available sequence databases for protein or DNA queries. Apache Lucene is a high-performance, full-featured, open-source text search engine library written entirely in Java. In our experiments, we have run the example indexing application provided with the Lucene library to index a large set of files previously deployed in the filesystem. POV-ray (Persistence of Vision Raytracer) is a high-quality free tool for creating three-dimensional graphics. ImageMagick is a software suite to create, edit, and compose bitmap images.

The placement of jobs on nodes over time is shown in Figure 5.32.

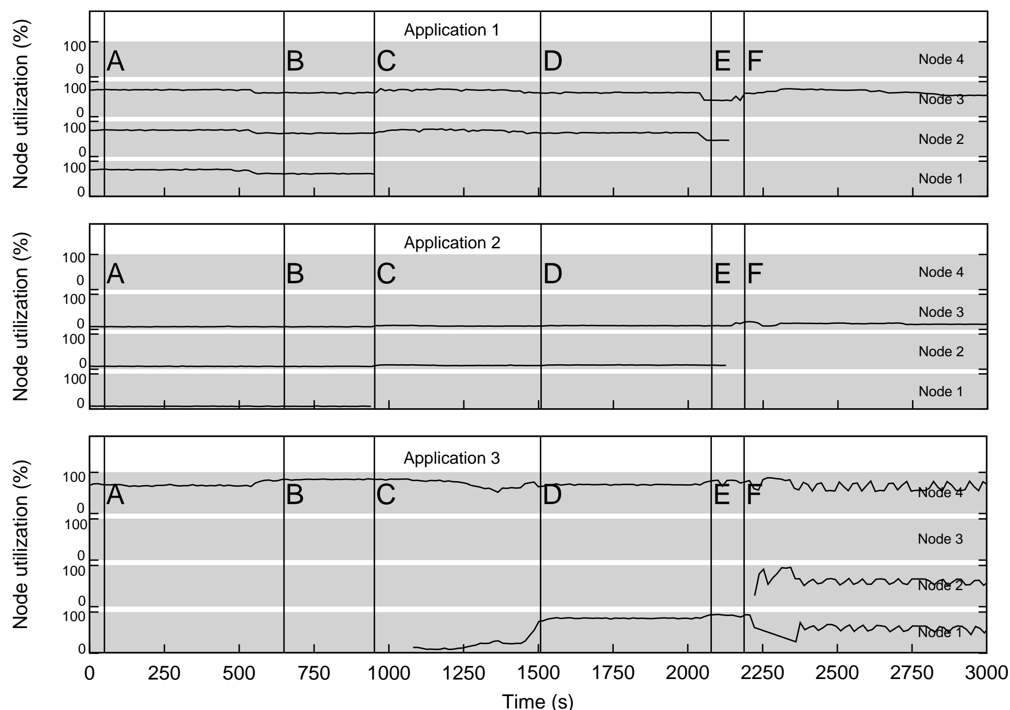


Figure 5.31 Per node allocation

	JOB1	JOB2	JOB3	JOB4	JOB5	JOB6
Job Type	BLAST	ImageMagick	POV-Ray	BLAST	Lucene	BLAST
Exec. time[min]	15	127	40	8	69	30
Class	Bronze	Platinum	Platinum	Platinum	Gold	Platinum
Max. speed [CPUs]	2	1	1	2	0.6	2
Memory [MB]	550	750	250	550	350	550

Table 5.6 Jobs used in experiments

We experiment with our system on a cluster of two physical machines, `xd018` and `xd020`, each with two 3GHz CPUs and 2GB memory. We used the XenSource-provided Xen 3.0.2 packages for RedHat Enterprise Linux 4.

We start the testcase by submitting JOB1 (A), which is started on `xd020` and takes its entire CPU power. Soon after JOB1 is submitted, we submit JOB2 and JOB3 (B), which both get started on `xd018` and each of them is allocated one CPU on the machine. Ten minutes later, we submit JOB4 (C), which has a very strict completion time requirement. In order to meet this requirement, APC decides to suspend JOB1 and start JOB4 in its place. Note that if JOB1 was allowed to complete before JOB4 is allowed to start, JOB4 would wait 5 min in the queue, hence it would complete no earlier than 13 min after its submission time, which would exceed its goal. Instead, JOB4 is started as soon

as it arrives and completes within 10 min, which is within its goal. While JOB4 is running, we submit JOB5 (D). However, JOB5 belongs to a lower class than any job currently running, and therefore is placed in the queue. When JOB4 completes, JOB5 is started on `xd020`. Since JOB5 consumes only 1 CPU, APC also resumes JOB1 and allocates it the remaining CPU. However, to avoid Xen stability problems in the presence of resource control mechanisms, we supress the resource control action, and resolving CPU contention is delegated to Xen hypervisor.

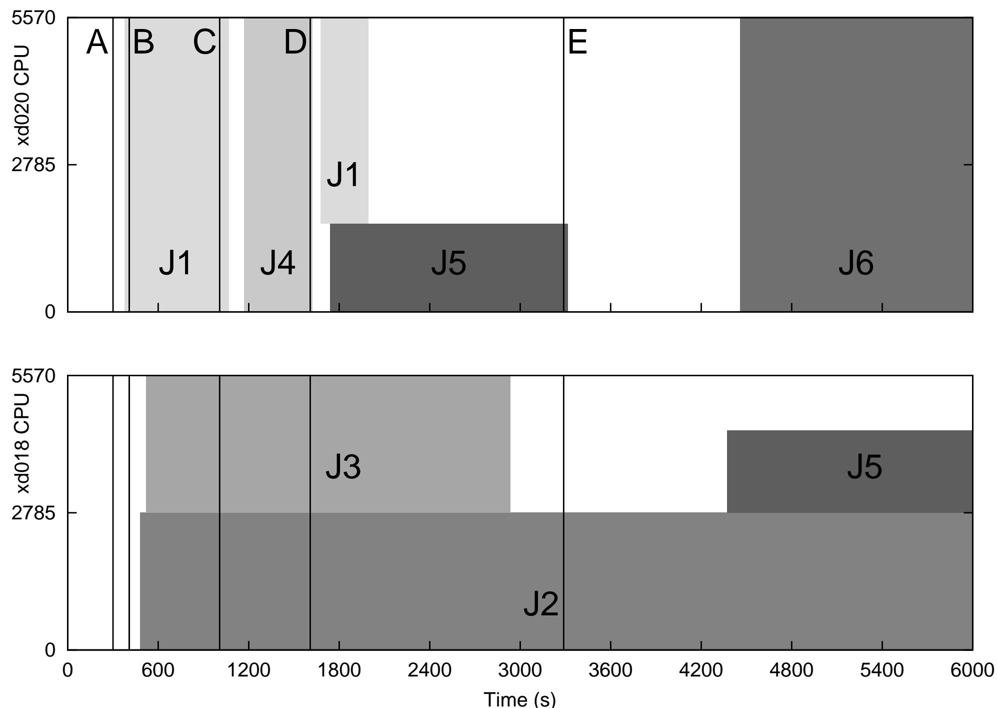


Figure 5.32 Node utilization by long running jobs.

In the next phase of the experiment, we demonstrate the use of migration. We wait until the completion of JOB1 and JOB3, and then we submit JOB6 (E). When JOB6 arrives, JOB2 and JOB5 each consume 1 CPU on `xd018` and `xd020` respectively. Since JOB6 requires 2 CPUs, APC may either (1) make it wait in the queue, (2) suspend JOB2 or JOB5, (3) collocate and resource control JOB6 with either JOB2 or JOB5, or (4) migrate either JOB2 or JOB5. Options (1)-(3) would result in wasted capacity on one or both machines. Moreover, options (1) and (3) would result in having *platinum* class job receive proportionately less CPU power than JOB5, whose service class is *gold*. This would clearly not be the optimal decision. Hence, APC decides (E) to move JOB4 to `xd018` (which it will now share with JOB5) and start JOB6 on the now-empty `xd020`.

Even though this experiment shows that APC correctly uses migration when machine

fragmentation makes it difficult to place new jobs, it also demonstrates a limitation of our optimization technique, which is currently oblivious to the cost of performing automation actions. Although in this experiment, 15 min is an acceptable price to pay for migrating a job, it is easy to imagine a scenario where this would not be the case.

5.7.3 Heterogeneous workloads

In this experiment, we use a single micro-benchmark web application that performs some CPU intensive calculation interleaved with sleep times, which simulate backend database access or I/O operations. We also use a set of non-interactive applications, which consists of well known CPU-intensive benchmarks, and which was already described in section 5.7.2.

In the experiments, we submit six different jobs, whose properties are shown in Table 5.6. We achieve differentiation of execution time by choosing different parameters, or by batching multiple invocations of the same application. All used applications except BLAST are single-threaded, hence they can only use one CPU. In addition, Lucene is I/O intensive, hence it cannot utilize a full speed of a CPU. We assign jobs to three service classes. Completion time goal for each job is defines relative to its profiled execution time and is equal to 1.5, 3, and 10 for *platinum*, *gold*, and *silver* class, respectively.

We experiment with our system on a cluster of two physical machines, `xd018` and `xd020`, each with two 3GHz CPUs and 2GB memory. We used the XenSource-provided Xen 3.0.2 packages for RedHat Enterprise Linux 4.

While testing our system, we determined that the resource control actions of our version of Xen are rather brittle and cause various internal failures across the entire Xen machine. Therefore, in our experiments, we have supressed resource control actions in the machine agent code.

We deploy *StockTrade* (a web-based transactional test application) in domain 1 on two machines `xd018` and `xd020`. We vary load to *StockTrade* using a workload generator that allows us to control the number of client sessions that reach an application. Initially, we start 55 sesstions and observe that with this load, response time of *StockTrade* requests is about 380ms and approaches response time goal of 500ms, as shown in Figure 5.33. At this load intensity, *StockTrade* consumes about 5/6 of CPU power available on both machines. Then we submit JOB5 (A). Recall from Table 5.6 that JOB5 is associated with *platinum* service class and therefore has completion time goal equal to 1.5 to its expected execution time. After a delay caused by the duration placement control cycle (B) and domain starting time, JOB5 is started (C) in domain 2 on `xd020` and, in the absence of any resource control mechanism, allocates it the entire requested CPU speed, which

is equivalent to 0.6 CPU. As a result of decreased CPU power allocaton to domain 1, on `xd020`, the response time for *StockTrade* increases to 480ms, but it stays below the goal. A few minutes after submitting JOB 5, we submit JOB1 (D), whose service class is *bronze*. JOB1 has a very relaxed completion time goal but it is very CPU demanding. Starting it now would take 2CPUs from the current *StockTrade* allocation.

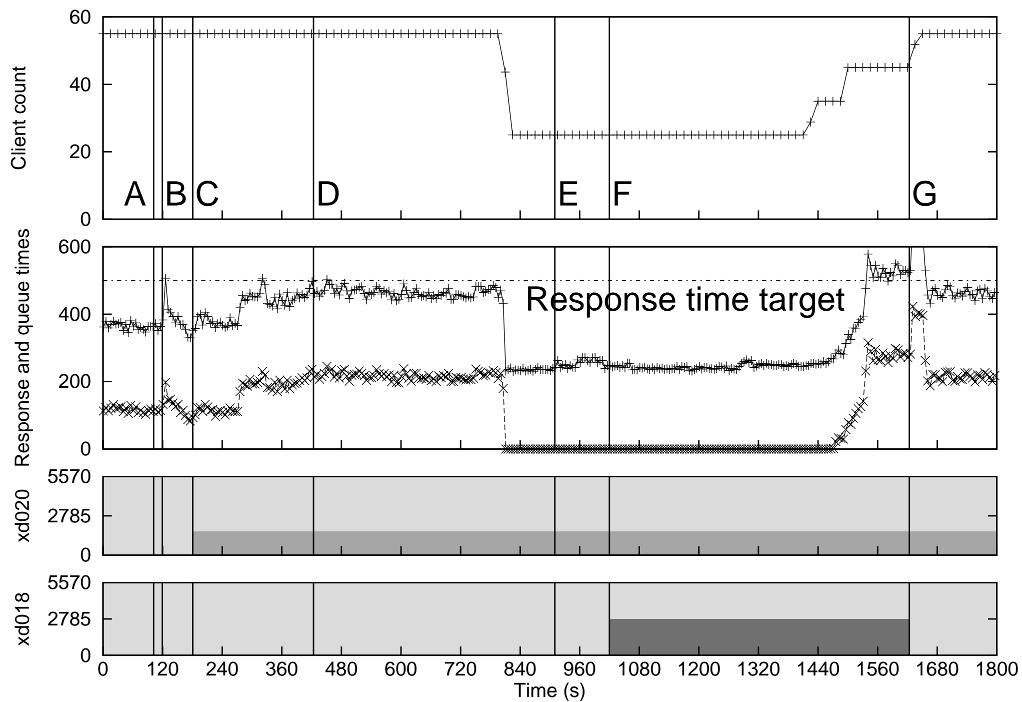


Figure 5.33 Response time for *StockTrace* and job placement on nodes.

At 800s since the begining of the experiment, we reduce load to *StockTrade* to 25 concurrent client sessions. When CPU usage of *StockTrade* reduces to about 50% of each machine, the placement controller decides (E) to start JOB1 (F) on `xd018`. After 1000s, we increase the number of client sessions back to 55, placement controller suspends JOB1 (G). Typically, JOB1 will later be resumed when any of the following conditions occur: (1) JOB5 completes, (2) load to *StockTrade* is reduced, or (3) JOB1 gets close enough to its target completion time so as to necessitate its resumption, even at the expense of worsened performance for *StockTrade*. However, the occurrence of the third condition indicates that the system is under-provisioned, hence SLA violation may not be avoided. This simple experiment demonstrates that with the use of server virtualization, our system is able to balance resource usage between web and non-interactive workloads.

5.8 Related Work

In this Chapter we present a technique that allows the management to high-level goals of collocated long-running and transactional workloads in virtualized environments. We use utility functions to model the satisfaction of both long-running jobs and transactional workloads for a particular resource allocation – the different types of workload have different characteristics, and different performance goals, and utility functions offer a mechanism to make their performance comparable. We run both workloads inside virtual machines, in order to properly manage their performance, and our management also exploits the clustering nature of transactional workloads.

Both the use of utility functions for workload management, and managing clusters of virtual machines, are areas already studied in the literature, but our approach is the first one that combines them to successfully manage heterogeneous workloads with fairness goals.

Heterogeneous workloads The explicit management of heterogeneous workloads was previously studied in [96], in which a number of CPU shares were manually allocated to run mixed workloads on a large multiprocessor system. This was a static approach, and did not run workloads within virtual machines. Virtuoso [71] describes an OS scheduling technique, VSched, for heterogeneous workload VMs. VSched enforces compute rate and interactivity goals for both non-interactive and interactive workloads (including web workloads), and provides soft real-time guarantees for VMs hosted on a single physical machine. VSched could be used as a component of our system for providing resource-control automation mechanisms within a machine, but our approach is broader as it addresses resource allocation for heterogeneous workloads across a cluster of physical machines. Our work could also be compared to numerous prior publications on job scheduling [39]. With this contribution we do not claim to advance the field of job scheduling. Nevertheless, we show that virtualization technology offers important control mechanisms that can be used to facilitate more effective scheduling of jobs and should be considered in future job scheduling techniques.

Utility-driven workload management The use of utility-driven strategies to manage workloads was first introduced in the scope of real-time work schedulers to represent the fact that the value produced by such a system when a unit of work is completed can be represented in more detail than a simple binary value indicating whether the work met its or missed its goal. In [56], the completion time of a work unit is assigned a value

to the system that can be represented as a function of time. Other work in the field of utility-driven management include memory- [40] and energy-aware [123] utility-driven scheduling, are summarized in [90] with special focus on real-time embedded systems. In [8], the authors present a utility-driven scheduling mechanism that aims to maximize the aggregated system utility. In contrast, our technique does not focus on real-time systems, but on any general system for which performance goals can be expressed as utility functions. In addition, we introduce the notion of fairness into our application-centric management technique – our objective is not to maximize the system utility, but instead to at least maximize the lowest utility across long-running jobs and transactional applications present in the system.

Outside of the realm of the real-time systems, the authors of [34] focus on a utility-guided scheduling mechanism driven by data management criteria, since this is the main concern for many data-intensive HPC scientific applications. In our work we focus on CPU-bound heterogeneous environments, but our technique could be extended to observe data management criteria by expanding the semantics of our utility functions.

In our work we consider monotonic and continuous utility functions to represent the satisfaction of both transactional and long-running workloads, but other approaches have been studied in the literature. In [69], the authors discuss the best shape for the utility functions (extending the work presented in [68]). The authors of [30] use user-defined utility functions to represent the value of resources, and their market-based batch scheduler is driven by these utility functions to allocate resources.

Management of clusters of Virtual Machines There is also some previous work in the area of managing workloads in virtual machines. Management of clusters of virtual machines is addressed in [42] and [32]. The authors of [42] address the problem of deploying a cluster of virtual machines with given resource configurations across a set of physical machines. Czajkowski et al. [32] define an API for a Java VM that permits a developer to define resource allocation policies. In [128] and [82], a two-level control loop is proposed to make resource allocation decisions within a single physical machine, but does not address integrated management of a collection of physical machines. The authors of [115] study the overhead of a dynamic allocation scheme that relies on virtualization as opposed to static resource allocation. Their evaluation covers CPU-intensive jobs as well as transactional workloads, but does not consider mixed environments. Neither of these techniques provides a technology to dynamically adjust allocation based on SLA objectives in the presence of resource contention.

VMware DRS [122] provides technology to automatically adjust the amount of

physical resources available to VMs based on defined policies. This is a achieved using live-migration automation mechanism provided by VMotion. VMware DRS adopts a VM-centric view of the system: policies and priorities are configured on a VM-level. A approach similar to VMware DRS is proposed in [64], which proposes a dynamic adaptation technique based on rearranging VMs so as to minimize the number of physical machines used. The application awareness is limited to configuring physical machine utilization thresholds based on off-line analysis of application performance as a function of machine utilization. Runtime requirements of VMs are taken as a given and there is no explicit mechanism to tune resource consumption by any given VM.

Unlike [122] and [64], our system takes an application-centric approach—the virtual machine is considered only as a container in which an application is deployed. Using knowledge of application workload and performance goals, we can utilize a more versatile set of automation mechanisms than [122] and [64]. We can vary the number of VMs over which a clustered application is provided, suspend a VM for a long-running job, and decide how much resource a VM should be allowed to consume. In addition, our system is able to utilize various kinds of virtualization for various applications. For example, for web workloads, we chose to use virtualization technology provided by application server middleware technology.

The adaptation problem for virtual environments has also been studied in [107]. The problem there is to place virtual machines interconnected using virtual networks on physical servers interconnected using a wide area network. Given the nature of the network, the primary concern in this problem is to allocate network bandwidth for virtual networks. VMs may be migrated, but their resource allocation is taken as a given. Our problem deals with datacenter environments, in which network bandwidth is of lesser concern, and our solution considers VM placement as well as resource allocation.

Application placement problem Placement problems in general (either in the presence of virtualization technologies or not) have also been studied in the optimization literature, frequently using techniques including bin packing, multiple knapsack problems, and multi-dimensional knapsack problems [61]. The optimization problem that we consider presents a non-linear optimization objective while previous approaches [58, 65] to similar problems address only linear optimization objectives. In [114], the authors evaluate a similar problem to that addressed in our work (but restricted to transactional applications), and use a simulated annealing optimization algorithm. Their optimization strategy aims to maximize the overall system utility while we focus on first maximizing the lowest utility across applications, which increases fairness and prevents starvation, as was shown

in [25]. In [124], a fuzzy logic controller is implemented to make dynamic resource management decisions. This approach is not application-centric – it focuses instead on global throughput – and considers only transactional applications. The algorithm proposed in [110] allows applications to share physical machines, but does not change the number of instances of an application, does not minimize placement changes, and considers a single bottleneck resource.

5.9 Summary

In this Chapter we have summarized the third contribution of this thesis that consists in a technique that allows an integrated management of heterogeneous workloads, composed of transactional applications and long running jobs, dynamically placing the workloads in such a way that equalizes their satisfaction. We use utility functions to make the satisfaction and performance of both workloads comparable. It is based on a utility-driven application placement algorithm to achieve equalized satisfaction across applications. Additionally it minimizes the number of placement changes necessary to achieve its goals.

The system has been implemented and integrated with a commercial application server middleware, what provides the support for executing placement decisions. Our system is driven by high-level application goals and takes into accout the application satisfaction with how well the goals are met. We have demonstrated, both using a real-system and a simulator, that this approach improves satisfaction fairness across applications compared to existing state-of-the-art solutions. We have also demonstrated that the system consistently achieves its goals independently of the workload conditions and the system configuration. It has been demonstrated that it not only performs well in presence of heterogeneous workloads but it also shows consistent performance in presence only of long running jobs as compared to other well-knonw scheduling algorithms.

The system introduces several novel features. First, it allows heterogeneous workloads to be collocated on any server machine, thus reducing the granularity of resource allocation. Second, our approach uses high-level performance goals (as opposed to lower-level resource requirements) to drive resource allocation. Third, our technique exploits a range of new automation mechanisms that will also benefit a system with a homogeneous, particularly non-interactive, workload by allowing more effective scheduling of jobs.

The work performed in this area has resulted in the following publications:

[25] D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguadé. **Utility-based Placement of Dynamic Web Applications with Fairness Goals**. In 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), April 7-11, 2008

[24] D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguadé. **Managing SLAs of heterogeneous workloads using dynamic application placement**. In the 7th IEEE Symposium on High Performance Distributed Computing (HPDC 2008). An extended version is available as Technical Report RC24469, IBM Research, Jan. 2008.

[95] M. Steinder, I. Whalley, D. Chess, D. Carrera, I. Gaweda, **Server virtualization in autonomic management of heterogeneous workloads**. 10th IFIP/IEEE International Symposium on Integrated Management (IM 2007), 2007

Chapter 6

Conclusions and future work

6.1 Conclusions

In this thesis we presented three complementary steps toward the creation of adaptive execution environments for application servers, with incremental value in terms of performance benefits, and also increasing level of automation. First we presented a highly-detailed automatic performance monitoring framework that makes possible a deep understanding of the behavior of the complex execution stack created by an application server middleware. The tool can be automatically driven by high-level performance metrics in order to trace the system without need of human interaction when the delivered performance doesn't meet some user-defined thresholds. Secondly, the use of the monitoring framework to analyze secure web applications uncovered a potential focus of improvement in the architecture of application servers. We proposed and evaluated a new architectural design for application servers that improves the efficiency of the intensive connection management performed by such a middleware resulting in a best performance in terms high-level objectives whereas the adaptability of the application server to the workload conditions is increased by reducing the need of human interaction to keep the server properly configured. Finally, we proposed and evaluated an automatic resource allocation technique for heterogeneous workloads that takes full advantage of the virtualization technology to enforce resource allocation decisions. The technique uses utility functions to make the performance of both transactional and long-running workloads explicitly comparable. We have proven that our technique achieves its performance objectives whereas it fairly satisfies all applications. Next we summarize in more detail the work presented in this thesis.

6.1.1 Automatic performance monitoring framework

The first contribution of this thesis consists in the creation of an automatic monitoring framework specially focused on web applications. Our proposal correlates detailed system information with high-level performance data in order to make possible a complete performance analysis of web applications.

The monitoring framework is composed of three differentiated tools that work coordinately to create a powerful performance analysis environment. Java Instrumentation Suite [23] (JIS) is a deep monitoring tool that produces extremely detailed insight on the system behavior – ranging from the operating system level to the user application code. The data collected by the monitoring tool can be later studied using Paraver [38], a powerful analysis and visualization tool. The JACIT tool (Java Automatic Code Interposition Tool) can be used to modify existing bytecodes of a Java application without

need of source code availability. JIS is the major component in the first contribution of this thesis.

Four levels are considered by JIS when tracing a system: 1) operating system, 2) JVM, 3) middleware (application server) and 4) user application. Information collected by all levels is finally correlated and merged to produce an execution trace file.

Information obtained from the operating system level covers threads' state and system calls. Thread information is obtained directly from the Linux scheduler routine and information from syscalls (I/O, sockets, memory management, thread management) is obtained by intercepting some entries of the syscall table. This task was divided in two layers: one based in a kernel source code patch and the other in a system device and its corresponding driver. When working with Java-based applications, collected information is limited to the JVM process, and other processes on the system are ignored.

Java semantics are just considered inside the JVM. Because of this, comprehensive instrumentation of Java applications must be composed, in part, by internal JVM information. This information is used by JIS to include Java application semantics on its instrumentation process

JIS allows the generation of events from both the middleware and the user application levels, that are later available in the tracefiles. Notice that the importance of these events is that they are generated on execution time and are automatically correlated with all the other performance data. Information relative to services (i.e. servlets and EJBs) or transactions can be obtained from the middleware level. Additionally, the user application code can be modified to inject events into the tracefile too, what improves the understanding of the real cost and effects of any portion of the user application code.

Finally, we have shown a performance-driven environment for WebSphere that can be used to control the fine-grain monitoring framework, what has been proved to be enough to have a "24x7" deeply traced server without need of human cooperation until the analysis step.

6.1.2 Adaptive architecture for application servers

The second contribution of this thesis consists in a hybrid web container architecture that combines the best characteristics of both a multithreaded design and an event-driven model. The proposed implementation into the Tomcat 5.5 code offers a slightly better performance than the original multithreaded Tomcat server when it is tested for a static content application, and a remarkable performance increase when it is compared for a dynamic content scenario, where each user session failure can be put into relation with business revenue losses. Additionally, the natural way of programming introduced by the

multithreading paradigm can be maintained for most of the web container code. But even more important, we have shown how the hybrid architecture naturally adapts to dynamically changing workload conditions without need to be reconfigured. This desired adaptability property reduces the need of human interaction in order to keep the server properly configured.

A preliminary study, that motivated this contribution, consisted of measuring Tomcat's vertical scalability (i.e. adding more processors) when using SSL and analyzing the effect of this addition on the server's scalability. The results confirmed that, since secure workloads are CPU-intensive, running with more processors makes the server able to handle more clients before overloading, with the maximum achieved throughput improvement ranging from 1.7 to 2.8 for 2 and 4 processors respectively. In addition, even when the server has reached an overloaded state, a linear improvement on throughput can be obtained by using more processors. The second part involved the analysis of the causes of server overload when running with different numbers of processors by using a performance analysis framework. The analysis revealed that the server can be easily overloaded if connections are not properly managed, demonstrating the convenience of developing advanced connection management strategies to overcome such a complicated scenario.

6.1.3 Integrated management of heterogenenous workloads

The third contribution of this thesis consists in a technique that allows an integrated management of heterogeneous workloads, composed of transactional applications and long running jobs, dynamically placing the workloads in such a way that equalizes their satisfaction. We use utility functions to make the satisfaction and performance of both workloads comparable. It is based on a utility-driven application placement algorithm to achieve equalized satisfaction across applications. Additionally it minimizes the number of placement changes necessary to achieve its goals.

The system has been implemented and integrated with a commercial application server middleware, what provides the support for executing placement decisions. Our system is driven by high-level application goals and takes into account the application satisfaction with how well the goals are met. We have demonstrated, both using a real-system experiment and a simulation, that this approach improves satisfaction fairness across applications compared to existing state-of-the-art solutions. We have also demonstrated that the system consistently achieves its goals independently of the workload conditions and the system configuration. It has been demonstrated that it not only performs well in presence of heterogeneous workloads but it also shows consistent performance in presence

only of long running jobs as compared to other well-known scheduling algorithms.

The system introduces several novel features. First, it allows heterogeneous workloads to be collocated on any server machine, thus reducing the granularity of resource allocation. Second, our approach uses high-level performance goals (as opposed to lower-level resource requirements) to drive resource allocation. Third, our technique exploits a range of new automation mechanisms that will also benefit a system with a homogeneous, particularly non-interactive, workload by allowing more effective scheduling of jobs.

We believe that using server virtualization for automatic performance management is an exciting research problem. It requires novel resource allocation algorithms capable of reasoning of many new automation mechanisms and many different levels at which virtualization may be provided. It also requires techniques to deploy and manage virtual images and to model multi-level relationships among resources. We must also consider multiple resources that may be virtualized.

6.2 Future work

The work performed in this thesis opens several interesting ways that can be explored as a future work.

- Traditionally, long running workload management techniques rely on good work profilers and accurate user estimates to make the job. Unfortunately, user estimates tend to be inaccurate and, in turn, unreliable to perform optimal job scheduling. Work profiling provides a good source of information for the management of jobs, but needs stable execution environments for work profiles to be usable in the future. Virtualized environments are all but unstable as far as virtual containers can be dynamically altered, with modified resource allocations. Performing work profiling in virtualized environments is a challenging task that implies not only the creation of accurate work profiles but also complex models that define in what proportion resources must be provisioned to one particular job in order for it to make significant progress.

- The use of a hybrid architecture to manage client connections improves the adaptability of the server to varying workloads but also eliminates the intrinsic overload protection mechanism that is present in the multithread model. In a multithreaded server, no more connections than existing worker threads can be established but an event-driven or hybrid model can accept a number of client connections that is far larger than what can be processed with reasonable good performance. In the context of load-balanced clustered deployments, protecting against overload conditions is not a must since load balancers should take care of such a situation. But in the scope of commodity solutions for which the hybrid architecture can introduce important benefits, introducing some kind of overload control mechanism could be crucial. Control theory could be a key player for this subject.

- Companies are now focusing more than ever on the need to improve energy efficiency. In addition to the cost of energy, a new challenge for them is the increasing social pressure to reduce their carbon footprint. Commercial power consumption is a major factor in rising atmospheric CO2 levels and data center equipment is stressing the power and cooling infrastructure to a level that implies that data center emissions are increasing faster than any other carbon emission. Dynamic workload management can be adapted to meet not only performance goals but energy efficiency goals too.

- Some particular types of long running applications can present very specific characteristics such as parallel execution requirements and multi-tier dependencies. In the work presented in this thesis, support for this kind of requirements is only provided implicitly. An explicit support of such a set of execution properties could be really beneficial in some heterogeneous execution environments that could combine commercial batch workloads with scientific and distributed jobs, alongside with transactional applications.

Bibliography

[1] Advanced Micro Devices (AMD). *AMD Virtualization (AMD-V) technology http://www.amd.com/virtualization*. 2.3.2

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, February 1993. 5.3.2.3

[3] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *Fifth Annual IEEE International Workshop on Workload Characterization (WWC-5)*, 2002. 4.5

[4] 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. *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 3–13, 25 Nov. 2002. 2.2.1, 3.4.3

[5] Apache Software Foundation. *Apache Lucene http://lucene.apache.org/*. 5.7.2

[6] Apache Software Foundation. *Apache Tomcat http://tomcat.apache.org*. 2.1.3, 3.3.3, 3.3.5, 1, 4.3.2

[7] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano-sla based management of a computing utility. *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 855–868, 2001. 5.1

[8] U. Balli and J. S. Anderson. Utility accrual real-time scheduling under variable cost functions. *IEEE Trans. Comput.*, 56(3):385–401, 2007. 5.8

[9] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998. 2.2.1

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM. 2.3.2, 5.5

[11] Basic Local Alignment Search Tool (BLAST). *The National Center for Biotechnology Information (NCBI) http://www.ncbi.nlm.nih.gov/blast*. 5.7.2

[12] V. Beltran, D. Carrera, J. Guitart, J. Torres, and E. Ayguadé. A hybrid web server architecture for secure e-business web applications. In *Proceedings of the 1st International Conference on High Performance Computing and Communications (HPCC'05)*, 2005. 1.1.2, 4.4.2, 4.6

[13] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Evaluating the scalability of java event-driven web servers. In *2004 International Conference on Parallel Processing (ICPP'04)*, pages 134–142, 2004. 1.1.2, 4.2.2, 4.4.1, 4.6

[14] V. Beltran, J. Torres, and E. Ayguade. Understanding tuning complexity in multithreaded and hybrid web servers. In *22nd International Parallel and Distributed Symposium (IPDPS'08)*, 2008. 1.1.2, 4.6

[15] Bill Foote. *Heap Analysis Tool (HAT). https://hat.dev.java.net/*. 3.5

[16] K. Birman, R. van Renesse, and W. Vogels. Navigating in the storm: using astrolabe for distributed self-configuration, monitoring and adaptation. *Autonomic Computing Workshop, 2003*, pages 4–13, 25 June 2003. 3.5

[17] Borland Software Corporation. *OptimizeIt Enterprise Suite. http://techpubs.borland.com/optimizeit/index.html*. 3.1, 3.5

[18] Borland Software Corporation. *OptimizeIt Server Trace. http://www.borland.com.tr/tr/products/servertrace_alm/index.html*. 3.5

[19] D. Carrera, V. Beltran, J. Torres, and E. Ayguade. A hybrid web server architecture for e-commerce applications. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, 2005. 1.1.2, 4.6

[20] D. Carrera, V. Beltran, J. Torres, and E. Ayguade. A hybrid connector for efficient web servers. *nternational Journal of High Performance Computing and Networking (IJHPCN)*, 5(5/6), 2007. 1.1.2, 4.6

[21] D. Carrera, D. García, J. Torres, E. Ayguadé, and J. Labarta. Was control center: An autonomic performance-triggered tracing environment for websphere. In *Proceedings of 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05)*, 2005. 1.1.1, 3.6

[22] D. Carrera, J. Guitart, V. Beltran, J. Torres, and E. Ayguadé. *Performance Impact of the Grid Middleware*, chapter Engineering the Grid: status and perspectives, pages 571–585. American Scientific Publishers, 2006. 1.1.1, 3.1, 3.6

[23] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, and J. Labarta. Complete instrumentation requirements for performance analysis of web based technologies. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2003. 1.1.1, 3.1, 3.6, 6.1.1

[24] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. Managing SLAs of heterogeneous workloads using dynamic application placement. In *7th IEEE Symposium on High Performance Distributed Computing (HPDC 2008)*, Boston, MA, 2008. An extended version is available as Technical Report RC24469, IBM Research, Jan. 2008. 1.1.3, 5.9

[25] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. Utility-based placement of dynamic web applications with fairness goals. In *11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008)*, Salvador Bahia, Brazil, 2008. 1.1.3, 5.8, 5.9

[26] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. *SIGPLAN Not.*, 37(11):246–261, 2002. 3.1, 4.5

[27] H. Chan and B. Arnold. A policy based system to incorporate self-managing behaviors in applications. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 94–95, New York, NY, USA, 2003. ACM. 3.5

[28] 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*, 2:516–524 vol.2, 2002. 4.2.2

[29] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on*

Parallel and distributed tools, pages 48–59, New York, NY, USA, 1998. ACM. 3.5

[30] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 30, Washington, DC, USA, 2002. IEEE Computer Society. 5.8

[31] C. Coarfa, P. Druschel, and D. S. Wallach. Performance analysis of tls web servers. *ACM Trans. Comput. Syst.*, 24(1):39–69, 2006. 4.5

[32] G. Czajkowski, M. Wegiel, L. Daynes, K. Palacz, M. Jordan, G. Skinner, and C. Bryce. Resource management for clusters of virtual machines. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1*, pages 382–389, Washington, DC, USA, 2005. IEEE Computer Society. 5.8

[33] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, August 1998. 5.3.2.3

[34] D. M. David Vengerov, Lykomidis Mastroleon and N. Bambos. Adaptive data-aware utility-based scheduling in resource-constrained systems. Sun Technical Report TR-2007-164, Sun Microsystems, April 2007. 5.8

[35] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003. 3.5

[36] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999. 4.3.1

[37] 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*, pages 276–286, New York, NY, USA, 2004. ACM. 5.2

[38] European Center for Parallelism of Barcelona (CEPBA). *Paraver http://www.cepba.upc.es/paraver/*. 1.1.1, 3.1, 3.2.2, 3.5, 3.6, 6.1.1

[39] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling – a status report. In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–16, 2004. 5.1, 5.8

[40] S. Feizabadi and G. Back. Automatic memory management in utility accrual scheduling environments. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 11–19, Washington, DC, USA, 2006. IEEE Computer Society. 5.8

[41] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, Hypertext Transfer Protocol – HTTP/1.1, 1999. 2.1, 2.2.2

[42] I. Foster, T. Freeman, K. Keahy, D. Scheftner, B. Sotomayer, and X. Zhang. Virtual clusters for grid communities. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 513–520, Washington, DC, USA, 2006. IEEE Computer Society. 5.8

[43] A. Goldberg, R. Buff, and A. Schmitt. Secure web server performance dramatically improved by caching ssl session keys. In *Workshop on Internet Server Performance, held in conjunction with SIGMETRICS'98*, 1998. 4.5

[44] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Characterizing secure dynamic web applications scalability. In *19th International Parallel and Distributed Symposium (IPDPS'05)*, 2005. 1.1.1, 3.1, 3.6, 4.3.1

[45] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. Designing an overload control strategy for secure e-commerce applications. *Comput. Networks*, 51(15):4492–4510, 2007. 1.1.1, 1.1.2, 3.6, 4.3.3.2, 4.6

[46] J. Guitart, D. Carrera, J. Torres, E. Ayguadé, and J. Labarta. Tuning dynamic web applications using fine-grain analysis. In *Proceedings of 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05)*, 2005. 1.1.1, 3.1, 3.6

[47] I. F. Haddad. Open-source web servers: performance on carrier-class linux platform. *Linux Journal*, 2001(91):1, 2001. 4.5

[48] Y. Hamadi. Continuous resources allocation in internet data centers. *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, 1:566–573 Vol. 1, 9-12 May 2005. 5.1

[49] J. Hardwick, E. Papaefstathiou, and D. Guimbellot. Modeling the performance of e-commerce sites. In *Proceedings of the 27th International Conference of the Computer Measurement Group*, 2001. 3.5

[50] S. Harizopoulos and A. Ailamaki. Affinity scheduling in staged server architectures. Technical Report CMU-CS-02-113, Carnegie Mellon University, 2002. 4.5

[51] J. Hu, I. Pyarali, and D. Schmidt. Applying the proactor pattern to high-performance web servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems. IASTED*, October 1998. 4.5

[52] J. Hu and D. Schmidt. *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. John Wiley & Sons, Inc., 2000. 4.5

[53] IBM Corporation. *WebSphere Application Server http://www.ibm.com/websphere*. 3.2.3, 3.4

[54] ImageMagick (TM). *ImageMagick http://www.imagemagick.org*. 5.7.2

[55] Intel. *Intel Virtualization Technology http://www.intel.com/technology/platform-technology/virtualization*. 2.3.2

[56] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *IEEE Real-Time Systems Symposium*, pages 112–122, 1985. 5.8

[57] K. Kant and R. Iyer. Architectural impact of secure socket layer on internet servers. In *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, page 7, Washington, DC, USA, 2000. IEEE Computer Society. 4.5

[58] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 595–604, New York, NY, USA, 2006. ACM. 5.2, 5.6.1, 5.6.1.2, 5.6.1.3, 5.8

[59] I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. A. Konstan, D. J. Lilja, and P.-C. Yew. Javiz: a client/server java profiling tool. *IBM Syst. J.*, 39(1):96–117, 2000. 3.5

[60] A. Keller and H. Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manage.*, 11(1):57–81, 2003. 3.1

[61] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004. 5.8

[62] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. 3.5

[63] D. J. Kerbyson, J. S. Harper, E. Papaefstathiou, D. V. Wilcox, and G. R. Nudd. Use of performance technology for the management of distributed systems. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 149–159, London, UK, 2000. Springer-Verlag. 3.5

[64] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373–381, 0-0 0. 5.8

[65] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic application placement under service and memory constraints. In *International Workshop on Efficient and Experimental Algorithms*, Santorini Island, Greece, May 2005. 5.8

[66] S. Kounev and A. Buchmann. Performance modelling of distributed e-business applications using queuing petri nets. In *ISPASS '03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 143–155, Washington, DC, USA, 2003. IEEE Computer Society. 3.5

[67] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance (extended abstract). In *LCTES/OM*, pages 182–187, 2001. 4.5

[68] C. B. Lee and A. Snavely. On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions. *Int. J. High Perform. Comput. Appl.*, 20(4):495–506, 2006. 5.8

[69] C. B. Lee and A. E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 107–116, New York, NY, USA, 2007. ACM. 5.8

[70] C. Li, G. Peng, K. Gopalan, and T. Chiueh. Performance guarantees for cluster-based internet services. *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 276–283, 12-15 May 2003. 5.1, 5.2

[71] B. Lin and P. A. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 8, Washington, DC, USA, 2005. IEEE Computer Society. 5.2, 5.8

[72] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen. Designing a test suite for empirically-based middleware performance prediction. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 123–130, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc. 3.5

[73] A. Mos and J. Murphy. Performance management in component-oriented systems using a model driven architecture" approach. In *EDOC '02: Proceedings of the Sixth International ENTERPRISE DISTRIBUTED OBJECT COMPUTING Conference (EDOC'02)*, page 227, Washington, DC, USA, 2002. IEEE Computer Society. 3.5

[74] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998. 2.2.2

[75] MySQL AB. *MySQL Server http://www.mysql.com/*. 4.3.2

[76] S. Nanda and T. Chiueh. A survey of virtualization technologies. Technical Report TR-179, Stony Brook University, Feb. 2005. 5.5

[77] R. Nou, J. Guitart, D. Carrera, and J. Torres. Experiences with simulations - a light and fast model for secure web applications. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 177–186, Washington, DC, USA, 2006. IEEE Computer Society. 1.1.1, 3.1, 3.6

[78] K. O'Hair. *HPROF: A Heap/CPU Profiling Tool in J2SE 5.0. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html*. 3.1, 3.5

[79] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef. Managing the response time for multi-tiered web applications. Technical Report RC 23651, IBM, 2005. 5.2, 5.4.1.1, 5.4.1.1, 5.4.1.1

[80] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. Dynamic estimation of cpu demand of web traffic. In *valuetools '06: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, page 26, New York, NY, USA, 2006. ACM. 5.1, 5.2

[81] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster-based web services. *Selected Areas in Communications, IEEE Journal on*, 23(12):2333–2343, Dec. 2005. 5.1, 5.2

[82] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 289–302, New York, NY, USA, 2007. ACM. 5.8

[83] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999. 4.2.2, 4.5

[84] W. D. Pauw and J. M. Vlissides. Visualizing object-oriented programs with jinsight. In *ECOOP '98: Workshop ion on Object-Oriented Technology*, pages 541–542, London, UK, 1998. Springer-Verlag. 3.5

[85] Persistence of Vision Pty. Ltd. *Persistence of Vision (TM) Raytracer http://www.povray.org*. 5.7.2

[86] Quest Software. *JProbe http://www.quest.com/jprobe/*. 3.1, 3.5

[87] Quest Software. *Performance Management Suite for Java and Portals*. http://www.quest.com/performance-management/. 3.5

[88] R. Nou, F. Julià, D. Carrera, K. Hogan, J. Labarta, J. Torres. Monitoring and analysis framework for grid middlewares. In *Proceedings of 15th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'07). An extended version is available as Technical Report UPC-DAC-RR-2006-33, Computer Architecture Department. Technical University of Catalonia (UPC). Spain. 2006.*, 2007. 1.1.1, 3.1, 3.6

[89] S. Rangaswamy, R. Willenborg, and W. Qiao. *Writing a Performance Monitoring Tool Using WebSphere Application Server's Performance Monitoring Infrastructure API*. 3.4.1

[90] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 55–60, Washington, DC, USA, 2005. IEEE Computer Society. 5.8

[91] E. Rescorla. Rfc 2818, HTTP Over TLS, 1999. 2.2.2

[92] RFC791. Internet protocol, September 1981. DARPA Internet Program Protocol Specification. 2.1

[93] RFC793. Transmission control protocol, September 1981. DARPA Internet Program Protocol Specification. 2.1

[94] S. Srinivasan. A thread of one's own. In *Workshop on New Horizons in Compilers*, 2006. 4.5

[95] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server virtualization in autonomic management of heterogeneous workloads. In *10th IEEE/IFIP Symposium on Integrated Management (IM 2007)*, Munich, Germany, 2007. 1.1.3, 5.9

[96] Sun Microsystems. Behavior of mixed workloads consolidated using Solaris Resource Manager software. Technical report, Sun Microsystems, May 2005. 5.8

[97] Sun Microsystems, Inc. *GlassFish http://glassfish.java.net*. 1

[98] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition (J2EE) http://java.sun.com/j2ee*. 2.1.3, 3.5

[99] Sun Microsystems, Inc. *Java 2 Platform, Standard Edition (J2SE). http://java.sun.com/j2se*. 2.1.3, 3.5

[100] Sun Microsystems, Inc. *Java Native Interface. http://java.sun.com/j2se/1.4.2/docs/guide/jni*. 3.1, 3.3.3

[101] Sun Microsystems, Inc. *Java Servlets technology. http://java.sun.com/products/servlet*. 2.1.3

[102] Sun Microsystems, Inc. *JavaServer Pages Technology. http://java.sun.com/products/jsp*. 2.1.3

[103] Sun Microsystems, Inc. *JVM Tool Interface (JVM TI). http://java.sun.com/j2se/1.5.0/docs/guide/jvmti*. 3.3.2

[104] Sun Microsystems, Inc. *PerfAnal: A Performance Analysis Tool. http://java.sun.com/developer/technicalArticles/Programming/perfanal/*. 3.5

[105] Sun Microsystems, Inc. *Remote Method Invocation (RMI) Home*. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp. 3.5

[106] Sun Microsystems, Inc. *Sun Java System Application Server 9.1 http://sun.com/appserver* . 1

[107] A. Sundararaj, M. Sanghi, J. Lange, and P. Dinda. Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments. *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 291–292, 13-16 June 2006. 5.8

[108] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 331–340, New York, NY, USA, 2007. ACM. 5.3.2

[109] M. Trofin. A self-optimizing application server design for enterprise java beans applications. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 396–397, New York, NY, USA, 2003. ACM Press. 3.5

[110] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002. 5.8

[111] J. Viega and J. Voas. Can aspect-oriented programming lead to more reliable software? *IEEE Softw.*, 17(6):19–21, 2000. 3.2.3

[112] D. Viswanathan and S. Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 39(1):82–95, 2000. 3.1, 3.3.2, 3.5

[113] W3C - The World Wide Web Consortium. Html 4.01 specification, 1999. 2.1.2

[114] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Wang. Appliance-based autonomic provisioning framework for virtualized outsourcing data center. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 29, Washington, DC, USA, 2007. IEEE Computer Society. 5.3, 5.8

[115] Z. Wang, X. Zhu, P. Padala, and S. Singhal. Capacity and performance overhead in dynamic resource allocation to virtual containers. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 149–158, May 21 2007-Yearly 25 2007. 5.8

[116] WebSphere Extended Deployment
http://www.ibm.com/software/webservers/appserv/extend/. 5.1, 5.2, 5.5

[117] M. Welsh. *NBIO: Nonblocking I/O for Java*
http://www.eecs.harvard.edu/ mdw/proj/java-nbio. 4.5

[118] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001. 4.4.1, 4.5

[119] M. Welsh, S. Gribble, E. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, UC Berkeley, April 2000. 4.5

[120] Wily Technology CA. *CA Wily Introscope.*
http://www.wilytech.com/solutions/products/Introscope.html. 3.5

[121] WMware.
http://www.vmware.com/. 2.3.2

[122] WMware. *VMware DRS*
http://www.vmware.com/products/vi/vc/drs.html. 5.8

[123] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. *Trans. on Embedded Computing Sys.*, 5(3):513–542, 2006. 5.8

[124] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, pages 25–25, 11-15 June 2007. 5.8

[125] K. Yaghmour and M. Dagenais. System administration: The linux trace toolkit. *Linux J.*, page 22, 2000. 4

[126] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Degenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Ottawa Linux Symposium*, 2003. 4

[127] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003. 4.5

[128] X. Zhu, Z. Wang, and S. Singhal. Utility-driven workload management using nested control design. *American Control Conference, 2006*, pages 6 pp.–, 14-16 June 2006. 5.8