Diss. ETH No. 14845

Inverting Middleware: Performance Analysis of Layered Application Codes in High Performance Distributed Computing

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Michela Taufer
Laurea degree in Computer Science - University of Padova - Italy
born April 23, 1971
citizen of Imer (TN), Italy

accepted on the recommendation of
Prof. Dr. Thomas M. Stricker, examiner
Prof. Dr. Daniel A. Reed, co-examiner

2002

”A machine is only useful provided the relevant aspects of its behavior are reasonably well predictable.”
Edsger W. Dijkstra

Abstract

Many important application codes were developed for vector supercomputers and SMP machines. In the last decade, the computing research has moved away from such expensive platforms to cheaper distributed systems such as clusters of PCs. Further on, the migration of these application codes to widely distributed systems, on so called desktop computational grids, looks tempting. Re-engineering applications for such distributed systems while at the same time maintaining good performance remains a challenging task, since we are still lacking tools and instrumentation for performance analysis that work well together with standard middleware. While middleware packages help the application writer to reduce the programming effort, they still cause some loss of control over performance issues which results in suboptimal usage of the machine resources.

To address this problem, we propose a novel method for performance analysis including a framework called inverted middleware. The inverted middleware assists the process of performance engineering by mapping low level performance information, monitored at the operating system layer, back to a higher level, i.e. the application layer. For a distributed system, our inverted middleware framework comprises software instrumentation at the OS level, tools for gathering relevant performance data and an analytical model for performance prediction on distributed systems. The inverted middleware is used side by side with the middleware packages.

We demonstrate the viability of our approach with the performance analysis of the scientific code Dyana for molecular dynamics and a standard OLAP (On-Line Analytical Processing) application, the TPC-D benchmark. With the inverted middleware, we provide a detailed performance characterization of different workloads according to their resource usage, quantify the scalability for new alternative platforms and investigate the impact of the network on the overall application performance. This information helps an expert to effectively re-engineer and successfully migrate application software to new platforms.

Kurzfassung

Viele wichtige Anwendungen wurden für Vektorsupercomputer und Symmetrische Mehrprozessorsysteme (SMPs) entwickelt. In der letzten Dekade hat sich die Forschung weg von solchen kostspieligen Plattformen hin zu preiswerteren, verteilten Systemen wie PC-Clusters bewegt. Vielversprechend sind im weiteren auch Migrationen der Anwendungscodes auf weit verteilte Systeme, sogenannte Desktop-Grids. Es hat sich aber gezeigt, dass die Anwendungen ein grundlegendes Reengineering benötigen, um auf verteilten Systemen betrieben zu werden. Das Erhalten der guten Leistung aber bleibt eine schwierige Aufgabe, da es an Werkzeugen und Instrumentenierungen für die Leistungs-Analyse fehlt, welche gut mit Standardmiddleware zusammen funktionieren. Während Middlewarepakete dem Anwendungsprogrammierer helfen den Programmieraufwand zu verringern, verursachen sie auch einen Verlust an Kontrolle über Leistungsaspekte, was zu suboptimaler Auslastung der Maschinen-Ressourcen führt.

Dieses Problem gehen wir durch einen neuen Ansatz an, nämlich durch eine Methode der Leistungs-Analyse, genannt Inverted Middleware (Umgekehrte Middleware). Unsere Inverted Middleware unterstützt den Prozess der Leistungs-Analyse und -Optimierung, indem sie die auf der untersten Schicht des Betriebssystems gemessenen Leistungs-Informationen zurück auf die höhere Applikationsschicht abbildet. Für ein verteiltes System enthlt die Inverted Middleware Software-Instrumentierung auf dem Betriebssystem-Niveau, Werkzeuge für die Erfassung und Bündelung der relevanten Leistungsdaten sowie ein analytisches Modell für die Leistungs-Vorhersage auf verteilte Systeme. Dabei wird die Inverted Middleware parallel zu den entsprechenden Middlewarepaketen benutzt.

Wir demonstrieren die Vorteile unseres Ansatzes mit der Leistungs-Analyse und -Optimierung eines wissenschaftlichen Codes aus dem Bereich der Molekular-Dynamik, genannt DYANA, und im weiteren einer OLAP-Anwendung (Online Analytical Processing), genauer dem TPC-D Benchmark, aus dem Bereich der Datenbankanwendungen. Die Inverted Middleware dient der detaillierten Leistungscharakterisierung der benutzten Maschinen-Ressourcen bei unterschiedlichen Arbeitsbelastungen, der Quantifizierung der Skalierbarkeit auf neue, alternative Plattformen sowie der Erforschung der Auswirkung des Netzwerks auf die Gesamtleistung der Anwendung. Die daraus entstehende Leistungs-Informationen helfen schliesslich einem Experten, einen Anwendungscode durch effektives Reengineering erfolgreich auf verteilte Systeme zu migrieren.

Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

First of all, I would like to thank my main supervisor, Professor Thomas Stricker. It was a great pleasure and challenge to me to conduct this thesis under his supervision. He provided a exciting and critical atmosphere during the many discussions we had.

I am also grateful to Professor Daniel Reed who agreed on a fairly short notice to be my co-examiner. With his supervision, Professor Reed provided me with precious help and invaluable advice.

My time at the ETH would not have been so enjoyable without my colleagues and my office mates for the past six years. In particular, I would like to thank Roger Karrer with who I shared long discussions about the importance of being an experimental computer scientist and Erwin Oertli who introduced me into the magic world of swiss ice hockey.

I am deeply in debt to Felix Rauch and Christian Kurmann for their help with the several nasty problems with the PC clusters at our Institute and in particular Felix for patiently dealing with my curiosity about LINUX OS. I would also like to thank Christian and Roger for reading carefully through several drafts of my work and Patrik Reali for his contagious enthusiasm and commitment.

In my approach to performance analysis of high performance applications, I tried to maintain the high profile interdisciplinary approach to science by looking at real problems in several application fields including the exciting field of the molecular biology and the challenging field of the distributed databases. Many, many people have helped me not to get lost in this fascinating but very complex world of applications.

I would like to thank the many people who made me discover the fantastic world of the molecular biology. I sincerely thank Martin Billeter, Peter Güntert, Peter Luginbühl and Kurt Wüthrich who created Opal and particularly Peter Güntert for his help and his chemistry advice. I enjoyed working on Dyana with Peter who wrote the sequential code of Dyana and with Gerard Roos who helped me in the migration of Dyana from SMP machines to clusters of PCs. I am also very grateful to Peter Arbenz, Walter Gander, Hans Peter Lüthi, and Urs von Matt, who created Sciddle to parallelize Opal.

Moreover, I would like to thank the database research groups of Professor Schek and Professor Alonso for their precious support when I started my ambitious travel through

the world of distributed databases. In particular I am specially in debt to Roger Weber for his invaluable advice about OLAP. I would also thank Patrik Stähli and Roland Brand for their work on the hardware counter library.

I like to thank Carol Beaty of the SGI/CRI and Bruno Löpfe of the ETH Rechenzentrum who helped with our many questions about the Cray J90 and Cray PVM. I am also very grateful to Nick Nystrom and Sergiu Sanielevici of the Pittsburgh Supercomputer Center who sponsored our parameter extraction runs for the performance prediction of the Cray T3E-900.

I am grateful to Theresa D'Oliveira who patiently went through this thesis parsing my long (too often too long) sentences.

I would like to thank my mother Pia, my sisters Maddalena and Margherita and my brothers Valentino and Leopoldo for their loving support and encouragement all these years.

Last but not least, I would like to thank my partner Andre whose contribution has been immeasurable ranging from reading all my technical writing to keeping high my belief in the research. Andre you made any difficulty over the last six years a small thing to overcome together. Thank you Andre for your love, patience and enthusiasms.

Contents

1 Introduction

1.1 Motivation

1.1.1 Trends in High Performance Computing

The general trend in high performance computing has moved away from expensive supercomputers to cost effective clusters of PCs and will ultimately lead towards computing on widely distributed architectures which are known as computational grids. They are called grids since these distributed computing systems are able to provide large computing power like a tradeable commodity and have high potential for scalability at low cost.

Many applications in high performance computing require larger and larger computation power at a lower and lower cost and therefore they must be migrated from the few supercomputer and SMP machines available to the more common PC clusters and distributed computing systems. However, most applications suffer from severe scalability problems and poor performance once they are moved onto existing distributed systems. In our experience with the performance investigation of several high performance applications on distributed computing systems, we have learned that the reason behind such a loss of performance is often extremely hard to find.

The choice of the most cost effective distributed platforms for an existing application and the re-engineering of an application for a distributed computing system with good scalability are both very challenging tasks. Most of the time, the choice of the platform for distributed high performance computing applications is determined by the most fashionable platform at the time while the solution of performance problems is left as a subordinated job for a “hacker”.

However, we think that both the selection of proper distributed platforms for a given application and the necessary re-engineering of the application to achieve good performance must be supported by better and more accurate performance analysis. Unfortunately, the increasing complexity of the distributed computing systems work against accurate and reliable performance analysis and so the evaluation of computing applications with respect to cost and performance on distributed systems is much harder than on more conventional systems (e.g. supercomputers and SMP machines). Nevertheless, working on several high performance computing codes, we have come to the conclusion

that modern high performance computing applications should be provided not only with a user manual and other proper documentations but also with an accurate characterization of resource usage, software efficiency and application performance.

In particular, to address the problem of migration to new platforms and to deal with the accidental inefficiency of highly distributed executions in high performance computing, a clean method of performance analysis based on performance modeling and instrumentation is valuable if not indispensable. Based on our experience migrating existing high performance applications from vector supercomputers and SMP machines to clusters of commodity PCs, we have come to understand that only a systematic combination of classical performance analysis, -modeling and -engineering provides efficient techniques for making high performance computing applications and future distributed computing systems compatible with each other.

1.1.2 Definition of Performance Analysis

Performance analysis should characterize an application at every stage during its lifespan including during the phases of design, of migration to new platforms and of optimization for a particular platform. We believe that proper performance analysis should comprise:

- the accurate *characterization of the workload* in terms of computational requirements and resource usage. This is required by an application to sort out the reasons behind poor performance and loss of scalability,

- some attempt to make accurate *performance predictions* of application behavior on new distributed platforms as a precondition for cost- and scalability effective migrations. In particular, before we start the migration of an application code to a new platform, we want to know what kind of performance results we might obtain.

Both workload characterization and performance predictions of an application on distributed systems have to involve detailed observations of many performance metrics at different levels of abstraction in the hardware and software system: this means performance metrics at the high level of abstraction of the application as well as more specific performance aspects at the low level of abstraction of the operating system and the hardware. At a high level of abstraction, the user looks at execution time components like the total time of computation or the total time for communication. However, to capture the causes of loss of performance in distributed computer systems or predict the application behavior on new platforms, the observation of resource usage at lower abstracted levels of computation is compulsory. In distributed computing systems, the observation of the resource usage down at the machine level, i.e. the level of the many compute nodes of such a system, cannot be interpreted in isolation but must be considered within a wider context of the performance behavior of the entire computing system, possibly including hundreds of nodes.

Still, accurate workload characterizations and reliable performance predictions provide a more solid basis for architectural decisions. Such a methodology of performance engineering goes hand-in-hand with a systematic, accurate experimental design and with the monitoring of complex computing systems like the distributed computing systems for high performance computing.

1.1.3 The Benefits of Middleware in Distributed Systems

The functionality and the integration of high performance computing applications with distributed computing systems have greatly increased over the past years. This increase in complexity has been enabled by numerous middleware packages solving the most difficult low level programming problems on behalf of the application writer.

Programming at the lowest abstraction level of a large software system remains a cumbersome task and is usually avoided by application developers whenever possible. In modern software systems, *middleware packages (MW)* allow the application writer to work on a problem at a higher level of abstraction. They are called middleware since they are pieces of software located in the middle, in between an operating system and the user-specific applications. Commercial DataBase Management Systems (DBMSs) like ORACLE are popular middleware packages in information processing. For high performance parallel computing, the most popular middleware packages deal with tasking and interprocess communication. MPI and PVM are examples of such middleware layers.

The middleware can effectively hide system-dependent details and permits applications to be migrated more easily from one machine to another. Middleware packages for distributed computing do not only provide portability but also offer increased productivity to application writers, because they can take software development to a higher level of abstraction and relieve the application writer from unnecessary details. The latter functionality is particularly important since the lifespan of a good application program is usually much longer than the lifespan of a computer platform.

As previously stated, in parallel and distributed computing, the middleware packages address the problem of distributing large workloads, in particular, they easily solve the problems of data communication between the distributed entities and some issues of data partitioning, data allocation and load balancing.

1.1.4 The Problems with Middleware in Distributed Systems

Engineering distributed systems for scalability and good performance remains a highly difficult task, since we are lacking tools and instrumentation for performance evaluation that work in distributed systems and with standard middleware packages. While middleware packages, like for example a database management system (DBMS), help the application writer to reduce programming effort, they often cause characteristic loss of control over performance issues and result in inefficient software that makes suboptimal

use of machine resources. Our experience shows that middleware packages can have an extremely negative impact on performance engineering of distributed applications since the performance data provided by the operating system can no longer be directly related to the relevant performance metrics of the application [103].

The fundamental problem with software systems using middleware packages is that the application writer no longer reasons at the system level best-suitable for performance debugging but rather at a higher level of abstraction. The middleware takes care of mapping the high level commands and directives to low level system calls, but when it comes to debugging or performance tuning, most middleware packages are not properly instrumented and therefore do not map the system state or the performance indicators back onto the higher level of abstraction related to their user API. We identify this as the most important hindrance to find performance bottlenecks in systems using middleware. It is the primary reason why many application writers are completely lost once they have to track down bad performance in the application or even find efficiency problems in the code of the middleware itself.

1.2 The Main Challenges Tackled in this Dissertation

1.2.1 Inverted Middleware as a Concept for Performance Analysis

At the beginning of this thesis, we claim that most existing standard middleware packages lack proper instrumentation for performance analysis related to the distribution of the computation and therefore hinder the detection of performance bottlenecks and architectural problems of various kinds (hardware and software).

A number of pioneering efforts have produced many useful monitoring tools which provide global, uniform figures on the performance behavior of complex distributed systems. However, most of this previous scientific work is limited to the successful implementation efforts or to some new successfully implemented algorithms for an application running on a specific target machine. Moreover, through our experience with manual instrumentation of specific middleware packages, we have learned that middleware developers rarely instrument their middleware for backward mapping system states onto user level abstractions.

The aim in this dissertation is not to compete with previous efforts, but rather to describe a general approach to the performance analysis which allows us to regain tight control over the machine resources in distributed computing systems and to point out inefficiencies of the middleware abstraction. We do this on both a conceptual and on an implementation level. Consequently, we propose a novel method for performance analysis, i.e. for workload characterization and for performance prediction, which we will call *inverted middleware (MW^{-1})*. The inverted middleware is actually an elaborate framework comprising architectural concepts, tools and models. Its general aim is to reduce

the complexity and to improve the quality of performance analysis on a wide range of distributed computers running different high performance computing applications on different sort of middleware packages regardless whether the middleware is a black box or open source.

1.2.2 The Decomposability of Distributed Systems

To justify the performance analysis of the complex distributed systems layer by layer, we claim and prove that such systems obey the *theory of near-complete decomposability* according to the criteria stated by Courtois [30]. Using the theory of near-complete decomposability, we can break the complex distributed computing system into several layers. For performance analysis, performance modeling and prediction, the theory allows us to look at the interactions within the layers as if interactions among layers would not exist, or to look at the interactions among layers without considering the interactions within the layers. Using the theory stated by Courtois, we show how the inverted middleware can monitor the critical performance resources without instrumenting the application under investigation or making intrusions into the middleware layer.

We propose that the inverted middleware is developed separately and is located side-by-side with the middleware. This dissertation work emphasizes the symmetries of mapping resource abstraction and reverse mapping resource usage in middleware and inverted middleware respectively. We present the middleware as a software package which maps some functionality from a user level API onto a functionality available at the low level of the operating and communication system. Then, the task of the inverted middleware will be to map the performance relevant system state back onto high level performance information that is in an appropriate form for the user. In this approach, we see the middleware as a process of mapping functionality downwards along a chain of mathematical functions which lead from high level standard APIs to low level system calls of the operating system. Looking at the system in this way, it appears as the most obvious choice to design the inverted middleware as a corresponding chain of inverse mathematical functions, mapping the performance monitoring information about the machine resources gathered at the operating system back onto the abstract world of high level standard APIs.

1.2.3 Completeness and Reliability in Performance Analysis

The design and implementation of the inverted middleware framework lead us to consider some quality criteria for the performance data used to describe the behavior of a system. In our approach to the study of the performance of distributed applications, we stress that the performance data under investigation has to be complete and reliable.

Complete performance data does not only provide information on the total amount of resources consumed, but also when, where and why these resources are required. We address the *completeness* issue in the inverted middleware framework by providing time

and frequency traces of the resource usage in addition to the summary of the total resource usage. So we are able to detect bottlenecks due to peak usage of the resources.

A reliable evaluation of the performance is characterized by performance data which are minimally affected by the monitoring process. Efficient collection of performance data is a precondition for the *reliability* of the performance information of a large distributed system. The performance data has to be sampled and collected making as few monitoring intrusions as possible in the hardware and software system.

Completeness and reliability can be in conflict with each other. Completeness may induce inefficiency and therefore leads to low reliability of the performance data since this data is badly affected by the monitoring process.

In our work, we investigate strategies for keeping the monitoring intrusions as low as possible and design them in the inverted middleware framework. So for keeping the communication intrusions low, we use UDP/IP for the monitoring traffic of the performance data. For reducing scheduling and execution intrusions, we introduce a loose notion of time in the distributed system based on cycle counters used as virtual barriers. Moreover, we leave the choice of the monitoring master allocation free and we do not introduce any unnecessary changes of the monitored application.

1.2.4 Viability of our Method for Performance Analysis

We demonstrate the viability of our performance method by two experimental studies in two different application domains: first, in the domain of advanced molecular biology and second in the domain of distributed databases running OLAP (On-Line Analytical Processing) applications on distributed computing systems.

In particular, in the context of the advanced scientific computations, the question of cost-effective migrations is at most priority. We compare the performance workload characterization and performance prediction using performance tuning by-hand with the benefits of our novel method of automatic performance analysis using the inverted middleware framework for two molecular dynamics codes: Opal and Dyana. The migration of both applications to widely distributed systems are limited by the number of nodes that a master can handle with good scalability of the corresponding computational methods. We use our inverted middleware framework to predict the maximal number of distributed nodes on which the application Dyana can run efficiently without a change of compute paradigms.

In the context of the distributed databases, migration to distributed systems has been attempted by adapting middleware layer functionality to the distributed platforms. However, most of the time, the performance expectation fails and the loss of performance efficiency remains hard to explain. To overcome this problem, we focus our attention on finding performance optimizations by means of the inverted middleware. With the inverted middleware, we are able to provide a detailed characterization of different work-

loads for the chosen applications according to their resource usage, quantify the scalability for alternative platforms and investigate the impact of the network on the overall application performance. We demonstrate the viability of performance analysis with the inverted middleware for performance optimizations for a standard OLAP application, the TPC-D Benchmark.

1.3 Roadmap for the Dissertation

In **Chapter 2**, we discuss the architectural characteristics of parallel and distributed systems looking briefly at clusters of PCs as well as at desktop grid computing architectures. We define a systematic approach for performance analysis and we look at how to build global views of the system behavior for distributed applications in parallel and distributed computing systems. We also discuss the intrusion problem of monitoring processes on distributed applications in distributed systems.

In **Chapter 3**, we look at the benefits and the problems of the commonly used middleware-oriented systems in which some middleware layer serves as a glue between the operating and network systems and the user-specific applications.

In **Chapter 4**, we outline the functionality of the inverted middleware as a method for performance analysis and we introduce the design the general structure of our inverted middleware framework. We also discuss the implementation of a first prototype for an inverted middleware framework to be used on clusters of PCs.

In **Chapter 5**, we briefly explain the concept of the near-complete decomposability theory introduced in the '70s by Courtois. We apply this theory to the distributed systems, such as clusters of PCs, for simplifying the process of performance analysis.

In **Chapter 6**, we start the evaluation of our approach by looking at a conventional method of performance analysis. We study workload characterization and performance prediction of Opal, a typical scientific computing application, doing the performance analysis by-hand (i.e. we work on the application and the middleware layer by introducing barrier synchronization and by engineering the code). We conclude this conceptional chapter of the thesis with a plea for new, more systematic performance tuning methods for performance prediction and analysis on distributed systems.

In **Chapter 7**, we show the benefits of our novel method of performance analysis using the inverted middleware framework together with performance modeling on Dyana as a code representative of molecular dynamics applications. In particular, we establish and calibrate the performance model of Dyana from existing migrations and use it to check the viability of migration to future platforms.

In **Chapter 8**, we investigate optimizations and isolate hardware configuration and query performance problems by means of the inverted middleware framework in the context of OLAP databases.

In the conclusion in **Chapter 9**, we summarize the main contributions of our study

in performance analysis using the inverted middleware for high performance applications on distributed systems and we introduce some further work.

2

Platforms, Applications and Methods for Performance Analysis

In this chapter we address some common issues about parallel and distributed platforms, applications and performance methods in high performance computing. We look at the main challenges in monitoring performance of distributed computing systems as well as briefly examining the state of the art of monitoring tools for scientific computation and database applications.

2.1 Parallel and Distributed Systems in High Performance Computing

2.1.1 Supercomputer Platforms

The supercomputer platforms that we have considered in our study are a Cray J90 at the ETH Zurich and a highly scalable supercomputer Cray T3E ”big iron”at the Pittsburgh Supercomputer Center.

The Cray J90 is a symmetric multiprocessor (SMP) system built from eight processors with efficient sequential and vector processing capacities (10 ns cycle), large memory (2048 MB), high memory bandwidth and efficient I/O capabilities (2 x HIPPI). The J90 is a CMOS-based vector machine using DRAM memory starting at $250,000, but with typical configurations running up to a cost of $1 million [56].

The Cray T3E is a massively parallel, highly scalable system. The Cray T3E multiprocessor is a shared memory system scalable to 2048 processors. Successor of the Cray T3D, the T3E augments the memory interface of the DEC 21164 microprocessor with a large set of explicitly managed, external registers used as the source or target for all remote communication. The external registers provide a highly pipelined interface to global memory. The T3E provides a rich set of atomic memory operations and a flexible, user-level messaging facility as well as a set of virtual hardware barriers that can be arbitrarily

embedded into the 3D torus interconnect [91, 26].

2.1.2 Clusters of PCs

Even the largest massively parallel supercomputers are limited in terms of their maximum performance when used independently, as single nodes. Clustering provides an architecture to go beyond that limit, while providing a better cost-performance ratio [10].

A cluster is a system of multiple computers interconnected via a high-performance local area network (LAN). Nodes in the clusters communicate using standard message-passing interfaces. Shared-memory clusters are less common, but do exist. Tandem introduced a 16-node cluster in 1975 [11]. SGI pioneered large, non-uniform memory access shared memory clusters in the 1990s [10].

In 1994 the Beowulf Project was started with the goal of building low-cost clusters using COTS (Commodity Off The Shelf) base systems [97, 12]. In summer 1994, Thomas Sterling and Don Becker presented a 16-node cluster of Intel 486 DX4 processors connected by channel-bonded Ethernet. The machine, called Beowulf, was very successful and the idea of using COTS machines and widely available software to build clusters spread quickly through the academic and research communities, leading to the Beowulf project as it exists today [12]. Beowulf clusters have brought do-it-yourself cluster computing to many groups where such computing power was out of reach before.

Modern clusters of PCs consist of hierarchical topologies of sub-clusters connected by multiple switches based on Gigabit Ethernet or Myrinet links [85].

2.1.3 Desktop Grids

With millions of PCs installed and networked world wide, a lot of computing resources are highly underutilized. The main idea behind a computational grid is to interconnect and aggregate computing resources over long distances to build a large, virtual supercomputer [41, 3]. Such a virtual supercomputer could eventually solve complex problems that are still too large for a single supercomputer and enable new classes of applications in different domains. The main feature of such a platform is the inherent heterogeneity and location of the resources. Computing power of nodes on a computational grid can range from desktop-class PCs to large supercomputers, providing a wide performance spectrum. Also, a widely varying range of network connectivity technologies are in use, from machines in the same LAN segment as the server (100 Mbit/s switched Ethernet) to machines at home connected via ADSL (from 128 kbit/s to 2Mbit/s) or low-bandwidth analog modem connections (56kbit/s).

Grid computing platforms can become arbitrarily large and complex. Adapting applications to scale well on large numbers of nodes, algorithms for scheduling and effective resource demand (reservation and utilization) are all critical issues in grid computing.

2.2 Applications in High Performance Computing

Applications in high performance computing can be classified as having non-deterministic or deterministic behavior in terms of results or in terms of performance. With regard to results, applications can be characterized as deterministic when, for the same input files and on the same platform they always provide the same results. When this condition is not fulfilled applications are non-deterministic. In the case of performance, applications have deterministic behavior when, for the same input files and on the same platform, the same performance values are provided. If this is not the case, we speak of non-deterministic applications.

We focus our attention on the performance factor to classify the applications considered and their behavior on distributed systems. Most applications on distributed systems are non-deterministic. They can be characterized by different levels of non-deterministic behavior. The main factors responsible for non-deterministic performance are:

- dynamic scheduling of the tasks in which the application run depends on intermediate results (e.g. protein folding using the scientific computation code CHARMM [102]),
- random generators present in the application (e.g. conformer computation using the scientific code Dyana [104]),
- lack of resources like network bandwidth during the application run (e.g. TPC-D Benchmark with disjointly distribution of the data among the nodes of the distributed platform [105]).

The migration of non-deterministic applications to distributed systems requires choosing the proper platform in terms of cost/performance ratio. The configuration of such distributed systems is a challenging task: there are different distributed systems with varying cost and performance features. Moreover, in the last few years, the need to buy large platforms with high computing power has been substituted by the possibility of buying computational cycles on already existing nodes of platforms reachable via the Internet. Therefore the user who is willing to improve the computational power of his application by moving it to distributed systems, is facing a challenging task which might require some research effort. However, an accurate performance analysis can make this research effort much easier.

2.3 Performance Analysis Methods

The study of the performance of a given application on a complex, distributed system should be supported by designing methodologies and techniques for the performance analysis. These must be able to catch the features and characteristics of the system-application interactions in the most manageable and accurate way. The performance study

requires more effort for a high degree of system-application interactions, which are not always easily manageable.

Design methodologies and techniques for performance analysis should include rigorous definitions of performance components and aspects within the system such as:

- the definition of measures of performance (e.g. performance metrics and workloads),
- the definition of a measurement environment,
- and the definition of the evaluation techniques.

The definition of measures and of a measurement environment is directly related to the design of a proper set of experiments for performance analysis. In these experiments parameters which are correlated or have no significant effect on the performance are cut off. A systematic experimental design helps to sort out the relevant and irrelevant effects of the numerous parameters that determine the performance of the applications on distributed platforms [49].

The data gathered locally during the monitoring of experiments have to be collected and combined into a global view. Aspects such as a global notion of time, granularity of the samples and structure of the information are critical issues in distributed system and will be analyzed in detail in Section 2.4.

2.3.1 Measures of the Performance

We attempt to gather the maximum amount of information about the application running on a distributed system with the minimum number of experiments and measures. Therefore the choice of the proper *response variables*, which best describes the application on the platform chosen, is crucial. Such a choice is related to the abstraction of the system resources that is taken into account for the investigation and the measurements. High level measurements, largely possible by re-engineering the code considered, provide the total time for computation, communication and other activities related to the application. Synchronization time (defined as the time for barriers and idle times) is an example of other activities. At a lower level of abstraction, it is possible to measure resource information like floating point operations, amount of data traffic on the network and data to and from the disks.

The kind of application also plays a role in the choice of the response variables. Commonly, in the context of scientific applications, the computation is measured by means of the amount of floating point operations [104], while for database applications the same aspect is investigated by counting the millions of instructions per second or MIPS [105].

2.3.2 Defining a Measurement Environment

The parameters that have a significant effect on the response variables are called *factors*. A quantification of their effect on the response variables is essential for a successful performance analysis. Factors can take a set of values. These values are called *levels*. We propose a classification of the factors into two different sets according to their dependency:

1. *Platform factors*: factors related to the characteristics of the platform components such as CPU speed and network speed.

2. *Application factors*: all the factors related to the application. In scientific computation applications like Opal, the size of the molecular complex in question or a cut-off parameter for reducing the amount of interaction among atoms, are typical examples of application factors [103]. Whereas, in distributed database applications like the TPC-D Benchmark, the rate of replication and fragmentation of tables are considered as application factors [105].

Keeping to a limited set of levels is strongly recommended, but a restricted choice of representative values should include both ends of the performance range for the factors. A *full factorial design* utilizes all the combinations of factors at all levels. Maintaining a clear overview of the distributed system using a full factorial design of the experiments, becomes difficult when the performance study is conducted with too many levels as this results in an oversized set of experiments for each factor considered. Consequently, a *fractional factorial design*, with only a reduced set of levels, is preferred.

2.3.3 Evaluation Techniques

Measurement, analytical models and simulation are the three most common performance evaluation techniques. We use them simultaneously. In particular, we can use analytical models to properly describe the application and its interaction with the complex, distributed system. The resource-usage is captured in analytical models as a function of the most relevant factors and response variables. An error can be introduced as well.

$$time_{application} = f(resource\_usage) = f(factors, responce\_variables) + err \qquad (2.1)$$

The validation of the analytical model is commonly done through measurement and simulation.

2.4 Challenges in Monitoring Performance of Distributed Systems

For the sake of higher execution speed of real applications that use multiple processors and distribute a large workload, proper engineering of the whole system for performance-

and scalability analysis becomes a crucial issue. The monitored applications should not be restricted to a specific monitoring tool, a specific programming model or computing architecture. On the other hand, the instrumentation for the performance monitoring should be inserted transparently and automatically, should run on different kinds of machines and not for a specific application or code. These aspects make the monitoring of application performance in distributed systems a challenging task.

2.4.1 Closing the Loop between Performance Data Collection, Analysis and Optimization

For closing the loop between performance data collection, analysis and optimization in distributed systems, it is crucial to locate the source of performance bottlenecks which affect the functionality, capability and scalability of large-scale parallel and distributed applications. Beside the problem of finding bottlenecks in distributed systems, another relevant issue is the search for techniques which allow the application under investigation to be responsive to the available resources or to allow its tuning at run time.

Finding bottlenecks in distributed systems

Application users are interested in sorting out the fundamental bottlenecks behind any poor application performance. An effective performance analysis should be able to find the reasons behind such a poor performance. Most of the time the performance analysis takes place through an interactive process and only a small set of possible bottlenecks have to be considered. Once the user has identified the bottlenecks responsible for the slowdown in performance, they have to identify the specific resource requirements related to the bottlenecks and exploit the application execution phases whenever these bottlenecks occur. This means testing the application's execution during different intervals of time. The proper definition of the time intervals can affect the accuracy and the volume of the data collected.

A precise understanding of resource utilization is not only required for fine tuning a running system, but also for the prediction of scalability or the study of viability of the application on new, alternative platforms such as, for example, clusters of low cost commodity PCs and desktop grid platforms.

The identification of bottlenecks requires the collection of detailed and accurate information from the application hosts. The collection can take place either by instrumenting the application or by instrumenting the system. The instrumentation of the application can be inserted at different phases of the application lifespan. First, the instrumentation can be inserted manually by the user into the source code. Second, it can be put automatically into the application by the compiler. Third, the collection instrumentation can be inserted by linking an instrumented library. Last but not least, the instrumentation can be inserted by modifying the linked executable. All the techniques cited above require that

the insertion of the instrumentation takes place before the application execution. Such an instrumentation remains fixed during the application execution. An other approach to performance data collection, alternative to the application instrumentation, consists of some system instrumentation in which the data are collected via dedicated monitoring processes or via the operating system. This approach requires the modification of the OS to collect traces of events and to forward the performance information to the user. Such an approach does not require efforts to instrument the application under investigation, but on the other hand, requires the instrumentation of the application hosts is required. Any change of the kernel of the OS might result in a time consuming modification process which has to be carefully considered together with the implications for the portability and the robustness of the modified OS.

The collection of performance data might introduce additional bottlenecks. During the collection, the minimization of any perturbation of the monitored application is compulsory. In other words, the collected data should represent the behavior of the application as if it were not monitored.

Several monitoring and performance tools provide performance graphs, metrics and tables which show the application behavior on the distributed systems. Most of the time the amount of information is so large that a performance expert is required to understand and interpret this data. The large amount of performance data available can be reduced by filtering the data and by storing only the interesting events. The control of the events recognized and the filtering of the performance information can take place statically or dynamically.

The problem of finding bottlenecks in distributed systems for large-scale parallel and distributed applications is accurately addressed by Hollingsworth in his Ph.D. work [58]. Hollingsworth introduces the W^3 Search Method to automatize the search for performance bottlenecks and a new model for data collection, called Dynamic Instrumentation, which permits instrumentation collection to be made and changed during execution. The W^3 Search Method addresses the three main questions:

- why is an application running so slowly?
- where are the bottlenecks responsible for the slow-down in performance?
- when does the problem occur?

Compared with traditional data collection, the Dynamic Instrumentation addresses the selection of what data to collect when the program is running. The W^3 Search Method and the Dynamic Instrumentation form an integrated system for dynamic on-the-fly selection of performance data to collect, combining decision support to assist users with the selection and presentation of the appropriate data for an application under investigation.

Addressing automatic adaptation

Post-mortem performance optimizations are not effective on widely distributed systems since most of the time applications running on such systems are characterized by behaviors which are non-repeatable. The optimization of applications and the adaptation of their behavior to the dynamically-changing resources on these systems is a challenging task.

Computational steering provides techniques to the user to alter the behavior of the given application at run-time e.g. by interacting with the data- and code structures in the code through hooks which allow the application's execution to be changed (an example is the Active Harmony system [59]), through subroutine calls added as instrumentation (e.g. by an automatic tool such as SvPablo [33, 86]) or by allowing the application semantics to be changed. Steering techniques include modifying program state, managing data output, starting and stalling program execution, altering resource allocations etc.

To program the environment for computational steering, a hierarchy of adaptation options are possible in which the user is called to choose the levels of adaptation and the roles for adaptation. While adaptation tools like the Active Harmony system [59] optimize the resource allocation by looking at multiple libraries and applications in a global frame, other tools like ATLAS [113] look at specific kind of libraries and applications (ATLAS considers linear algebra libraries) or, like AppLeS [13] adapt each application or library independently. Even the layer at which the adaptation takes place can chance. In the tools listed above the adaptation takes mainly place at the middleware and application layers. Tools like Autopilot [87] focus on resource adaptation and management at the operating system layer.

Mainly at the base of an adaptation there is the need for the recognition of changing conditions both at the level of the distributed application and the underlying distributed system. The steering adaptation can be manual and then the steering process requires the user to monitor program or system state and have the ability to make changes to the program themselves. But it can also be automatic, and then an interpreter automatically translates adaptation from a manager or a sensor to the executing system without any intervention of the user.

An interesting example of steering adaptation is addressed by Vraalsen and others in [112]. The approach proposed in [112] allows to predict performance and to detect unexpected execution behaviors by using monitoring to find performance execution violations. However, this technique delegates to the user or an external agent the possible optimizations. Automatic adaptations require complex algorithms to handle the selection of application or program parameters. Only a few projects exist which are studying how to address the automatic adaptation of real applications to the resources on computational grids. One of these projects is the GrADS project which aims to design an execution framework for adaptive applications on the grid. The design of the project together with

the motivations behind it are illustrated in [68]. At the moment, only a preliminary prototype of the framework for demonstrative purposes is available.

2.4.2 Building a Global View of Distributed Systems' Behavior

Performance analysis in distributed systems means monitoring the behavior of a distributed, heterogeneous environment. Therefore finding either performance predictions or workload characterizations go hand-in-hand with the study of the distributed environment and the observation of the key characteristics of complex distributed systems.

Building global performance views of distributed applications running on concrete distributed systems is a task which requires much effort. Many aspects and critical issues have to be taken into account. First of all, a large amount of nodes and therefore a large amount of performance data have to be monitored. This performance data has to be moved within the distributed system. An exhaustive representation of the performance might become very expensive. One solution could be the reduction of the amount of information about the system by using sampling of performance data, however that implies to move from a deterministic to a probabilistic environment.

A distributed system is characterized by heterogenous components such as CPUs with different clock rates and different network connections with different bandwidth and latency. A proper definition of the different machine-resources and the scheduling of the resource demands (i.e. sequence and amounts in which the different resources are required and used) is needed.

Further critical aspects related to the sampling of performance information is the kind of information to be monitored and collected (i.e. frequency traces and data traces) as well as the granularity of the sampling.

The mechanisms for sampling can comprise different sample settings from a master-slave setting to an all-to-all setting. In a master-slave setting, the global performance view is available to a master which receives the information and can take decisions about what to do with it. On the other hand, in an all-to-all setting, the global performance view is made available to each single node of the system.

The collection of the local sampling can take place by either putting each single performance shot or a set of shots into network packets. During the re-building of global view of the local data collected from distributed, heterogenous nodes it has to be considered how to recover lost data during the collection phase as well.

Last but not least, the cost of monitoring, collecting and recovery of performance data has to be taken into account since it can seriously affect the monitored application. Accurate monitoring can affect the reliability of the information collected (aspects related to these issues will be studied in detail in the next section).

2.4.3 Reducing the Effect of Performance Monitoring

Notion of monitoring intrusions

Monitoring tools which provide accurate performance information during the execution of an application, measure not only the amount consumed and by how many resources, but also when and where these resources were required. Accurate gathering of performance data can negatively affect the reliability of the information because monitoring tools may intrude on the monitored processes thus competing with the application for the machine-resources and in particular for the communication subsystem.

In order to be reliable, the performance information should not be affected by side-effects such as the monitoring process. Therefore the least system disturbance due to the monitoring processes is crucial to maintaining reliable performance information. Minimizing system disturbance due to the monitoring processes and maintaining the amount of disturbance on large distributed systems lower than the 10% of the total run time is desirable for maintaining reliable performance information.

The different kinds of intrusion of the monitoring tools are known as:

- communication intrusion[117],
- scheduling intrusion [116] and
- execution intrusion [116].

Communication intrusions take place when the order and timing of message transmission and delivery in the network is altered because of monitoring data traffic. Scheduling intrusions and execution intrusions take place when the decisions of process scheduling and the outcome of the monitored processes is altered. The communication overload due to re-transmission and delivery of packets and maintaining a global notion of time by the monitoring tools in the distributed system can cause alterations affecting the content of the messages as well as the execution of the monitored processes.

Common techniques for reducing intrusions

Possible approaches to reduce the monitoring intrusions are hardware- [109], and software supports [77, 44] and program simulations [24, 64]. The solution presented in [109] relies on additional equipped hardware which limits the choice of applications that can be monitored. The techniques which refer to the software approaches presented in [77, 44] consist of software techniques for the real-time estimation of the application computation by means of compensation of the intrusion induced effects. Unfortunately, these kind of approaches do not properly suit the monitoring of high performance applications. Program simulation techniques do not need runtime monitoring.

The approach presented in [24] requires the prediction of future executions of the application, while the approach in [64] needs hardware support. A clean distributed system approach tries to reduce the monitoring intrusions by re-engineering the underlying communication system (e.g. the socket library), to restore the original order of messages and to remove the service communication entirely from the performance picture of the monitored application [115]. This approach is limited to a fully sequential token-ring system or requires a fully connected point-to-point interconnection.

2.5 State of the Art of Monitoring Tools

2.5.1 Common Monitoring Tools for Scientific Computation Applications

Many existing performance analysis tools can be used in conjunction with middleware packages for scientific computation. Some work targets the scalability based on one aspect of the system such as scheduling [100] or load balancing for heterogeneous clusters of PCs [18, 79].

The classical approach to performance engineering attempts to develop tools, including some visualization tools, for the analysis of software and hardware performance behavior. Some work on performance in embedded systems provides only functional diagrams of systems with performance annotations [99]. The *Pablo/SvPablo* framework provides a programming language and an architecture independent system for performance analysis and visualization [33, 86]. Additional instrumentation with embedded calls to trace capture libraries is introduced into the source code of the application. Dynamic statistical projection pursuit is used in the Pablo environment to provide a filter for identifying interesting performance metrics, to reduce monitoring perturbation and to limit the data volume [110].

A different solution collects event traces through instrumentation incorporated into the middleware or the operating system [63]. *ParaGraph* uses trace files collected by a Portable Instrumented Communication Library [55, 54]. *Xab* collects event traces by instrumenting calls to the PVM [9]. *Conch* is based on PVM and is a network computing system that includes build-in instrumentation for collecting trace data [108].

VAMPIR [80, 21] is a graphical tool that can be used to analyze the trace files produced by a program. VAMPIR relies on the VAMPIRtrace library. VAMPIRtrace is linked to a program together with the MPI library, so that, when the program runs, a trace file containing the profiling data is produced. VAMPIR runs on distributed memory systems using message passing programming models like MPI in which the parallelism is implicitly designed while the data movement is explicitely designed. On shared memory systems using middleware packages like OpenMP, which are designed for implicit data movement and explicit parallelism, other monitoring tools are commonly used. A well-established tool for performance monitoring on OpenMP is *GuideView* [66].

VGV [57] is a performance visualizer for data analysis and event trace analysis on multi-level communication and memory hierarchical systems. This performance tool combines VAMPIR and GuideView respectively for the distributed memory MPI environment and for the monitoring of shared-memory environment OpenMP located on top of MPI. The hybrid programming paradigm MPI + OpenMP is implemented and monitored in [57] using the SWEEP3D program as an application.

2.5.2 Common Monitoring Tools for Database Applications

Performance analysis and monitoring tools are crucial to locate and remove inefficiencies due to parallel execution in distributed databases. There are standard approaches that allow the performance analysis by common performance tools. These tools are commonly provided by the database middleware packages or are developed by the user through the instrumentation of the application.

Many application writers in the database field recognize that hiding the complexities of the network, database, application processor, and operating system behind an API is not ideal with respect to a good, robust and transparent performance behavior. They propose solution-oriented middleware where the middleware becomes more integrated and is itself the object of investigation [74, 78, 32]. In such cases where the application is tightly bound to the middleware, and where the user is forced to move the focus from the master-slave architecture to the middleware, the attention focuses on resources such as queues, APIs or database slaves. However, in this situation, the target platform (clusters of PCs), its resources and its low level monitoring instruments are, unfortunately not discussed because they focus on issues not usually included in the database research.

2.5.3 Limitations of Existing Monitoring Tools

Most existing monitoring tools lack in generality, introducing restrictive platform or application dependencies. The kind of monitored applications and system architectures significantly affect the design, implementation and use of the monitoring tools which, consequently, are able to provide performance data only for a given programming language or programming paradigm. A combination of different monitoring tools (like for the VGV performance visualizer) is a possible solution to deal with complex system structures in which different computing paradigms are presented (e.g. data and task parallelism). Nevertheless, it is still an open question whether such a combination of tools can result in an effective and general solution in the context of large, decomposable distributed systems.

For most of the existing monitoring tools, the re-engineering of the computing system either at the application or the middleware level is required before the application execution, while aspects such as the effect of the middleware on the performance or the revision of the monitoring decisions at run-time are rarely taken into account. Moreover, performance prediction and performance analysis based on accurate evaluation techniques like

analytical model techniques is not supported in most existing performance tools. Because of the specificity of these tools to a particular system or particular application, any re-modeling for new applications could indeed result into a high time consuming task.

All these issues motivate the need for further investigation and design of new performance methods for distributed systems characterized by a higher system independency, clearness and accuracy. In Chapter 4, we will show how the performance method presented in this thesis cope with exactly these problems and we will compare our resulting novel performance framework for performance analysis with some of the most common monitoring tools.

3

The Middleware Layer

In the following sections, we briefly introduce the evolution of software systems from monolithic designs to middleware-oriented systems. We rigorously define the concept of middleware according to a pure system architecture approach and we point out the main benefits and problems related to the middleware in a distributed computing system.

3.1 The Evolution towards Middleware-Oriented Systems

The functionality and the integration of application programs has greatly increased over the past 10 years. In the early days of computing, most solutions were equipped with monolithic software designs that are specific for a problem domain. APIs (Application Programming Interfaces) were not yet existent for many common functions and standard communication protocols were not used. The applications had to be adapted again and again to run on a different system. Entire systems (hardware/software) could only be designed and programmed by a single engineer unless a big software design project with a large overhead was started. The problem was that programming at a low abstraction level was and remains a cumbersome task. Many platform and programming system specific details have to be taken care of, e.g. storage management, exception handling, proper initialization of service routines and networking.

With a rapidly growing scope of computing solutions today, even the simplest end-user applications comprise thousands of lines of code because of the rapidly increasing functionality and the growing integration of the application programs. Modern highly networked systems on which the applications run are characterized by heterogeneity and by increasing complexity related to the distribution of the computation and the data. To solve these problems, vendors provide their systems with general purpose services which have standard programming interfaces and use standard protocols. The software systems providing these services are called *middleware* because they are located in a layer in the middle, in between a processor architecture, an operating system on a network software system and the user-specific applications. The management of handling presentation, computation, information storage, communication, control and system resources are all highly popular examples of middleware services [14].

Typically, middleware packages take care of the details and offer some functionality at a higher level of abstraction. The oldest pieces of middleware packages in the history of computing were possibly the scientific libraries written in Fortran. For example, LINPACK [38] raises the level of abstraction from floating point numbers and basic arithmetic operations to matrices and their operations.

Most middleware-oriented systems impose specific structure formats on the applications built on them. Their API software layers are responsible for inter-operability and are well defined [114]. Moving from monolithic systems toward middleware-oriented component-based systems has solved many problems and shown many advantages and strengths. However, it has also introduced some new challenges and weaknesses when it comes to questions of performance and efficiency.

3.2 Definition of Middleware Layer

In our following discussions, a middleware layer (MW) is defined as a set of middleware components providing middleware services. The layer is built on top of a structure-aware backend (i.e. operating systems and network software systems) and provides different structural abstractions to the user-specific application located on the top of the middleware.

The typical architecture for distributed high performance computing defines three layers of functionality within the entire system (see Figure 3.1). The lowest layer comprises the operating system on top of the distributed hardware (e.g. clusters of PCs). The middleware layer comprises a set of components providing different levels and kinds of abstraction of the lowest hardware resources and their services to the application. This layer maps the user-desired functions onto the low level system functions available. Most middleware components are generic and can run on multiple platforms. Middleware services are often widely distributed and offer standard interfaces to common protocols [14]. The higher middleware layer offers different user-specific services to applications. On distributed systems, most parts of the middleware are distributed by themselves, but successful middleware packages manage to offer a coherent view of a single system to the application programmer. These kind of systems are called middleware-oriented component-based systems. Standard API protocols are responsible for inter-operability between layers and between components within the layers.

3.3 Middleware Layers in Distributed Computing

In distributed computing, many middleware layers address the problem of distribution. In particular, they solve the problems of data communication between the distributed entities, the problem of data partitioning, data allocation and load balancing. A well-known example of a distributed middleware layer is CORBA (Common Object Request Broker

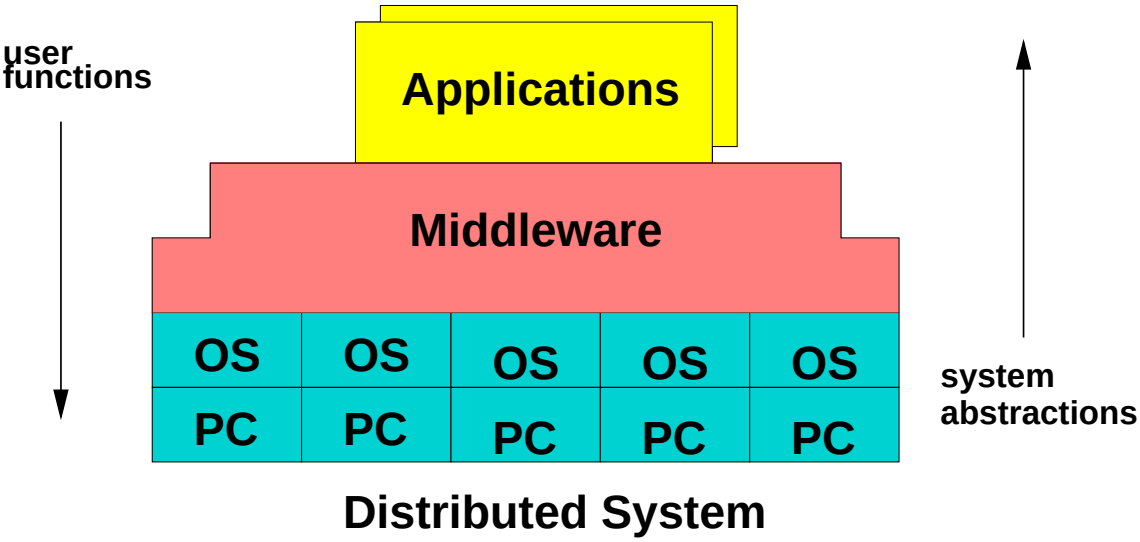

***Figure 3.1**: Functionality of the middleware layer for distributed computing, i.e. task parallelism, distributed task execution, automated data distribution.*

Architecture) [81], a system that allows objects to be distributed on different computers by giving transparent access to them over a communication facility (e.g. over a TCP/IP network). In high performance distributed computing, the data is distributed to a large number of computers so they can work in parallel and achieve speeds that are much higher than on a single computer. Message passing libraries such as PVM and MPI evolved into middleware packages offering many more services than simple send and receive calls for data communication.

In the ideal middleware layers for distributed computing, the programmer would just specify a sequential program and the middleware layer functionality would take care of detecting parallelism and modifying the program so that it can run in parallel. In practice this is not yet the case and the programmer must help the system with hints for partitioning the computation into parallel tasks or by specifying what could be done in parallel and how data could be partitioned if needed. There are dozens of software packages (e.g. open source MPI [94] - a message passing library, Sciddle [6] - a task scheduling system, TP-Lite - a software package [17] for partitioning SQL queries) which act as a middleware layer to facilitate parallel or distributed computation and which offer many different programming paradigms at various levels of abstraction. For the investigation of the performance aspects discussed in this thesis, we are focusing on software systems that include explicit task parallelism and that distribute a computation according to the master-worker programming paradigm.

3.4 Functionality of the Middleware Layer

3.4.1 The Benefit of Middleware Layers

The middleware layer allows the application writer to work on their problem at a *higher level of abstraction*. The tremendous increase in functionality of applications was enabled by numerous middleware packages which solved most of the difficult programming problems on behalf of the application writer. The middleware provides the user with higher-level programming interfaces so called APIs. The high level APIs mask the complexity of the distributed systems, which are characterized by networks that might be unreliable, by heterogeneity and by non uniform notion of time or non uniform ordering of the events. Due to this masking, the user is allowed to focus just on application-specific issues.

Message passing libraries commonly used for scientific computing (e.g. MPI and PVM) as well as relational DataBase Management Systems (e.g. ORACLE or MS SQL server) are indirectly accessed by the users of more high level middleware layers like e.g. Sciddle or TP-Lite. These higher level middleware layers take care of the distribution and help the application writer to think at a higher level of abstraction. A (relational) DBMS raises the level of abstraction from data records to processing relational queries given in a high level language like SQL.

Because of the middleware abstractions, the users gain *portability for their application codes*. Standardized middleware services running on multiple platforms enhance the platform independence and availability of the application. Middleware layers can effectively hide system dependent details and permit applications to be easily ported from one machine to another. This is particularly important since the lifespan of a good application program is usually much longer than the lifespan of a computer platform. The middleware offers high level, standard services by means of its APIs that can then be used to produce user applications at much lower cost than before.

Middleware services also support *inter-operability within heterogenous distributed systems*. Applications on different platforms can use the standard middleware services to exchange data with each other. The best example of this trend are the virtual machines of all kinds e.g. JavaVM [48, 71], VMWare [111], PVM [47], etc.

Adopting (commercial or open source) standardized components for the middleware layers increases the reliability and decreases the maintenance effort of large software systems in scientific computing as well as in commercial data processing.

3.4.2 The Problems with Middleware Layers

The interest in the middleware components has increased tremendously. Every year we see the presentation of new technologies and some application designers too quickly declare the new technologies to be the solution to all the problems in computing. Unfortunately, less and less attention is paid to the underlying operating system and network

services on which the applications also depend.

In most cases, middleware layers succeed to make distributed programming much easier, but they can make debugging for correct execution and maximal performance much harder since we are lacking tools and instrumentation for the analysis of correctness, performance and efficiency. Engineering a complex system for scalability and good performance remains a challenging task. The fundamental problem with non-instrumented middleware layers is that the application writer no longer thinks at a system level suitable for debugging but at a much higher level of abstraction. The middleware layer takes care of mapping the high level commands and directives onto some low level system calls. But in contrary, when it comes to the needs of debugging or performance tuning, most middleware packages do not map the complete system state back onto the API level. This is the most important hindrance to debugging distributed systems and many application writers are completely lost once they have to track down conceptual errors in the application or even malfunctions in the middleware or the system itself. Many middleware layers offer some incomplete tools for debugging and performance analysis but systems with a comprehensive concept of matter are very rare. The DBMS layer often incorporates elaborate instrumentation for performance monitoring and tuning, but most of them work on the basis of independent data collection and not on the basis of mapping operating system state or performance monitoring outputs back onto an abstraction level that is appropriate for a user.

Because distributed computing is often considered for the sake of higher speeds due to parallel execution, the task of performance engineering becomes a crucial problem. A precise understanding of performance and resource utilization is not just required for tuning but also for the prediction of scalability or viability of an application on new or alternative platforms. Most high performance computing applications need to know whether they require the strong memory systems of traditional vector supercomputers or whether they could run on cheaper clusters of PCs. Similarly, database applications need to know if they will only run on a symmetric multiprocessor with a single operating system image or if they could also execute on a large farm of independent PCs. The current middleware packages are ill equipped to give answers to these important questions.

4

The Inverted Middleware Framework (MW^{-1})

In Chapter 3 we have stated that most existing middleware packages lack instrumentation for performance analysis related to the distribution of a computation on distributed computing systems. In particular, the middleware packages obstruct the detection of performance bottlenecks and the analysis of architectural problems in the distributed systems. To address these problems, we propose a novel method for performance analysis called inverted middleware. As a framework, our inverted middleware (MW^{-1}) comprises: some new software tools, the activation of some processor features for monitoring performance information and a new analytical model to construct some global performance data from this information.

Our technique for performance analysis emphasizes a few new computer architecture concepts as well as a few new system software concepts. It is based on the theoretical foundation of the near-complete decomposability of distributed computing systems and the aggregation of their computer system hardware and software architecture into a hierarchy of spanning layers with different levels of abstraction (a detailed description of the theoretical foundation is attempted in Chapter 5). In our novel solution, we combine traditional middleware packages and our inverted middleware framework for performance analysis side by side. Since we complement each middleware, considered as a layer of the hierarchical system, by a corresponding inverted middleware layer and we refrain from changing the middleware itself, we can deal with commercial middleware packages as a black box.

4.1 Definition of the Inverted Middleware

The inverted middleware framework supports systematic performance monitoring and assists the process of performance engineering by mapping all low level performance information monitored at the operating system layer, back onto the appropriate higher level of abstraction for which the application code is written. For the particular case of a distributed computing system, our inverted middleware framework comprises of some

software instrumentation at the OS level, some tools for gathering relevant performance data and some analytical models, which help to give a suitable global performance picture, at the application level.

Most computer users in high performance computing are already well instrumented with detailed performance monitoring facilities like high precision timers, performance counters or system call tracing facilities inside the operating system, but these are running separately on each node of a distributed computing system. So for most distributed computing systems, the information for performance engineering is already present at the hardware and operating system level, but currently it cannot be used at the application level due to the middleware environment that often causes a loss of direct control over the machine resources in terms of performance and software efficiency. The functionality of our inverted middleware framework properly addresses this important problem by accessing and translating all the data available for performance studies and performance evaluation even in different context of distributed computing systems (i.e. for scientific computation applications and for distributed databases).

4.2 The Idea of "Inverting" Functionality in Software Systems

4.2.1 Compilers and Debuggers

Compilers and programming language run-time systems (such as JDKs) are highly similar to middleware in distributed computing systems in the sense that they also allow a machine to be programmed in a high level programming language or more generally speaking in a higher level of abstraction. As with an application using middleware, the system state of an executing program must be analyzed when failure (bugs) occurs or when software deficiencies are observed. In the compiler world the tools with inverted compiler functionality are well known - they are called *source level debuggers*. While much research has been done to improve compilers over the past 10 years (code optimization or automatic parallelization), only little effort has been made to improve the state of the art of debuggers (in particular in the presence of optimization or parallelizing code transformations). Among the notable exceptions there is some work which deals with the feasibility of an inverted compiler for debugging in the presence of common code optimizations [1].

4.2.2 Middleware and Inverted Middleware

In our work, we emphasize the matching and reverse matching of abstractions in middleware and inverted middleware respectively. Middleware packages map some functionality most convenient for a user onto a functionality available at the low level of the operating and network system. This process of mapping functionality downwards can be layered extensively and resembles a chain of mathematical functions leading from the high level

standard API to the low level system calls of the underlying operating systems. We express the idea of a layered middleware system as follows:

$$MW = MW_n \circ MW_{n-1} \circ \cdots \circ MW_1 \tag{4.1}$$

The parameters handed from layer to layer would be the sum of all the system calls.

Therefore it is a most obvious choice to design a similarly layered implementation of the inverted middleware to map the system state onto a high level API. The inverted middleware will resemble a chain of inverse mathematical functions, but this time mapping the performance monitoring information about the machine resources gathered at the operating system onto the abstract world of high level standard APIs. In a forward representation, this can be written as:

$$MW^{-1} = MW_1^{-1} \circ MW_2^{-1} \circ \cdots \circ MW_n^{-1} \tag{4.2}$$

This time the parameters of the layered functions would be the performance-related system data.

We propose that the inverted middleware is developed separately and that its functionality is not to be integrated with the middleware itself. This approach might impose some limitations in what performance data can be gathered and what problems can be analyzed, but in our experience with manual instrumentation of specific middleware packages we have learned that middleware developers will rarely care to instrument their middleware for backward matching system state into user level abstractions [5] and that the only way to use commercial middleware packages as a black box is to provide the inverted middleware separately.

4.2.3 Structural Symmetries of Middleware and Inverted Middleware

Figure 4.1 summarizes the relationships between middleware and inverted middleware as two parts of complex software systems permitting the execution and the performance analysis of distributed applications hand-in-hand. The graph shows some obvious structural symmetries of middleware and inverted middleware executing with an application code on a particular system platform. The middleware fits the fairly high level abstract requirements of a computation into specific demands for the machine resources provided by the common distributed platform such as processors, disks, network bandwidth.

The resource management within the middleware is often done by a system independent part on top of a system dependent part. The application is directly using the middleware's system independent part which provides some API calls. Such a part is independent from the platform executing the code and hides the system dependencies for the sake of portability and inter-operability of the different components in a heterogenous distributed system. For the system independent part, a system dependent part is responsible for the correct adaptation of the user requests to the platform features and the

distribution of the computation among the local resources of the distributed computing system.

On the other hand, the inverted middleware maps the machine resources usage into user functionality for more effective performance analysis. The distributed computing system is directly connected to an application independent part which is responsible for sampling, collecting and patching the low level performance data into global information. The performance data is then provided to an application dependent layer to be re-arranged in a more effective way for the user. Figure 4.1 describes the interesting combination of horizontal and rotational symmetries in our software framework.

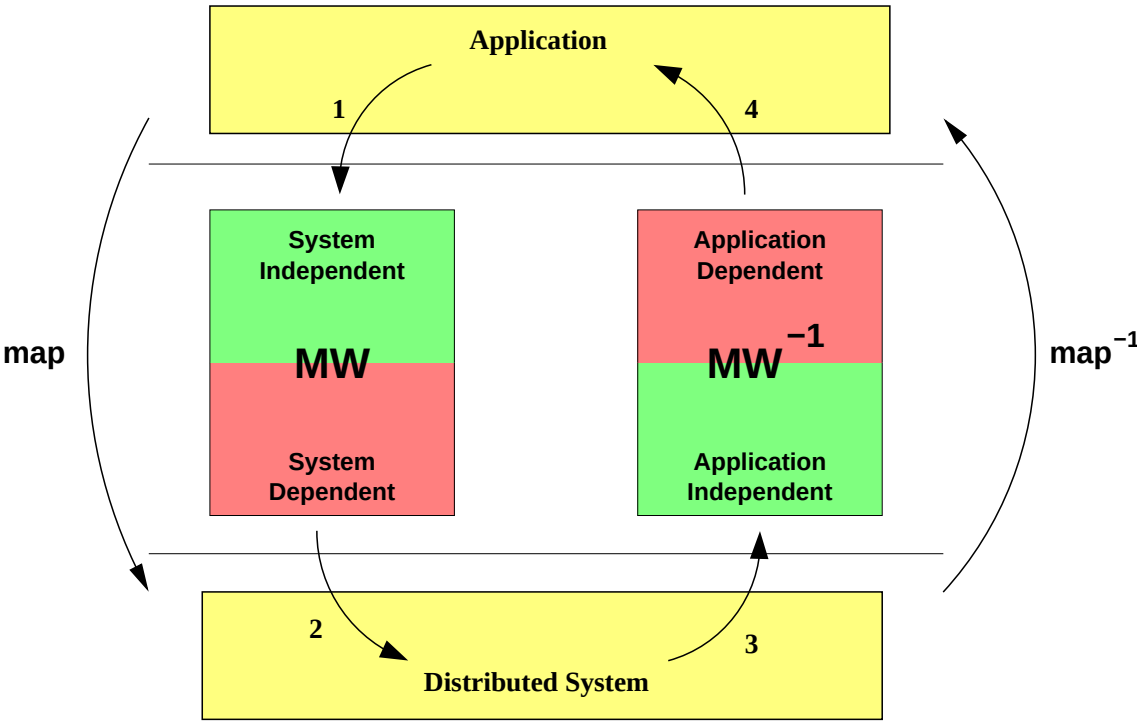

***Figure 4.1**: Symmetric structure of middleware and inverted middleware executing with an application code on a particular system platform.*

The interactions between the different layers of the computing system in Figure 4.1 take place by means of:

1. **API calls** between application layer and system independent part of the middleware layer,
2. the **system calls** between the system dependent part of the middleware layer and the distributed system layer,
3. **performance monitoring hooks** of an extended `/proc` file system and some hardware performance counters of the processors (registers) that are sampled by the

application independent part of the inverted middleware,

4. a **performance analyzer** which acts on applications found to have bottlenecks or loss of performance to improve them of to move them onto more effective architectures.

4.2.4 Closing the Loop of Software Functionality

The inverted middleware regularly pulls performance data about machine resources and returns information for performance profiling and for performance predictions. Through this information, a performance analyzer should be able to suggest a list of changes to the relevant application factors for optimization. Most of the time, combining multiple changes may have an opposite effect on the performance and therefore the performance analyzer has to identify and assign priorities to the possible optimizations. In most cases a fully automatic loop of optimization is unrealistic and possible optimizations due to the data provided by the inverted middleware finally requires the intervention of the user who has to act on the application or code. It remains open to what extent an automatic optimization of applications based on elementary knowledge about resources usage is possible. In a previous section we have stated that the relation between middleware and inverted middleware is similar to the relation between compiler and debugger. This similarity can help us envisage possible solutions for automatic optimization of applications using the inverted middleware.

For automatic optimizations of an application, the data provided by the inverted middleware has to be used to influence the characteristics of an application thereby improving its performance without any user intervention. Referring to the compiler/debugger domain, it means that the information provided by the debugger is directly used in the program for solving errors and improving performance.

In scientific computation, a modular approach for automatic tuning of real applications is already present in tools like the Active Harmony system [59], an automated runtime tuning system which provides the tuning of applications during their run-time execution by monitoring the underlying library performance and switching between underling libraries as need. Most of these tools just focus single application components like the libraries but still suffer from the poor modularity which is characteristic of real applications in the scientific computation domain.

In the compiler/debugger domain, an example of automatic optimization with runtime information is proposed by Kistler and Franz in [69, 70]. Kistler and Franz use a history of past profiling data to dynamically re-generate parts of the computing system on-the-fly to achieve re-optimization of the code in the domain of object-oriented and component-based applications. Their approach relies on an estimate of memory performance (e.g. cache miss penalty) gathered from profiling data. In the dynamic optimization

space, a new version of all affected procedures is automatically generated according to a cost-benefit model provided by profiling data.

To use the inverted middleware and its information for any automatic optimization successfully, an application requires a modular structure in which its components are well-defined. To use the inverted middleware for dynamically re-engineering applications by means of the inverted middleware-filtered performance data, it is necessary to re-think the structure of applications in terms of layers and components. Any application should be decomposable into components (e.g. parts of queries for database applications, procedures for the scientific applications) and each component should be suitable for optimizations suggested by the inverted middleware framework. Components of an application can be added, removed or modified according to the architectural properties of the underlying compute platform e.g. the speed of the single nodes or the network.

The performance analyzer for automatic re-optimization is configurable as a set of software parts on top of the inverted middleware and should be driven by the data provided by the inverted middleware itself. It concentrates its efforts on optimizing the most critical and most beneficial components of an application. Each optimization requires a well-defined domain and the techniques change with the application and its modularity. The performance analyzer should have a certain platform- and application awareness due to the data provided by the inverted middleware.

As for the work proposed by [69, 70], the re-optimization should be repeated continously and dynamically as long as the performance analyzer on top of the inverted middleware and the inverted middleware itself are running on the distributed computing system side-by-side with the application components (e.g. queries, procedures) running on top of the middleware.

4.3 Internal Structure of Inverted Middleware Framework

The inverted middleware framework has a hierarchical structure as well and is structured into three different layers: an application-specific layer, a distribution-specific layer and a system-specific layer. Figure 4.2 shows the three layers and how they interact with each other and with the other components of the distributed system.

The *system-specific layer* of the inverted middleware framework monitors and collects system-specific performance data. System-specific performance data includes information on resource usage and bottlenecks gathered over the lifespan of the application-run. In our current prototype, we monitor the resource usage for resources like the CPU, the memory, the disks and the local network as sampled information on each platform node.

The *distribution-specific layer* gathers all the performance-related data from several nodes and patches it together into a single coherent view of the overall performance in the entire system. The information collected by this layer is handed over to the application-specific layer. The inverted middleware framework inherits the master-slave setting from

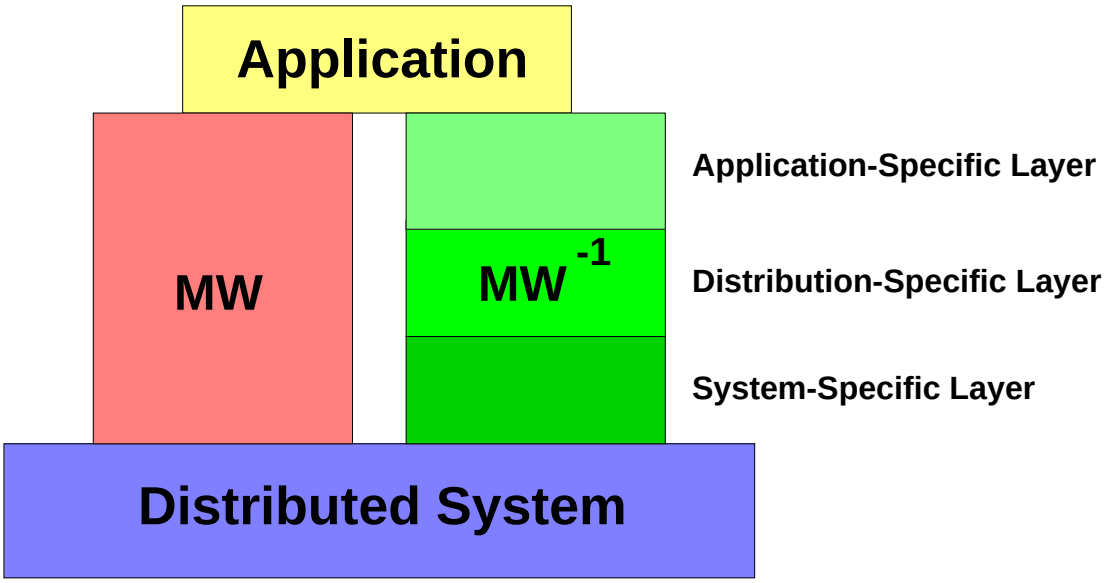

***Figure 4.2**: Structure of the inverted middleware framework in the distributed environment.*

its middleware layer counterpart. The information gathered by the master is already properly filtered by the slaves and ready for processing at the application-specific layer.

The *application-specific layer* uses the global performance data of the entire system for
application-level optimizations and performance predictions. The top-most layer uses the analytic performance model of the application to map the typical performance related response variables of the global view of the application computing onto some suitable suggestions for performance tuning, e.g. onto suggested changes to parameters that are performance-relevant factors of the computation.

4.4 Inverted Middleware for Clusters of PCs

In this section we describe certain design issues that had to be resolved during the implementation of the inverted middleware framework for clusters of PCs. When we speak about clusters of PCs, we mean large Beowolf type clusters of commodity PCs but also well-engineered high-end clusters of PCs. All our software related to the inverted middleware is written in C.

4.4.1 The System-Specific Layer: Sample Data

The purposes of the system-specific layer is to sample performance information locally on the compute nodes of the distributed system. The sampling process not only looks

at the execution time and its components but also deals directly or indirectly with the machine-resource usage on the distributed platform.

Granularity of the sample

We examine the distributed computation of an application run over an extended amount of time. Therefore we have to cope with a large amount of performance data. We call this data response variables according to [62] because our experiment is about application performance. An exhaustive representation of all relevant performance data is rather expensive. In particular if the distributed system has a large number of nodes to be monitored, a huge amount of information would have to be exchanged within the inverted middleware layers and this would lead to a high overhead. Therefore the amount of the system information is filtered. Instead of recording every event possible, only some performance related metrics must be captured by sampling. If samples are not taken frequently enough (the granularity is too coarse), it is possible that the statements about the system are inaccurate. On the other hand, if they are taken too frequently (the granularity is too fine), the system is perturbed by floods of monitoring traffic. The *granularity of samples* must be a proper trade-off between accuracy of the system view and the cost of the system monitoring. For an efficient and accurate sampling of performance data, an appropriate granularity of the sampling according to the size of the monitored system and the features of the application is necessary.

Sampling mechanism

Possible sampling mechanisms include either methods that gather resource information about usage allocation of machine resources based on application-driver events (e.g. standard performance tuning in ORACLE) or methods that gather dynamic resource information at regular intervals. The data gathering in the inverted middleware framework relies on the latter method. The performance monitoring framework samples at regular intervals performance data whose size is related to the kind of application, the lifespan of its run and the needs of the user. This mechanism not only allows us to know what consumes how many resources, but also when and where these resources are required.

Level of integration into the operating system layer

One important point in the design and development of the inverted middleware is how deep the system-specific layer should be integrated into the operating system. There are several aspects which have to be considered. First of all, the speed of the the new functionality provided by the system-specific layer. Second, the complexity of the implemented functionality. Last but not least, the portability of the system-specific layer to different distributed architectures. These aspects demand a strategy of least possible intrusion into

the operating system and in particular limit the work that can be done in the kernel of the operating system.

Our solution is to implement the system-specific layer of the inverted middleware framework outside the kernel as a daemon. This renders monitoring slightly slower and less accurate than as it would be an in-kernel implementation, but remains much easier to maintain with new versions of the OS kernel (which frequently appear with the LINUX OS).

Our current prototype is external to the kernel and is based on the LINUX */proc file mechanism* to write performance data. However, the information available in the `/proc` file is significantly extended in our framework. The inverted middleware also uses the *hardware performance counters* of the Pentium processors and these are made accessible through a library we have implemented ourselves [96].

The extention of the `/proc` file system

The total amount of blocks and the number of bytes exchanged from and to the local disks is a performance information already gathered by the kernel but not made directly accessible through the `/proc` file system. Therefore the kernel does not have to be changed, but the disk information has to be reported into some `/proc` files. We choose to integrate the important disk information into the `/proc/stat` file. The additional information integrated is: the amount of bytes read and written and the amount of bytes accessed sequentially and non-sequentially. Most information about the communication (i.e. the amount of packets and the amount of bytes received and transmitted for the different communication networks) is already reported in the `/proc` file system.

Using the hardware performance counters of the Pentium processors

Modern microprocessors like the members of the Intel Pentium Series offer so called Performance Monitoring Counters (PMC) which are special purpose registers on the chip that count specific events which take place in the CPU. Examples of performance-related events are: the number of cache accesses or the number of floating point operations.

A library has been implemented to permit a simple but efficient way to access these performance counting registers from the system-specific layer [96]. The library is easy to use and offers a uniform interface under LINUX and Windows NT. It supports the monitoring of symmetric multiprocessing (SMP) machines (nodes of the PC cluster with more than one CPU).

Most of the functionality related to the performance counters has been implemented directly in the library. Nevertheless, a reduced number of changes inside the kernel was necessary. In order to maintain a good portability of the library on future kernels, we have tried to access the drivers only when and where it has been indispensable. Our library works with the LINUX 2.2.x and 2.4.x kernels.

Some new functions for reading the counters more efficiently or setting some flags on all the CPUs have been inserted into the kernel. A kernel module provides support to initialize, to start, to stop the performance monitor counters and to read their values. The library uses the interface provided by the kernel module which, in turn, uses the new kernel support for reading the performance information.

The library always runs the register-read calls on all the CPUs at the same time. Reading the registers of all the CPUs or for just one CPU does not make any real difference of the performance and overhead. Our library has been validated by tests and some performance measurements with cross checks against other mechanisms [19, 96].

Cost of sampling

Intrusion of the performance monitoring overhead must be kept to a minimum. Most of the monitoring time is spent in the non-operational phase managed by timing routines. The granularity of the sampling determines the size of the regular intervals by which the information about the local nodes is collected. The information of each sample is written into packets whose size is less than 512 bytes. The total collection time for a packet is irrelevant: the total execution time of a monitored application and the total execution time of the same applications without the monitoring remains the same.

4.4.2 The Distribution-Specific Layer: Collection of Data

The distribution-specific layer re-collects the usage information about the machine resources. If the middleware layer distributes the task workload among the nodes of the distributed network, the inverted middleware framework on the other hand collects the performance data by means of a distribution-specific layer.

Master-slave setting of the collection

Most high performance applications in our environment mainly exploit task parallelism with a master-slave compute setting. In a similar way, the inverted middleware framework collects the performance data in a master-slave setting.

To ensure a consistent and accurate performance picture of the system and to catch the real-time behavior of the monitored application regardless of monitoring tool loading, frequent and fast transmissions of performance data have to take place from the monitored slaves of the distributed system to a monitoring master.

In our prototype, we dedicate a separate node to be the monitoring master and do not have to run the application master on the same node. In this setup we leave the application user to decide where to put the application master, while the inverted middleware system can decide where to put the monitoring master.

All information for the performance analysis is gathered at every node of our distributed system in parallel by the system-specific layer. Our inverted middleware framework is responsible for putting together the information in a global view. The monitoring processes on the individual nodes send the performance information immediately to the monitoring master increasing the transmission of frequent, small packages of data on the network.

The protocol for the monitoring traffic: UDP/IP vs. TCP/IP

Most programmers are building distributed monitoring tools on the basis of the TCP/IP protocol for the monitoring traffic, since it is reliable and handles flow control [27, 28, 98]. The TCP/IP protocol is probably mandatory when monitoring is based on coarse grain application-driven events. In this case the user cannot afford to lose any information and the reliability of the data is one of the most relevant issues of the performance data collection. On the other hand, monitoring based on sampling of low level of performance data at regular intervals can afford the loss of packages and therefore no longer requires a TCP/IP connection. To reduce the overhead and permit fine granularity, we can transmit the data through more time- and resource-effective communication protocols such as UDP/IP [83].

The system-specific layer relies on monitoring based on sampling at regular intervals. The layer sends the performance information to the monitoring master immediately. These transmissions of frequent, small packets of data over the network using the TCP/IP protocol is inefficient since the traffic is bursty. In such a scenario, the UDP/IP protocol is characterized by less communication disturbance.

The advantages of the UDP/IP protocol

Using TPC/IP for the performance transmissions may cause network overload due to the flow control which characterizes this communication protocol. The flow control induces acknowledgment messages which act as synchronizations. Moreover, flow control is responsible for retransmissions in case of delay or when the data gets lost, thereby increasing communication intrusions. For reduced communication intrusions due to the monitoring traffic, the network overload related to the acknowledgment and retransmission of performance packets has to be kept low [27, 28].

Without re-engineering the transmission protocol of the application or going into the complexity of complete removal of monitoring intrusions, we propose to use UDP/IP for keeping the monitoring traffic of performance data low in clusters of PCs. UDP/IP is a simple datagram oriented protocol without a procedure for ensuring that individual messages are not lost in transit. Consequently there is no perturbation on the monitored application due to unwanted synchronization. Because UDP/IP has no constraints on the send rate of the performance monitoring packets and does not need to maintain connection

states of the monitoring processes, it supports a larger amount of monitored nodes than TPC/IP [83].

The UDP/IP protocol works quite well and transmission errors are infrequent in clusters of PCs in which links are short and full crossbar switches are common. However, for large-size clusters or loosely-connected distributed systems, the communication between the monitoring master and the monitored slaves takes place by means of unreliable messaging. When the network is overloaded during bursts, some performance packets may get lost. However, we regard the loss of packets as a sort of regulation mechanism for coping with overload on the network.

The loss of performance messages is no longer a problem in obtaining accurate performance pictures of the system and consequently, we can afford losses of performance packets, when the monitoring tool works on samples of performance information and not on whole populations [62]. The performance samples should be frequently gathered and integrated into a probabilistic sampling framework.

Constraints for a proper use of UDP/IP

When the absolute accuracy and reliability of the performance data is no longer compulsory, while the perturbation of the monitoring on the monitored applications have to be minimal, the TCP protocol for the gathering of the performance data is no longer the proper protocol. However, some constraints have to be fitted in for a proper switch from the TCP/IP to the UDP/IP protocol.

First of all, the mechanisms for collection of performance data have to rely on the gathering of dynamic resource information at regular intervals. Such mechanisms not only allow us to know what consumed how many resources during the execution of an application, but also when and where these resource consumptions occurred. The nodes sample and send the performance information at regular times to the monitoring master.

Moreover, a master/slave setting should characterize both the monitored application and the monitoring. The monitored application has to be characterized by a long run-time t_{run}, while the interval of time among two samples t_{sam} has to be much smaller than t_{run}:

$$t_{sam} \ll t_{run} \tag{4.3}$$

The time for sending a package from the slave to the master t_{lat} should be much smaller than the sample interval t_{sam}:

$$t_{lat} \ll t_{sam} \tag{4.4}$$

Under such constraints, we can afford an accurate reconstruction of a global view of the performance of the distributed system through the statistic interpolation of the information received and the treatment of the lost packages as sample errors.

Notion of time on distributed systems

The monitoring master has to be able to patch together the performance data of the several nodes in a consistent manner with a global wall clock time scale. To address this issue, a consistent, single notion of time among the several nodes of the distributed system has to be maintained. To overcome this problem, the monitoring tool has to introduce some mechanisms of synchronization during the performance sampling. Mechanisms based on ordered events such as barriers for synchronizing the computation and the communication may change the run-time behavior of the monitored application introducing idle times and therefore changing the scheduling and execution of the processes.

To cope with the problem of maintaining a global notion of time and at the same time reducing scheduling and execution intrusions, we propose a notion of time based on accurate built-in cycle counters. We avoid unnecessary synchronizations, which are normally introduced in the code by mechanisms based on ordered events, through a highly precise synchronization at the start of the monitoring session. As the execution continues, the synchronization process is relaxed to a loose synchronization [42] in which the monitoring tool synchronizes sampling by looking at the highly accurate cycle counters in the CPUs. The built-in cycle counters mechanism acts as a *virtual barrier*. The cycle counters have to be delivered as timestamps in the performance packets containing the sample information sent to the monitoring master.

Recovery of performance figures despite loss of packets

Based on our concepts and early experiments, we can carry out an accurate and reliable building of performance figures for the monitored distributed application despite unreliable performance messaging due to the UDP/IP protocol. We can afford the loss of packets treating the lost information as sample errors and interpolating the performance data in the statistical framework of a sampling performance monitor.

The performance data that the monitoring master gets from each monitored slave are ordered sets of workload samples on the local nodes. The notion of time that we use with its timestamps of the cycle counters helps to maintain the concept of order for the samples. Probabilistic statements about the range in which the performance of most nodes would lie can be made [62]. The concept of a confidence interval can be used to measure the quality of the transmission. The less packets get lost, the larger is the number of samples and at the same time, the higher the confidence in the performance picture will be.

To estimate a global view of the workload of the nodes in the distributed system as a function of the performance data gathered by the monitoring tools, regression models can be used when the performance data sampled are quantitative as well as categorical or non-numerical [62].

On the proper usage of log data

The amount of data gathered locally and sent as a sample in a performance packet is small in size however, the total amount of information gathered from all the nodes grows rapidly once the distributed system has a large amount of nodes. Old information is less relevant for performance evaluation. Therefore we propose the immediate consumption of the data by the application layer.

The possible choices are: a log file for each node versus a single large log file for all the nodes. In our approach, we may have both a single log file for each node and a single log file for all the nodes; the choice is left to the user. Each item of information related to a specific node is marked by the node identification and the CPU clock time when the information was picked up on the local node.

4.4.3 The Application-Specific Layer: Modeling of Data

The collection of performance data is worthless as long as we do not act on the data for building useful information for the user. Possible methods for the rebuilding of data rely on mathematical functions for designing performance models. The application-specific layer of the inverted middleware includes a performance model of the application that can translate the elementary knowledge about the resource usage into high level answers to performance questions and bottlenecks suitable for suggesting performance optimization and performance predictions to the user. In this section we will go through some relevant critical issues in a general way, however, a real implementation of the layer is done for the two specific domains chosen (i.e. scientific computation domain and distributed database domain) in Chapter 7 and in Chapter 8.

Performance modeling

Our current models are simple sets of formulas which allow the calculation of the individual execution time components due to the usage of each relevant machine resource.

During a first phase of design and parallelization/migration of the applications, we can derive an analytical time complexity model that captures all essential parameters of the real applications. The predicted outcome of the model is the execution time of the application written as the sum of the several time components due to the machine-resource usage. The model parameters are the factors chosen and the response variables measured during the local monitoring.

We distinguish between application and platform factors. The application factors are directly related to the features of the application such as the size or amount of data considered in the application for the particular run. On the other hand, the platforms factors are strongly related to the features of the platforms and to the resources which are abstracted by the middleware.

Modeling tradeoffs

A mathematical model is not as detailed as the system itself, but can be a good representation of the system and therefore can be more or less accurate. Modeling is based on executions and on elaboration of a number of simulations. Tradeoffs between accuracy and elegance of the model are compulsory: accuracy, complexity, cost, transparency are all factors which influence the quality and the tractability of the model considered.

Model validation

Validation means ensuring that the assumptions used in developing the analytical model are reasonable. The results produced by the analytical model should be close to the application performance observed in the real system. Since the real system also exists and measurements are possible, this leads to a natural process of validation of the analytical model by comparing the model results to the real system measurements.

4.5 Comparing Existing Monitoring Instrumentation with Inverted Middleware

In our performance analysis, we emphasize the aspect of mapping and reverse mapping of abstractions in the middleware and the inverted middleware respectively. To the best of our knowledge, no existing monitoring system attempts to work with middleware packages in a similar way.

We take a conceptual approach in which we try to model, isolate and invert the functionality of the middleware without violating its integrity. The middleware can remain a black box. Because of this approach, we can deal with vastly different classes of software systems such as middleware packages for scientific computation and database management systems and with vastly different classes of applications (e.g. scientific computation applications, distributed databases). Our inverted middleware is a highly independent tool which creates a reverse mapping of the low level metrics into high level application parameters. Traditional tools for performance analysis measure the time components related to high level application features directly (e.g. computation time, communication time) by means of re-engineering of the application code or of the middleware code. In our performance study of a distributed computing system, we look at the run time in terms of time components, each one attributed to some critical machine resources. Each time component is accurately modeled starting from the observation of these machine resources. For each time component, we isolate and sort out which resource contributes to the slowdown in the overall performance. We consider frequency and time traces of the resource usage and collect the information at run time.

4.5.1 Existing Monitoring Tools vs. MW^{-1} in Scientific Computation

Most of the existing monitoring tools come in the form of toolkits or subroutine libraries. The prevalent principle is *code reuse* [43] instead of design reuse. Our inverted middleware comes in the form of a framework which also works well in the different environments of distributed computing. Frameworks emphasize *design reuse* over code reuse.

In the context of the *performance analysis of scientific computation applications*, most of the current methods for performance analysis require that the decisions related to instrumentation are taken at the beginning of a monitoring session and remain fixed during the execution of the monitored application. When the user uses a subroutine library or a toolkit for the monitoring of performance, the monitored application needs some changes of the instrumentation and the change of the performance metrics to be monitored requires a re-engineering or at least a relinking of the application binary or even a redefinition and rewriting of the performance monitoring library. For example using tools like VAMPIR [80, 21], a program requires relinking with the VAMPIRtrace library in most cases. If application-defined events are to be recorded, a recompile action is necessary. Toolkits like SvPablo [33, 86] capture performance data on platform architectures using C, Fortran and HPF compilers and visualize the information. They mainly relay on instrumentation in the source code of the applications. Such instrumentation can be generated automatically with some compilers like the PGI HPF, or manually. The instrumentation records high level events such as loops, functions and procedures. The use of SvPablo requires the generation of a new version of the source code containing links to the selected events to be monitored. The PAPI hardware performance counters [75, 39] and the MIPS R10000 hardware performance counters [118] have recently been integrated in SvPablo, enabling the collection of hardware performance data on different platforms. Hardware information is generated automatically once the application is instrumented with the enhanced SvPablo library.

On the other hand, our inverted middleware framework has an overall systems structure and comprises parts of code which are reused independently of the application monitored or performance metrics investigated. When the inverted middleware framework is used, up to the highest level its application-dependent layer has to be adapted to the kind of application which is monitored and to the performance metrics which are being investigated. This is only necessary for the application dependent part of the framework. Our approach, based on inverting the system functionality, tries to accomplish the goals without any intrusion into the application code. A re-engineering or relinking of the application is not required. The design decisions that a user has to make when he/she works on the monitored application are not conditioned by the monitoring process.

There are other methods like the W^3 Search Method [59] which also try not to embrace the fixed approach to data collection present in the tools listed above. Similar to the inverted middleware, the W^3 Search Method is application and machine independent.

Both the inverted middleware and the W^3 Search Method allow to investigate possible performance bottlenecks in distributed systems characterized by both heterogenous combinations of machines and homogeneous machines. In addition, the inverted middleware is middleware independent. Moreover, both the inverted middleware and the W^3 Search Method allow the search for performance problems during the execution of the monitored application by dynamically turning on and off the collection of the performance data. The inverted middleware addresses the issue of the excessive volume and the kind of performance data collected by gathering only significant data at regular intervals. The inverted middleware allows the analysis of data during the collection phase and during the execution of the monitored application. The user can redefine the data collected or the interval between two performance samples by changing the parameters of the daemon processes locally on the nodes. By changing the size of the interval between two performance samples, the user can dynamically act on the granularity of the data collection.

Looking at the perturbations of the applications due to the performance monitoring, the inverted middleware tries to keep intrusions low by minimizing the instrumentation overhead. In our framework, the trace data is not kept locally in each processor's memory (as in performance libraries like VAMPIRtrace), but is sent in small packets to a monitoring master. In tools like VAMPIR, the performance data is saved to disk when the application is about to finish and post-processed for post-mortem optimizations. On the contrary, the inverted middleware allows a dynamic processing of the information at run time.

The inverted middleware provides performance characterizations and predictions a real application on distributed systems at run-time of by means of performance monitoring and observations. The method proposed by Vraalsen and others in [112] is also based on modeling and predictions using monitoring and observations at run-time. Vraalsen introduces the use of performance contracts in which application-system commitments are specified. An application signature model is defined for the prediction of application performance. A fuzzy logic is used within the decision procedures of the Autopilot monitoring tool to detect if the monitored information fulfills the expectations specified in the performance contract. Contrary to the method proposed in [112], we do not need to insert any embedded components into the code of the application under investigation before the application is executed (Autopilot provides read access to the remote application hosts via software sensors embedded in the application code). We also do not specify any a priori expectation of the performance to be observed. Similarly to the method proposed in [112], the inverted middleware still requires the intervention of the user or an agent for optimizations.

4.5.2 Existing Monitoring Tools vs. MW^{-1} for Distributed Database

In the context of the *performance analysis of distributed database applications*, most DBMS packages (e.g. ORACLE) incorporate elaborate instrumentation for performance monitoring and performance tuning, but such instrumentation normally works on the basis of data collection within the DBMS and not on the basis of mapping the operating system state or the basic performance monitoring data back to an abstraction level that is more appropriate for a user. In particular, the performance monitoring instrumentation of ORACLE [53] only accounts for the total count of operations and neither allows us to efficiently sample performance counts at certain intervals nor allows this information to be collected efficiently from a large number processing nodes in a cluster. While some of the logical or table access counts would certainly be interesting, most performance information of the database management system does not directly relate to the usage of physical resources in a distributed system and is therefore hard to use in a framework that aims at predicting the execution time based on application and platform parameters.

5

Decomposability of Distributed Computing Systems

The theory of near-complete decomposability deals with the design and arrangement of complex systems in a hierarchy of abstract machines. The system components and subcomponents are aggregated into subsystems, also called layers, in a process which aims to simplify the system by breaking it up into layers of smaller dimension and of lesser complexity. The interactions of the components and subcomponents within the layers (*strong interactions*) are plentiful and fast compared with the interactions between layers (*weak interactions*) which are sparse and slow. The layers of a hierarchical, well-organized system are characterized by different levels of abstraction for the system resources.

The theory of near-complete decomposability is not specific to computer science. It has been introduced and studied in economics by Ando and Simon in 1961 [93]. The theory has been used for economic structures and extended to a variety of biological models. In computer science, a detailed theory of the near-complete decomposability has been established by Courtois in his monograph [30]. Courtois's principal insight is that complex systems such as computer systems are hierarchies of components within which interactions are strong and fast compared to interactions between components.

In computer science, the near-complete decomposability theory was originally used to *facilitate step-by-step constructions of complex systems*. Such a multi-level aggregation approach can be extended to distributed computing systems and therefore *facilitate a stepwise performance analysis* (i.e. workload characterization and prediction) of such complex systems. According to Courtois, in complex systems "...the interactions between the levels of an *efficient* and *convenient* hierarchy of abstracts machines are likely to obey the condition for near-complete decomposability" [30]. In this chapter, we attempt to adopt this idea of the near-complete decomposability as stated by Courtois and adapt it to complex distributed systems in order to decompose them into layers and so make the performance analysis for this class of software systems much easier.

Efficient and *convenient* hierarchical decomposition of distributed systems is a subjec-

tive matter. Nevertheless, we think that the concept is highly useful to properly cope with performance analysis of complex systems. In the following sections, we will focus our effort to find proper decomposition levels for the distributed systems following concrete and objective criteria.

5.1 The Theory of Near-Complete Decomposability

5.1.1 Mathematical Definitions

The systems to which the theory of near-complete decomposability refers are stochastic systems of the form:

$$\tilde{y}(t+1) = \tilde{y}(t)Q \tag{5.1}$$

where $\tilde{y}(t)$ is a transposed, row probability vector and Q is a transition matrix of order n. The stochastic system has n states. $\tilde{y}_l(t)$ is the unconditional probability of the considered system of being in the state l (for $l = 1...n$), at the time t. In the matrix Q, the element q_{kl} represents what is called the conditional probability that the system is in the state l at the time t assumed that it was in the state k at the time $t-1$.

The matrix Q can be written as:

$$Q = Q^* + \varepsilon C \tag{5.2}$$

where Q^* is a block-diagram matrix i.e. a matrix of square stochastic sub-matrices along the diagonal and zeros elsewhere. Each sub-matrix represents a subsystem and is supposed to be indecomposable. C is a square matrix of the same order of Q^*. It assures that both matrices Q and Q^* are stochastic. Last but not least, ε is a real positive value. If $\varepsilon = 0$ then the system is completely decomposable into independent aggregates of states described by the sub-matrixes in Q^*. On the other hand, if ε is small then the system is near-completely decomposable. It means that most transitions occur with aggregates, but there are occasional transitions between aggregates.

A near-completely decomposable system is characterized by two stages: a *short-term period* and a *long-term phase* when the short-term period is over. During the short-term period, the system behaves approximately as complete-decomposable and each subsystem converges to its own local equilibrium independently from the state of the other subsystems. Once the short-term interval is over, the states of equilibrium for each subsystem remain while the whole system evolves towards a global equilibrium under the influence of the interactions among the subsystems, called weak interactions.

5.1.2 Hierarchical Aggregation of Items of Functionality and Levels of Resource Abstraction

A system which obeys the theory of the near-complete decomposability can be decomposed into smaller subsystems or layers. For such a system, either the interactions within

the layers can be evaluated as if interactions among layers would not exist, or the interactions among layers can be investigated without considering the interactions within the layers. The decomposition of a complex distributed computing system into layers makes it more suitable for performance analysis, performance modeling and prediction.

Each layer comprises a hierarchical aggregation of items of functionality. A concept strongly related to this is the concept of abstracted representation of the resources at the different layers of the distributed systems.

Most applications on distributed systems require fast access to a large amount of different kind of machine resources (e.g. CPU, network, disks) at a low level of abstraction of the distributed platform in order to reach proper speed-up. But fast and efficient access means direct low level dependence on different resources which change with the underlying hardware architecture of the single nodes and with the application. For a more portable access to the machine resources, some abstraction is typically provided by system software layers (i.e. the middleware layer) which may simplify access but may also slow-down the speed thus decreasing the application efficiency. These are the typical advantages and disadvantages of system software layers which act in the system providing hardware abstraction.

A near-completely decomposable system is thereby organized as a hierarchy of $L+1$ layers which act as abstract machines:

$$A_0, A_1, ..., A_l, ..., A_L \tag{5.3}$$

realized with different hardware and software technology. The rate of interactions of the components of an abstract machine A_l with its underlying abstract machine A_{l-1} is higher than the rate of interaction with the components of the next upper abstract machine A_{l+1} and so are the level of abstraction implemented.

Figure 5.1 summarizes a possible hierarchical organization of a system in layers. Each layer is characterized by different levels of abstraction for the machine resources and the functionality provided. As we move from the bottom to the top of the hierarchical organized system in Figure 5.1, the layers make access to the resources of the system in an increasingly user-friendly and less tedious way, but at the same time the speed and efficiency of these accesses to the resources decreases.

The application under test has an overall execution time that is the time to access the resources at the higher level of the hierarchical system (A_L in Figure 5.1). Given the hierarchical structure in the system, the overall execution time is the sum of the execution times needed to climb up the layers of abstraction. A resource effective hierarchical system offers a fast and direct access to the machine resources despite the overlap of several layers and the execution time of the application approximates the time of the faster layer A_0 minimizing the time components of the additional layers or abstract machines $A_1, ..., A_l, ..., A_L$. On the other hand, for non-effective hierarchical software systems, the overall execution time of an application located on top of the complex software system is

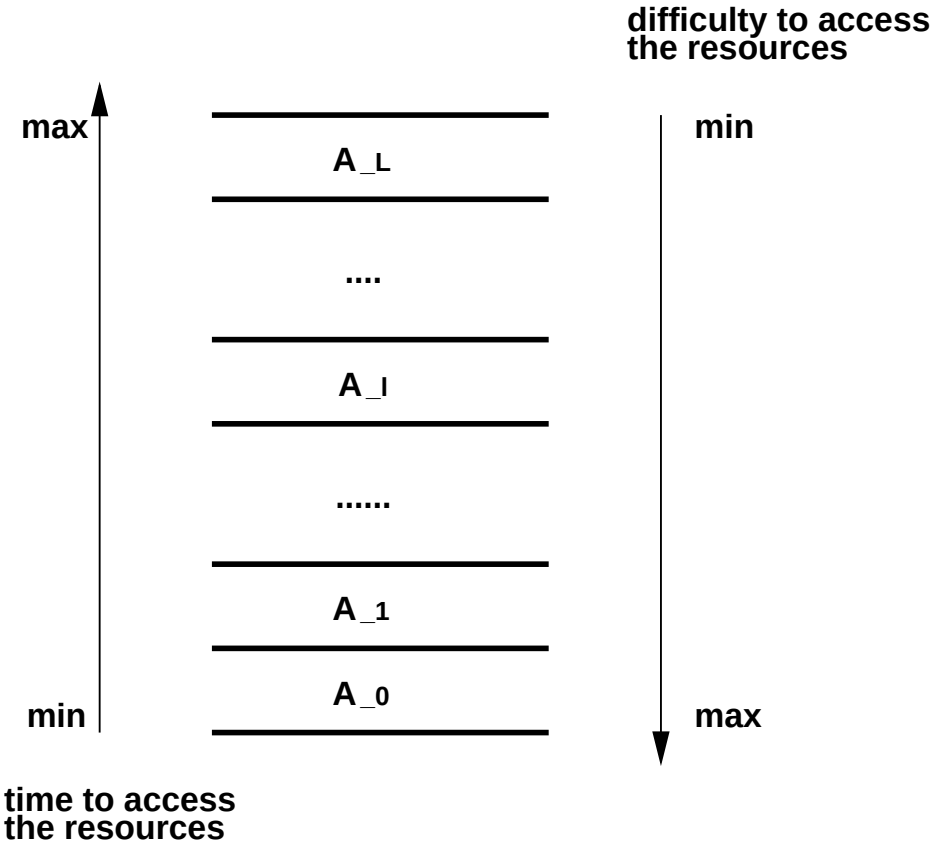

Figure 5.1*: Hierarchical organization of the system in layers: the layers ($A_0, A_1, \cdots, A_l, \cdots, A_L$) are characterized by different levels of abstraction for the machine resources.*

dominated by overheads within the lower layers.

5.2 Near-Complete Decomposability in System Software

5.2.1 Near-Complete Decomposability in Program Behavior

Near-complete decomposability in program behavior has been used by Dijkstra for hierarchical modeling and organization of the software of multiprogramming computer systems.

Dijkstra showed in [34, 35] how advantageous it is from a point of view of the designer to structure a computer operating system as a hierarchy of levels of abstraction. The levels of abstraction are considered as an ordered sequence of machines each one defined and executed in term of the previous one. A practical example of such a hierarchy of levels of abstraction is reported by Dijkstra in [34].

More in general, according to Dijkstra [37] through abstract machines the hardware resources are provided at upper levels in a more user-convenient way. Such an approach helps with the construction and validation of software systems and is studied in detail

in [34, 35, 36] and in [82].

5.2.2 Agent-Oriented Decomposition of Complex Software Systems

Jenningsin assumes in [65] that most complex software systems are near-completely decomposable. He introduces an agent-oriented decomposability for analyzing, designing and implementing complex software systems. The system becomes a collection of interacting agents.

According to [65], adopting an agent-oriented approach to software engineering means decomposing the problem into multiple, autonomous agents that can act and interact in flexible ways to achieve their set of objectives. An agent is identified by a problem-solving entity with well-defined boundaries and interfaces. It is located in a particular environment in which it is partially controlled and observed. Organizational relationships exist between the agents.

Modeling and managing the relationships among agents is used to deal with the dependency and interactions in complex systems. The concept of independent agents is orthogonal to the concept of an efficient computation and is very far from the typical master-slave setting observed in high performance computing. Therefore we do not follow this line of research in this thesis.

5.3 Near-Complete Decomposability of Distributed Computing Systems

5.3.1 Near-Complete Decomposability in our View of MW and MW^{-1}

The inverted middleware framework is used side by side with the middleware layers and inverts middleware functionality with respect to performance data. Because of the near-complete decomposability theory, we can break up distributed computing systems into layers. Each layer strongly resembles the concept of chaining in mathematical functions. The structure of the inverted middleware itself resembles a chain of mathematical functions which correspond to the reversely ordered functionality of the middleware layers. This approach is viable as long as the distributed computing system under investigation shows the property of near-complete decomposability as stated by Courtois [30, 31]. Holt [60] provides useful criteria for checking the system decomposability according to the theory of Courtois:

- the complexity of the interactions between layers should be less than the complexity of interactions within the internal structure of layers,
- the functional decomposition of the system should precisely follow the dynamic behavior of the system.

A system whose decomposability into layers follows the Holt criteria is likely to belong to the group of well formed systems in terms of near-complete decomposability.

We take these observations of Holt and apply them to order the hardware and software components of the distributed system into layers. We have found a three-level hierarchy of abstraction where the layers are: the operating and network system layer, the middleware layer and the application layer (see Figure 5.2).

The interactions between middleware layer and operating system take place by means of system calls supported by run-time libraries within the operating system APIs. The intensity of the interactions between application and middleware layer is defined by the frequency of calls within the application to the middleware API. The degree of interaction between application layer and middleware layer as well as between middleware layer and operating system layer is smaller than the degree of interaction within the layers. Moreover, the functional decomposition of the system follows the dynamic behavior of the distributed system for which the application layer is user-specific domain and the middleware provide the service links to the operating system on top of the hardware system.

Some highly experienced computer scientists have stated the near-complete decomposability for complex hardware and software systems and used this property of those systems for stepwise facilitating system design and analysis. In Section 5.2 we have taken a brief look at the work of Dijkstra and Jenningsin in system software. Courtois used his theory for the successful analysis of stochastic networks of interconnected queues under the pre-condition that queued networks enjoy the property of near-complete decomposability. In another application of its theory, Courtois evaluated the computer system performance by looking at the memory system demands of these software systems and by a detailed analysis of an aggregative model of such a computing system [30].

5.3.2 Designing Distributed Systems in Hierarchical Levels of Abstraction

The choice of the decomposition of a computing system into layers related to the level of resource abstraction may be quite arbitrary. Decomposing a system into levels of resource abstraction (e.g. into two-levels, three-levels, multilevel hierarchy of abstractions) may become a difficult task. Different levels of abstraction for performance analysis are possible. The closer the layer is to the hardware, the more direct is the relationship between machine resources and the performance information. In the lowest layer, the control over the resource demand is less abstract and we are dealing with reservation and utilization of the machine resources.

Each resource at a certain level of abstraction is related to some lower level resources with lower level of abstraction and is characterized by a resource-usage directly related to an aggregate of response variables. Figure 5.2 shows an example of levels of abstraction for a resource (i.e. CPU usage) for our hierarchical aggregation of the system. The picture shows how the resource usage may be measured in different ways at different levels of

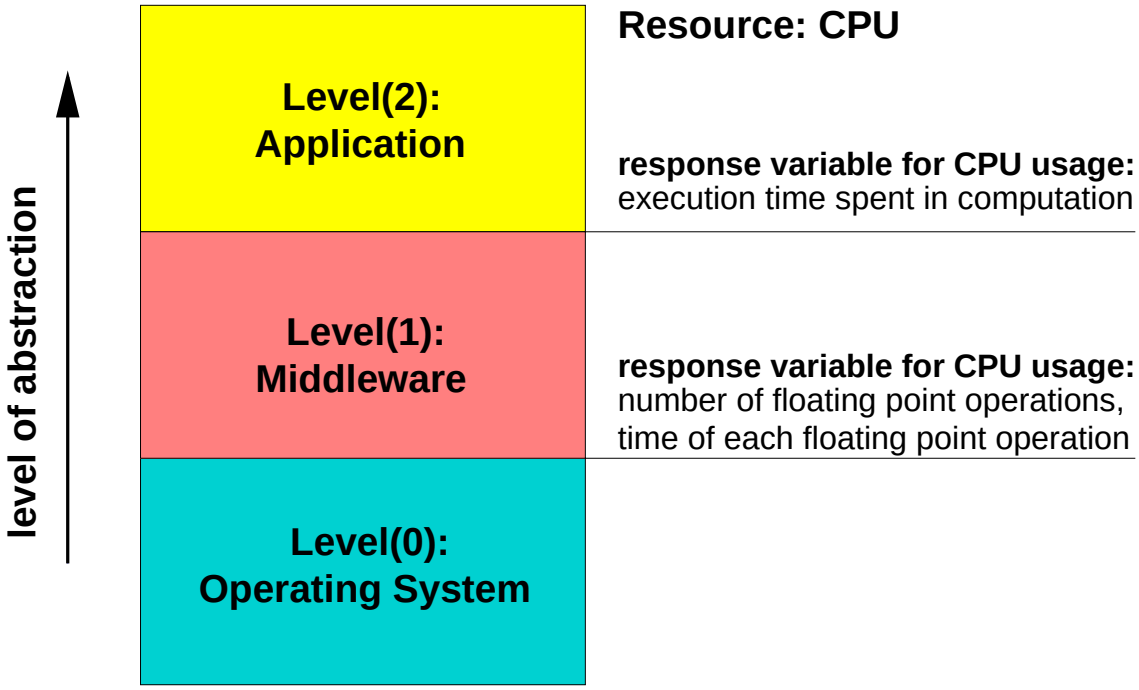

***Figure 5.2**: Example of levels of abstraction for a resource (i.e. the CPU) in a complex computing system.*

abstraction. The use of abstracted resources on top of the middleware layer should not prevent the machine resources from being properly controlled and efficiently used [30]. In order to maintain an efficient usage of the machine resources, the middleware should make use of abstracted resources at least as efficiently as the operating system layer does. In general, for an efficient allocation of the resource usage and a proper control of the resources, any middleware should be characterized by fast access to the abstracted resources which approximates the control speed of the same resources at the underlying layer (i.e. the operating system).

Unfortunately, it often happens that the middleware layer introduces overheads when it abstracts the resources. At the same time, the middleware layer with its abstractions might induce indirect hindrances to detect the more detailed resource behavior at the lower layer. In such a situation, the performance data gathered at a high level of abstraction, e.g. the execution time spent for the computation, is no longer representative of the workload distribution at the lower level of abstraction (i.e. the operating system layer) and ceases to provide any information to detect the non-effective allocation of the resource usage by the operating system itself. On the other hand, a comparison between response variables at different layers (e.g. on top of middleware and on top of the operating system) can show how the middleware affects the resource usage with its abstraction. Monitoring of response variables at different levels of abstraction helps to identify some faulty information about the usage of the underlying machine resources and to detect the

layer responsible for the loss of performance.

To analyze the performance of our hierarchical system and the delays introduced by the middleware, it is essential to dynamically maintain a subset of information at a higher layer (application layer) which would otherwise be lost with the decrease of the rate of interactions and the increase of the abstraction.

5.3.3 Horizonal and Vertical Decomposition

Modeling the dynamic behavior of a computer system helps us to gain insight and conceptual clarity on the issue of resource usage. We use *horizonal decomposition* into layers of a system to facilitate the design of computer system analytical models for real applications and *vertical decomposition* of performance data into resource specific metrics to obtain results showing performance problems.

We look at *horizontal decomposability* of the system and try to model some of its parts for studying solutions to the loss of control in terms of performance and software efficiency introduced by middleware packages in distributed systems. We transform the hierarchical structure of distributed computing systems into a hierarchy of levels of abstraction for the machine resources and we propose the aggregation of the computer system hardware and software architecture in three main layers: the application layer executing the application, the conventional middleware layer running on each node and the computing system layer including the operating and network system. We have a hierarchical organization of the system according to the degree of abstraction of its components. We range from a low level of resource abstraction at the operating and network system to high level of abstraction for the resources at the application layer. With this horizontal decomposition, we can study whether a hierarchical computing system controls and allocates the hardware resources in an efficient and reliable way.

Besides the horizontal decomposition of the system into layers, we propose a *vertical decomposition* of the system performance into components directly related to a set of critical resources. We break the modeling of the total execution time of the application under investigation into a number of execution time components each one correlated to the usage of a critical performance resource.

If we assume that the overall performance picture is also near-completely decomposable, we can evaluate the impact of the single resource within the system in an isolated way, i.e. as if the other resources are not existing. In the analysis of applications in Chapter 6, 7 and 8, we apply such a concept of confounding factors to the study of resource usage [62].

As soon as the computing system is no longer near-completely decomposable, our approach starts to fail. We have found cases for which the vertical near-complete decomposability is not fulfilled for some queries of the TPC-D Benchmark. For such cases, the inverted middleware cannot provide accurate performance information and can no longer

identify the loss of performance to a single critical resource. These cases are presented in detail in Chapter 8.

6

Conventional Performance Analysis

In this chapter, we consider the application package Opal for studying performance analysis by means of a conventional approach to performance monitor re-engineering and tuning of a code chosen as an example. We present an analytical model of computational complexity to predict the execution time for Opal simulations with different input parameters. We calibrate the model with a systematic experimental design and show what we learned from detailed analysis of computation and communication performance. We use the model together with some architectural key data to predict the efficiency of Opal on modern platforms including clusters of commodity PCs. The problems using such a performance tuning by-hand are entered as a plea for using our inverted middleware framework for more systematic performance analysis of applications.

6.1 The Scientific Computation Application Opal

Opal performs the simulation of the molecular dynamics of proteins and nucleic acids in vacuum or in water through energy minimization [5, 101, 103]. As a representative of an entire class of scientific codes (e.g. CHARMM, GAMESS, AMBER), Opal distributes the computation among different processors in a cluster of PCs or on a computational grid. Opal relies on a master-slave setting. The code is characterized by some computation intensive phases followed by a communication phase.

Opal relies on classical mechanics, i.e. the Newtonian equations of motion, to compute the trajectories $\vec{r}_i(t)$ of n atoms as a function of time t. Newton's second law expresses the acceleration as:

$$m_i \frac{d^2}{dt^2}\vec{r}_i(t) = \vec{F}_i(t), \tag{6.1}$$

where m_i denotes the mass of atom i. The force $\vec{F}_i(t)$ can be written as the negative gradient of the atomic interaction function V:

$$\vec{F}_i(t) = -\frac{\partial}{\partial \vec{r}_i(t)} V\big(\vec{r}_1(t), \ldots, \vec{r}_n(t)\big).$$

A typical function V has the form:

$$V(\vec{r}_1,\ldots,\vec{r}_n) = \sum_{allbonds} \frac{1}{2}K_b(b-b_0)^2 + \sum_{allbondangles} \frac{1}{2}K_\Theta(\theta-\theta_0)^2 + \sum_{improperdihedrals} \frac{1}{2}K_\xi(\xi-\xi_0)^2 + \sum_{dihedrals} K_\varphi\big(1+\cos(n\varphi-\delta)\big) + \sum_{allpairs(i,j)} \Big(\frac{C_{12}(i,j)}{r_{ij}^{12}} - \frac{C_6(i,j)}{r_{ij}^6} + \frac{q_i q_j}{4\pi\varepsilon_0\varepsilon_r r_{ij}}\Big).$$

The first term models the covalent bond-stretching interaction along bond b. The value of b_0 denotes the minimum-energy bond length, and the force constant K_b depends on the particular type of bond. The second term represents the bond-angle bending (three-body) interaction. The (four-body) dihedral-angle interactions consist of two terms: a harmonic term for dihedral angles ξ that is not allowed to make transitions, e.g. dihedral angles within aromatic rings or dihedral angles to maintain chirality, and a sinusoidal term for the other dihedral angles φ, which may make 360° turns. The last term captures the non-bonded interactions over all pairs of atoms. It comprises the van der Waals and the Coulomb interactions between atoms i and j with charges q_i and q_j at a distance r_{ij}.

A first sequential version of Opal, Opal-2.6, was developed at the Institute of Molecular Biology and Biophysics at ETH Zurich [76]. It was written in standard FORTRAN-77 and optimized for vector supercomputers through a few vectorizable loops. In the sequential version of the code of Opal-2.6 a single processor runs the whole computation. Opal-2.6 spends most of the computing time during a simulation evaluating the non-bonded interactions over all pairs of atoms of the molecular system (the last term of the atomic interaction function V). Fortunately, these calculations also offer a high degree of parallelism in addition to the vectorizable inner loops.

The parallel version of Opal [5, 101] distributes its work among multiple processors in a master-slave setting: multiple slaves share the computation of the Van der Waals and Coulomb forces while one master computes the few remaining interactions and coordinates the work. The computation repeats for every time step. For a molecular complex [1] of n atoms, the number of non-bonded interactions between atoms, which must be evaluated, is of the order of n^2. In the new version of Opal, this sequential complexity of the molecular energies evaluation is reduced by neglecting many of the non-bonded interactions from the molecular energy computation: only the pairs of atoms, whose distance is less than a *cut-off* parameter, are taken into account. At first, the data describing the non-bonding interaction parameters between the solute-solute, solute-solvent, solvent-solvent atoms pairs are replicated on all the slaves. This global information, whose volume depends on the problem size and does not scale with the number of processors, allows each

[1] A molecular complex is the combination of a solute such as proteins or nucleic acids and a solvent, e.g. water.

slave to achieve a large independence. With its data, each slave runs its tasks of the simulation requesting no further parameters at each step from the master than the atom coordinates.

A simulation proceeds by repeating the same computation tasks (i.e. simulation step) continuously. At the end of each simulation step, the information about the total energy, volume, pressure and temperature of the molecular complex is displayed. In the first stage of each simulation step, which we call *update phase*, each slave selects a distinct subset of the atom pairs, checks the distance among the atoms of each pair and adds the pair to its own *list of all active pairs* when the atoms are not beyond the given distance *cut-off*. In the second stage of the simulation step, the slaves compute partial non-bonded energies (Van der Waals energy and Coulomb energy) using their *list of all active pairs*. At the end of this stage, each slave sends its partial results to the master which gathers them and sums the total molecular energy of the molecular complex as well as its volume, pressure and temperature. The data in each list are updated periodically. The interval between successive updates can be selected by the user through the setting of an Opal parameter called *update*. The value of the *update* parameter expresses the number of interaction steps after which all the *lists of all active pairs* are updated.

The distribution of the atom pairs for the evaluation of the energies (Van der Waals energy and Coulomb energy) due to the non-bonded interactions is done using a *pseudo-random strategy*. Randomization should help to balance the workload among the slaves and to avoid duplication of work.

6.2 Modeling the Performance Tuning By-Hand of Opal

In this section, we discuss the most relevant issues related to the performance tuning by-hand of the scientific computation code Opal using decomposability of the complex system into layers and removing the overlap of communication and computation in the code by means of synchronization barriers provided in the middleware layers.

6.2.1 Multi-Level Middleware Layers

The master-slave setting of parallel Opal runs on Sciddle [7], a highly portable communication library and a remote procedure call (RPC) system extension to the PVM communication library. It has been ported to LINUX PCs, UNIX workstations, the Intel Paragon, and supercomputers like the Cray J90 and the NEC SX-4.

Sciddle comprises a stub generator (the Sciddle compiler) and a run-time library. The stub generator reads the remote interface specification, i.e. the description of the subroutines exported by the slaves, and generates the corresponding communication stubs. The stubs take care of translating an RPC into the necessary PVM message passing primitives. The application does not need to use PVM directly for RPCs: the master simply calls a

subroutine (provided by the master stub), and the Sciddle run-time system invokes the corresponding slave subroutine (via the slave stub). However, Sciddle allows the process management (starting and terminating of slaves) directly with PVM calls.

Unfortunately, none of the two middleware layers, neither PVM, nor the run-time library of Sciddle, do indeed provide adequate support for detailed quantification of the resource usage or any other performance related variable such as total computation time and communication time. Therefore we decompose the system and monitor the code by means of instrumenting each layer of the middleware packages by-hand and by means of providing the application source with performance monitoring hooks. Figure 6.1 shows the multilevel architecture (Sciddle+PVM) of Opal's middleware and the layers considered.

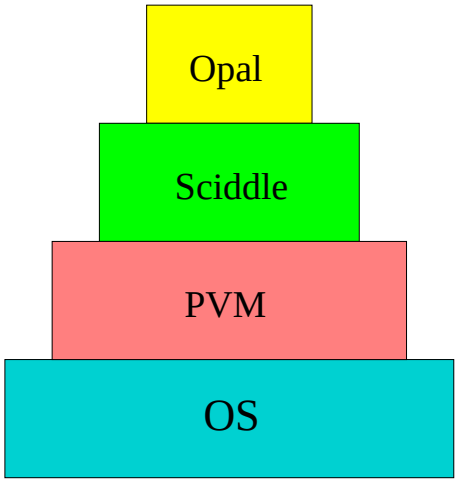

***Figure 6.1**: Middleware architecture of Opal application.*

6.2.2 Experimental Setup of the Performance Study

For an accurate performance characterization on several platforms for parallel and distributed computing, we apply an accurate and systematic method for the performance analysis. In particular, we define the relevant experimental factors and response variables. We also derive an *analytical model* based on these factors and variables.

Experimental factors

The parameters directly related to the application are called *application factors*. The application factors are: the number of simulation steps s, the number of slaves p on which the Opal application runs, the frequency of the list updates u, the number of mass centers (atoms and water molecules) of the whole molecular complex n, the average number of neighboring atoms considered for their total energy calculation $\tilde{n}$ which is a function of

the cut-off radius c and the volume density of the molecule and last, but not least, the the ratio of number of water molecules to the total number of mass centers γ.

The parameters relevant to the platforms (i.e. the parallel machines) are called *platform factors* and are:

- the communication rate a_1 measured in MByte/s,
- the communication overhead b_1 measured in seconds,
- the computation rate in MFlop/s which is indirectly obtained by a weighted sum of:
  - the computation time spent to generate a pair of atoms and calculate the distance between them, a_2, in seconds,
  - the time spent to compute the non-bonded energy contribution of a single pair of atoms, a_3, in seconds,
  - the time needed for each atom of the molecular complex to evaluate its bonded interactions, a_4, in seconds,

  over the number of floating point operations counted by our hardware monitors,
- the time to synchronize processes b_5 in seconds.

Response variables

We spend a considerable amount of effort and time to instrument Opal for an accurate allocation of the time spent in computation, in communication or with overhead. We decided to opt for a tightly synchronized execution model and introduced some PVM barriers into the application code by-hand. Because of these PVM barriers, we could estimate the fraction of time in which each single slave is busy servicing requests (*parallel computation time*) or remained unused (*idle time*), the time the master is busy servicing the slave calls (*sequential computation time*) and the time during which the master requests and the slaves respond or vice versa (*communication time*).

The introduction of the physical barriers was done at the cost of loosing some overlap between computation and communication and with an overhead of additional synchronization (*synchronization time*). Still for the case of Opal, we have proven that the slowdown due to this synchronization concept and due to extra barriers was less than 5%. This is a small expense given that we achieve a lot of clarity and reproducibility. However, the introduction of additional barriers for synchronization would no longer comply with the requirement of a “black box approach” to performance tuning with inverted middleware. There are numerous code changes in parts of the application code and in Sciddle (one layer of the two middleware) layers. The changes require a deep knowledge of the code by the performance analyst and could only be done because the authors of the Sciddle

system were in-house. In the case of a so called black box middleware, the middleware is either not accessible for the user or even too complex to be modified.

6.2.3 Modeling Design

In a second phase of our performance study using tuning by-hand, we derive and validate a simple analytical model of the execution time of Opal which incorporates the key technical data of several parallel machines [103]. This model can be used for workload characterizations on existing computing systems and performance predictions of Opal on future platforms.

During the design and parallelization of Opal, we derived an analytical time-complexity model that captures all essential parameters of the real application. The predicted outcome of the model is the execution time of Opal in seconds, written as a sum of several partial result variables which are computed separately and also measured separately during validation:

$$t_{Opal} = t_{tot\_par\_comp} + t_{tot\_seq\_comp} + t_{tot\_comm} + t_{tot\_sync}$$

Parallel computation time

The total parallel computation time, $t_{tot\_par\_comp}$, is the computation time spent by the slaves servicing the request for the computation. Each slave runs two routines: *the update routine* that updates the list of atom pairs, and the *energy evaluation routine* that evaluates the partial energies of the non-bonded interactions (i.e. Van der Waals energy and Coulomb energy) using the same list of atom pairs.

$$t_{tot\_par\_comp} = t_{update} + t_{nbint} \tag{6.2}$$

The computation time of the update routine always grows quadratic with problem size because each time the slaves update their own list, all the pairs of atoms must be checked. At the same time, the update time decreases linearly with the increase of the time interval between two list updates.

$$t_{update}(n,\gamma) \approx a_2 \frac{su}{p} \frac{(1-2\gamma)^2 n^2 - (1-2\gamma)n}{2} \tag{6.3}$$

where:

- s is the number of simulation steps.
- p is the number of slaves on which run the Opal application.
- u is the frequency of the list updates (update parameter).
- n represents the number of mass centers (atoms and water molecules) of the whole molecular complex.

- γ (gamma) is the ratio of number of water molecules to the total number of mass centers.
- a_2 represents the computation time spent to generate a pair of atoms and calculate the distance between them.

On the other hand, the time for the energy evaluation routine is is affected by the cut-off parameter: the length of the list of atom pairs increases drastically with the increase of the cut-off distance. The energy-evaluation routine grows quadratically up to the number of atoms within the cut-off radius and linear beyond that.

$$t_{nbint}(n,\tilde{n}) \approx \begin{cases} a_3 \frac{s}{p} \frac{n(n-1)}{2} & \text{when } n < \tilde{n} \\ a_3 \frac{s}{p} \tilde{n} n & \text{when } n > \tilde{n} \end{cases} \tag{6.4}$$

where:

- $\tilde{n}$ is a function of the cut-off radius and the volume density of the molecular complex. The meaning of $\tilde{n}$ is is the average number of neighboring atoms considered for their total energy calculation.
- a_3 is the time needed to compute the non-bonded energy contribution of a single pair of atoms.

The energy evaluation entirely dominates the parallel computation time when $n < \tilde{n}$:

$$t_{nbint} >> t_{update}$$

The introduction of the cut-off parameter reduces the amount of the computation spent during the energy evaluation routine when $n > \tilde{n}$. But despite its lower asymptotic complexity due to the introduction of a cut-off parameter, the energy evaluation routine still dominates the update process in this second case for all practical problem sizes. The crossover point in n, for which the update time equals the energy evaluation time, depends on the molecular complex volume as well as the cut-off parameter. For our simulations, crossover happens for unrealistic numbers of water molecules or protein atoms i.e. for sufficiently high values of n. Furthermore, by a reduction of the update frequency it is possible to decrease the fraction of update computation arbitrarily and to restore the relation of:

$$t_{nbint} > t_{update}$$

We summarize the total parallel time as:

$$t_{tot\_par\_comp} \approx \begin{cases} s\left(\frac{1}{2p}\left(a_2 u(1-2\gamma)^2 + a_3\right)n^2 - \frac{1}{2p}\left(a_2 u(1-2\gamma) + a_3\right)n\right) & \text{when } n < \tilde{n} \\ s\left(\frac{1}{2p}\left(a_2 u(1-2\gamma)^2\right)n^2 + \frac{1}{p}\left(a_3\tilde{n} - a_2 u\frac{(1-2\gamma)}{2}\right)n\right) & \text{when } n > \tilde{n} \end{cases}$$

Sequential computation time

The total sequential computation time, $t_{tot\_seq\_comp}$, is the total time spent by the master to compute the energy-, pressure-, volume-, and temperature values of the molecular complex from the partial energies and forces computed in the parallel step. It depends on the number of steps of the simulation and on the number of atoms of the molecular complex:

$$t_{tot\_seq\_comp} = a_4 sn \tag{6.5}$$

where a_4 is the time needed for each atom of the molecular complex to evaluate its bonded interactions.

Communication time

The total communication time, t_{tot_comm}, is the time spent by the communication processes between the master and the slaves during the entire simulation. The master calls two different kinds of procedures (subroutines) that are run on the slaves: the subroutine for list updates and the subroutine for energy evaluation. We enhanced the communication environment with some synchronization tools that allow us to separate the communication times properly from other computation and idle times and therefore allow us to explain all communication components precisely. More details about these synchronization tools and their underlying model are explained in [101, 103]. Thanks to this model, the resulting communication time of the master's RPCs can be decomposed into:

$$t_{tot\_comm} = t_{call\_upd} + t_{return\_upd} + t_{call\_nbi} + t_{return\_nbi} \tag{6.6}$$

In addition to a constant overhead, the communication time spent by the master in sending the atom coordinates necessary for the list update phase and the energy computation in the slaves, is linear in the problem size n.

$$t_{call\_upd} = t_{call\_nbi} = \frac{\alpha}{a_1} n + b_1 \tag{6.7}$$

in addition to the quantities defined above we define:

- α (alpha) is the number of bytes used to represent the coordinates of a single atom.
- a_1 is the communication rate including the overhead in the communication environment (Sciddle and PVM)
- b_1 is the communication overhead, in seconds, used to transfer an empty block from the sender to the receiver

For the update RPC, the master does not retrieve any data from the slaves when they arrive at the end of the update routine: the master just waits for a result message which assures the end of the slave tasks.

$$t_{return\_upd} = b_1 \tag{6.8}$$

On the other hand, the amount of data returned by each slave to the master upon termination of the energy evaluation routine comprises the Van der Waals and Coulomb energies, and the gradients of the atomic interaction potential.

$$t_{return\_nbi} = 2\frac{\alpha}{a_1} + \frac{\alpha}{a_1}n + b_1 \approx \frac{\alpha}{a_1}n + b_1 \tag{6.9}$$

Altogether, the total communication time of equation (6.6), can be rewritten as:

$$t_{tot\_comm} = s\left(p\frac{\alpha}{a_1}(u+2)n + 2pb_1(u+1)\right)$$

Synchronization time

The total synchronization time, t_{tot_sync}, is the time to synchronize the processes with each other. t_{tot_sync} is the sum of four terms:

- t_{str_upd}, the total time to synchronize the master and the slaves when the update routines are called,
- t_{end_upd}, the total time to synchronize the master and the slaves when the update routines finish,
- t_{str_nbi}, the total time to synchronize the master and the slaves when the energy evaluation routines are called,
- t_{end_nbi}, the total time to synchronize the master and the slaves when the energy evaluation routines finish.

We assume that the four different synchronization times do not depend on the number of slaves nor on the problem size. Our formulation just states that each term increases linearly in the number of simulation steps and that the contribution of t_{str_upd} and t_{end_upd} decreases as the update parameter decreases. We assume that each synchronization process takes a constant time b_5.

$$t_{tot\_sync} = sub_5 + sub_5 + sb_5 + sb_5 = 2s(u+1)b_5 \tag{6.10}$$

More details about the synchronization model and the tools developed, are reported in [101].

6.2.4 Calibration between Analytical Model and the Measurements

In order to verify the analytical model against the implementation, we compare the measured execution time on the Cray J90 with the computed values of the model for the same machine.

Figure 6.2 displays the space of factors that we have considered during calibration of our model. We have chosen three different molecular complexes, each one with a different

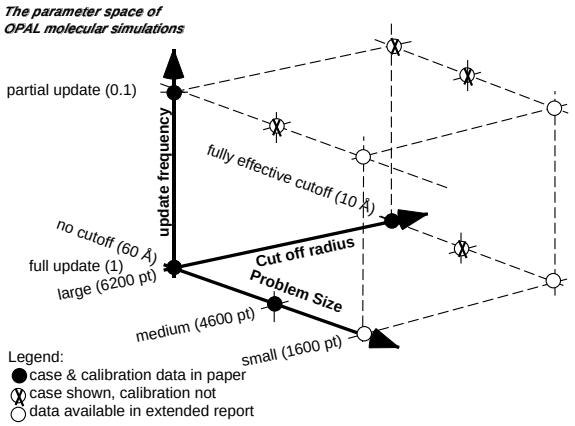

***Figure 6.2**: Model of the space of factors for Opal molecular simulation.*

problem size: small, medium, large size. Furthermore, we have run the simulation of each molecular complex picking the update frequency and the cut-off parameter at the two extreme values: a list update each ten simulation steps (partial update) versus a list update at each simulation step (full update) and a simulation without cut-off parameter (i.e. all the atoms interactions are considered) versus a simulation with cut-off parameter (i.e. for each atom only interactions within the range of 10 Å are considered). At the same time we have computed the outcome of the simulation through the analytical model for each one of these cases and adjusted the parameters for a last square fit to the corresponding measurements.

Figures 6.3(a)-(d) show the comparison of the wall clock times measured for the execution on a Cray J90 SMP against the times predicted by the analytical model for the same machine with different numbers of slaves, different cut-off radii, update frequencies and large or medium molecular complexes. For brevity, we list only the data of a reduced $(7 \cdot 2^{3-1})$-design although the data was achieved with a full factorial design of 84 experiments. During the calibration the differences between model and measurement have been investigated and plotted for each case. The overall fit of the model to the measurement for the cases in Figures 6.3(a)-(d) and for the remaining cases is excellent. The full data is listed in a more detailed technical report [101].

6.3 Performance Analysis of Opal using By-Hand Tuning

The analytic complexity model and the careful instrumentation for performance monitoring leads to the characterization of the workload and to discover interesting performance issues and anomalies such as:

- understanding the components of the execution time,
- the load imbalance for even number of slaves,

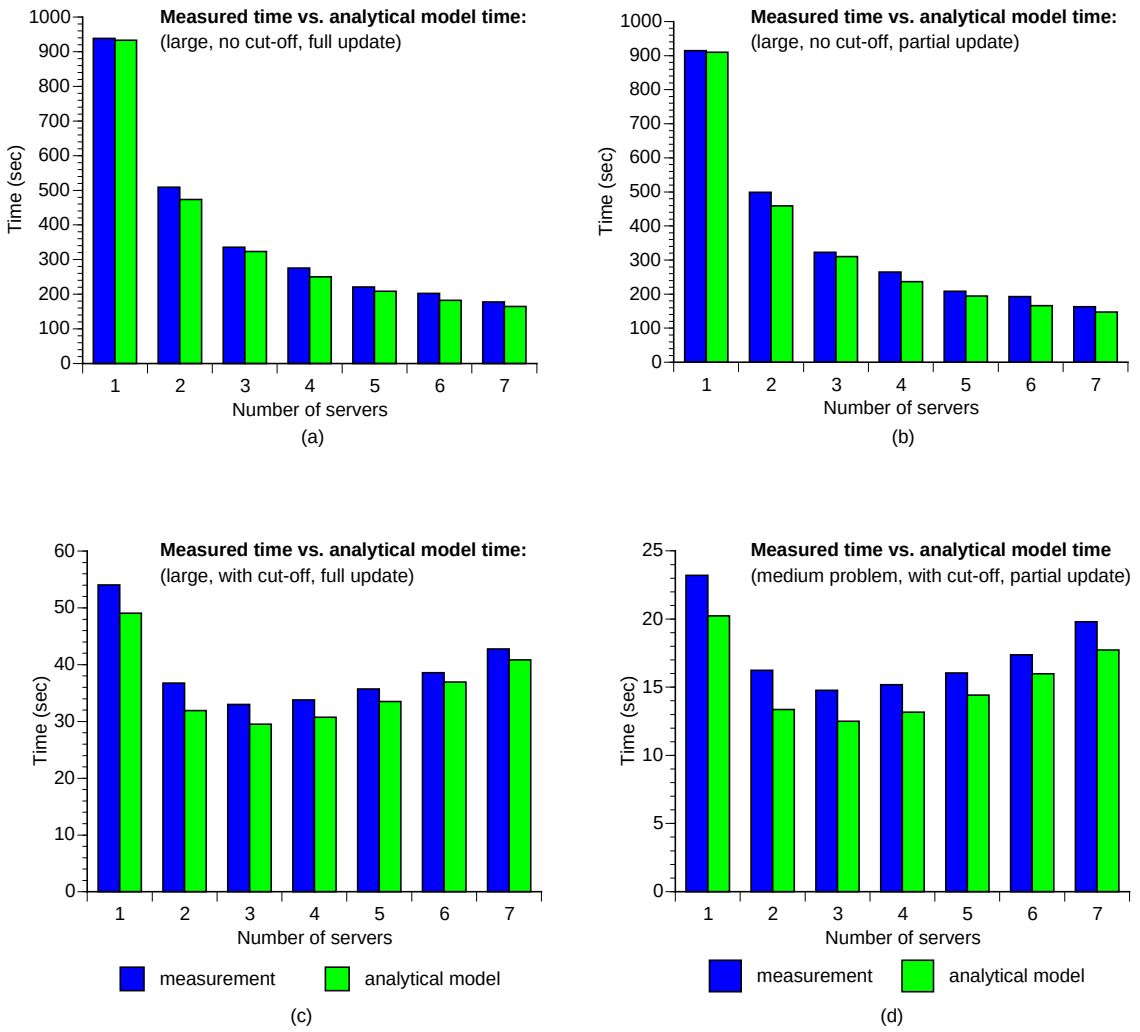

***Figure 6.3**: The difference between the wall clock times measured and the times predicted by the analytical model.*

- the differing number of floating point operations for different kind of processors
- several communication speed anomalies.

Moreover, we can use the calibrated model for predicting with certainty how the application would run on different platforms and for different levels of the chosen factors.

6.3.1 Workload Characterization of Opal

We investigate the performance of the Opal code by measurements of simulation execution times (i.e. the parallel computation time, the sequential computation time, the communication time, the synchronization time, and the idle time) of two molecular complexes with different sizes. We measure the detailed breakdown of the wall clock execution time for ten simulation steps.

The first molecular complex is a medium size example of the simulation problems that Opal can handle: it is the complex between the Antennapedia homeodomain from Drosophila and DNA [45], composed of 1575 atoms and immersed in 2714 water molecules or a total of 4289 mass centers (*medium problem size*). The second molecular complex is considered to be a large size problem: it is the NMR structure of the LFB homeodomain, composed of 1655 atoms and immersed in 4634 water molecules, a total of 6289 mass centers (*large problem size*).

We run the code for different levels of parallelism: the number of slaves ranges from one to seven. At the same time, we measure the execution times when the simulation is fully accurate and the computation complexity is a quadratic function of the protein size (i.e. *no cut-off*) and when the simulation is approximate and consequentially the computation complexity becomes linear (i.e. *with cut-off*). Finally, we investigate the role of the list update: we run the simulation either with an update of our lists upon every iteration (*full update*) or with a partial update every 10 iterations (*partial update*). The study of the different cases permits us to investigate the precise impact of the problem size, the frequency of the lists update and the values of the cut-off parameter on the performance of the simulation.

Figures 6.4(a)-(d) display a detailed breakdown of the wall-clock execution time for 10 simulation steps in the medium size molecular complex with different choices for the number of slaves, the cut-off and the update parameters. The chart in Figure 6.4(a) shows that without cut-off, the time in parallel computation is the largest fraction of the execution time and that it decreases as expected when more slaves are added. At the same time, the communication time increases almost linear with the number of slaves, but its overall contribution remains small, even for seven slaves. The synchronization time and the sequential computation time remain insignificant in the overall execution time. To the surprise of the Opal implementors, our instrumentation reveals a load balancing problem for runs with an even numbers of processors. Figure 6.4(b) shows an Opal execution

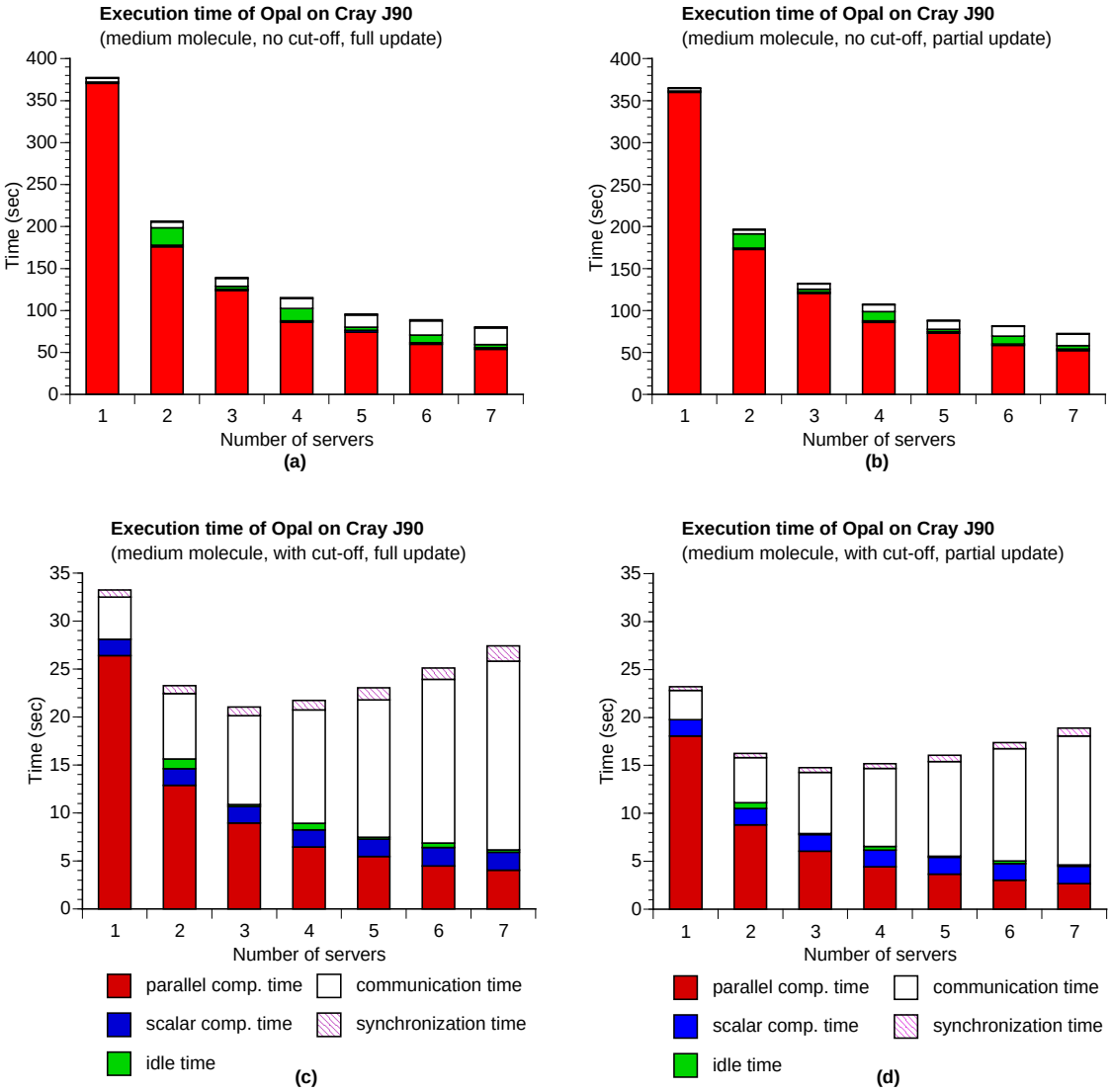

***Figure 6.4**: Detailed breakdown of the measured execution times for 10 iterations of an Opal simulation with a medium molecule.*

with reduced updates. As expected, the lower update frequency does not much affect the overall performance on simulations because the large amount of parallel computation dominates the execution time. Figure 6.4(c) shows a simulation with an effective cut-off parameter (at 10 Å). The cut-off parameter determines the asymptotic computational complexity: the amount of the parallel computation is smaller than in the cases above and its overall contribution becomes comparable to the other measured execution times. The sequential computation time, the synchronization time and the communication time gain a high importance for the overall performance. Figure 6.4(d) displays a run with both the cut-off and the partial update option in effect. The frequency of the list updates leads to a notable difference in the performance of simulations with small cut-off radii.

The problem size (the number of atoms of the whole molecular complex) has a varying impact on the different components of the execution time. The time components (i.e. the parallel computation time, the communication time, the idle time, the sequential computation time and the synchronization time) increase each one in a different way with the number of atoms: while the size of the problem has a super-linear impact on the parallel computation time and the idle time, it has only a linear moderate impact on the sequential computation time and the communication time. Figure 6.5(a)-(d) show the detailed breakdown of the wall clock execution time for 10 simulation steps in the large size molecular complex. While the order of the measured execution time doubles as we increase the problem size from a medium amount of mass centers to a large amount of mass centers, the behavior of the computation time components remains almost the same.

6.3.2 Performance Predictions for Alternative Platforms

We use our analytic model together with some standard performance data of alternative computer platforms to predict the performance of Opal for the case that we could port the code to that platform. Two different classes of MPP (Massively Parallel Multi-Processors) are considered for our study about suitable platforms in addition to the existing Opal version for the Cray J90: First, the Cray T3E, a “big iron” MPP and second, three different flavors of PC clusters called: *slow CoPs* (cluster of PCs), *SMP CoPs* and *fast CoPs*. We named the first PC cluster slow CoPs since it is optimized for lowest cost and gains its performance by a large number of slower nodes; weakly connected with a shared 100BaseT Ethernet medium, its uni-processors are some older Intel Pentium Pro PCs running at 200MHz. The SMP CoPs platform is based on similar Intel Pentium Pro processors, but in a twin processor configuration (2 x 200 MHz) and interconnected by a low latency SCI shared memory interconnect technology. Finally, the fast PCs cluster features some single 400MHz Intel Pentium Pro PCs as nodes, connected by a Gigabit/s communication system based on fully switched Myrinet interconnects. Comparable clusters of PCs installations are described in [8, 16, 95].

The parameters of our analytic model have been intentionally chosen in a way to

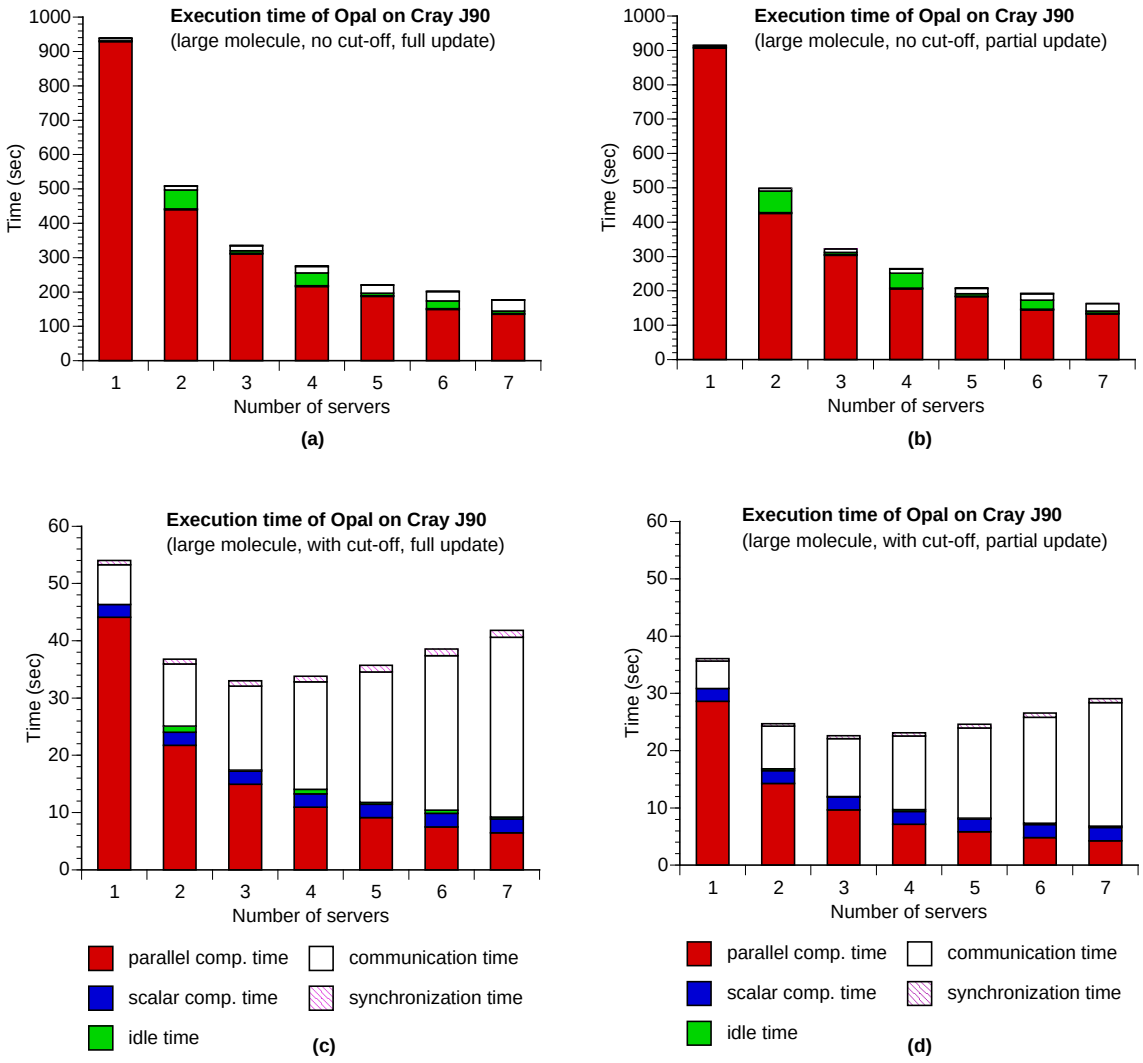

***Figure 6.5**: Detailed breakdown of the measured execution times for 10 iterations of an Opal simulation with a large molecule*

include all major technical data usually published for parallel machines. This includes among others: message overhead, message throughput for large messages, computation rate for SAXPY inner loops and the time to synchronize all participating processors. For each new platform, we determine the key parameters by the execution of a few micro-benchmarks, carefully verified against other published performance figures [46]. An overview of the data used is found in the Tables 6.1 and 6.2.

MPP Node Type (clock speed)	Execution Time on single node [s]	Floating Point op. counted on single node [MFlop]	Computation Rate on single node [MFlop/s]	Relative Time [%]	Adjusted Comp. Rate on single node [MFlop/s]
Cray T3E-900 (450 MHz)	9.56	811.71	85	138	52
Cray J90 (100 MHz)	6.18	497.55	80	100	80
Slow CoPs (200 MHz)	10.00	327.40	32	65	50
SMP CoPs (2*200 MHz)	5.00	327.40	65	65	100
Fast CoPs (400 MHz)	4.85	325.80	67	65	102

***Table 6.1**: Computation speed parameters used for performance prediction of Opal on all five different platforms. The numbers reflect the performance of the isolated Opal application kernel used as a micro-benchmark.*

MPP Node Type	MByte/s on single node (hw peak)	MByte/s on single node (observed)	Latency on single node (observed)
Cray T3E-900 (MPI)	350	100	12 μsec
Cray J90 (PVM/Sciddle)	2000/8	3	10 msec
Slow CoPs (Ethernet)	10	3	10 msec
SMP CoPs	50	15	25 μsec
Fast CoPs (Myrinet)	125	30	15 μsec

***Table 6.2**: Communication speed parameters used for performance prediction of Opal for all five different platforms. The numbers are obtained from micro-benchmarks and verified against published values*

We use these figures together with the formulas of the analytical model to predict

the execution time of Opal with a medium and large size molecular complex in varying configurations. Some model parameters are intrinsic to Opal itself and invariant across the different machines. For the discussions in this chapter these parameters are simply kept at their level measured with the J90. On the other hand, we have had to select the most important platform parameters, which depend on the new machines features. The communication throughput observed in MBytes/s listed in Table 6.2 is incorporated into the model as parameters a_1 and b_1. The model parameters a_2 and a_3 are obtained by computing the average total time of the following three phases: the generation of a pair of atoms, the calculation of the distance between the two atoms and the computation of the non-bonded energy contribution of the pair of atoms.

Since the model captures the parallelization and adjusts to different processor performance, we can calculate the estimated execution time and the relative speed-up achieved on each new platform and compare it with the measured speed-up achieved on a Cray J90.

As for many scientific codes one routine of Opal dominates the compute performance. This routine has been benchmarked on each platform using the most accurate hardware cycle counters and floating point performance monitoring hardware that is actually present on all five machine types. The most important surprise has been a significant difference in floating point operations for the different platforms although the arithmetic was 64 bit in all cases and the results were bitwise identical (or within a reasonable floating point epsilon for comparisons between Cray and IEEE arithmetic). The differences are due to the different compilers and different runtime libraries with intrinsics functions. We eliminate the effect of this difference by assuming that the best compiler (i.e. the PGI compiler for the PCs [50]) is setting quasi a theoretical lower threshold for the computation and we adjust the local computation rate (MFlop/s) of other platforms accordingly.

The communication performance is even more difficult to compare in real applications. Some unfortunate interactions between middleware and PVM library reduce the measured communication rate on the Cray J90 processor to about 3 MByte/s, despite a crossbar interconnect that can carry more than one GByte/s between the 8 processor boards and the corresponding memory banks. The authors of the Sciddle middleware claim that they measured a throughput up to 7 MByte/s for a synthetic Sciddle RPC example and that this rate matches just about the performance of raw PVM 3.0 on the same machine [6]. The performance delivered certainly remains far below what this machine is capable of in shared memory mode.

We suspect that with the proper configuration of PVM flags or with a rewrite of the Sciddle middleware to use MPI in true zero copy mode, we could significantly improve the performance of Opal on the J90, but such work is outside the scope of this performance study. For the new platforms, we assumed an MPI or PVM based re-implementation without Sciddle and deduced our performance numbers mainly from our MPI micro-benchmarks and from similar numbers published by independent researchers on the Internet (e.g. [46]).

The complexity model incorporates the key technical data of most parallel machines as parameters. Therefore it is well suited for performance prediction.

In the first two graphs of Figures 6.6(a)-(d), we look at the predicted execution times for ten Opal iterations with a medium size molecule. Platforms include the Cray T3E MPP, the Cray J90 vector SMP (reference) and three clusters of PCs (fast CoPs, SMP CoPs and slow CoPs). Since we also list the absolute execution time in seconds, we can directly compare the performance of all five platforms when 1-7 processors are used in Charts 6.6(a) and 6.6(c). The success or failure of the parallelization of the Opal code becomes most evident, when we plot a relative speed-up with 1-7 processors in the Charts 6.6(b) and 6.6(d). The well specified synchronization model guarantees that we are not subject to the pitfalls of a badly chosen uni-processor implementation.

In the upper Charts 6.6(a) and 6.6(b), the cut-off radius is too large to reduce computation and therefore the runs are largely compute bound. The execution time reflects the different compute performance of the processors with a slight edge for the SMP CoPs architecture which becomes slighter and slighter with the increase of the number of processors. An entirely compute bound operation inevitably leads to excellent speedup, as seen in Chart 6.6(b).

The main users of Opal in molecular biology assured us repeatedly that for certain problems a simulation with a 10 Å cut-off parameter is accurate enough to give new insights into the proteins studied. Consequently, we ran the second test case of the same molecule with a computation reduced by cut-off. In the lower two Charts 6.6(c) and 6.6(d) the computation is accelerated with an effective cut-off parameter and therefore gradually becomes communication bound as the parallelism increases. In this case the communication performance of the machine matters considerably. The Cray J90 and the slow CoPs (Ethernet) cluster of PCs are severely limited by their slow communication hardware or by their bad software infrastructure for message passing. This is visible in predicted execution times: as soon as the number of processors increases and exceeds the value of three, the overall execution time of the application on the Cray J90 and the slow CoPs (200MHz with Ethernet) is no longer decreasing but rather increasing. The increase of the communication time offsets any gain due to parallel execution and leads to an overall loss of performance for a larger number of nodes. This aspect is displayed by some speed-up curves in Charts 6.6(d) which actually turn into slow-down curves when too many nodes are added. For these two architectures we achieve no benefit in putting more than three processors at work.

For a small number of processors the SMP CoPs and Fast CoPs architectures start out with a better execution time than the big MPP and the vector SMP machines, possibly due to the better compiler. With the increase of the number of processors, the speed of the Cray T3E MPP catches up quite rapidly due to the better communication system. This trend is also evident in the speed-up curves where the Cray T3E architecture achieves better gain and almost ideal speed-up. For all platforms with a good communication

system, we can scale the application nicely to 7 processor with a speed-up of 4 or greater.

As we can see in the two Graphs 6.6(c) and (d), speed-up curves cannot be interpreted properly without looking at the absolute execution times simultaneously; while the Cray T3E MPP has by far the best speed-up, it still ends up behind fast CoPs and SMP CoPs when comparing absolute performance for seven slaves.

The same performance scalability relationships are reflected in the Figures 6.7(a)-(d) for a large size problem. The charts show predicted execution times and speed-ups for a large problem. A comparison between the Charts 6.7(a)-(d) and 6.6(a)-(d) shows how the behavior of the execution time remains quite similar to the medium size problem. At the same time, we notice that the increase of the amount of the computation for a large size problem leads to slightly better speed-ups in Chart 6.7(b). Still both charts indicate flat speed-up for more processors due to an overhead in the communication systems. In Chart 6.7(d) we do not have the extreme slow down seen in Chart 6.6(d), but we can conclude that the increase of the amount of the computation has just pushed the point of the break down further outwards on the curve. With a larger number of processors we would probably encounter the same saturation point at which adding processors would stop the increased performance.

6.4 A Plea for Inverted Middleware Framework in Complex Systems

6.4.1 Advantages of the By-Hand Performance Tuning Approach

We can state three different potential conclusions from our accurate by-hand performance tuning. First, an analytic complexity model and a careful instrumentation for performance monitoring leads to a much better understanding of the resource demands of a parallel application. We realize that the basic application without cut-off is entirely compute bound and therefore parallelizes well, regardless of the system. The optimization with an approximation algorithm using an effective cut-off radius changes the characteristics of the code into a communication critical application that requires a strong memory and communication system for good parallelization. Second, we discovered interesting anomalies in the implementation, e.g. the load imbalance for even number of slaves and the differing number of floating point operations for different processors. Third, we can use our model to predict with reasonable certainty how the application would run on *slow Cops*, *SMP CoPs* and *fast CoPs*, three low cost cluster of PCs platforms connected by Gigabit Networks, like SCI or Myrinet.

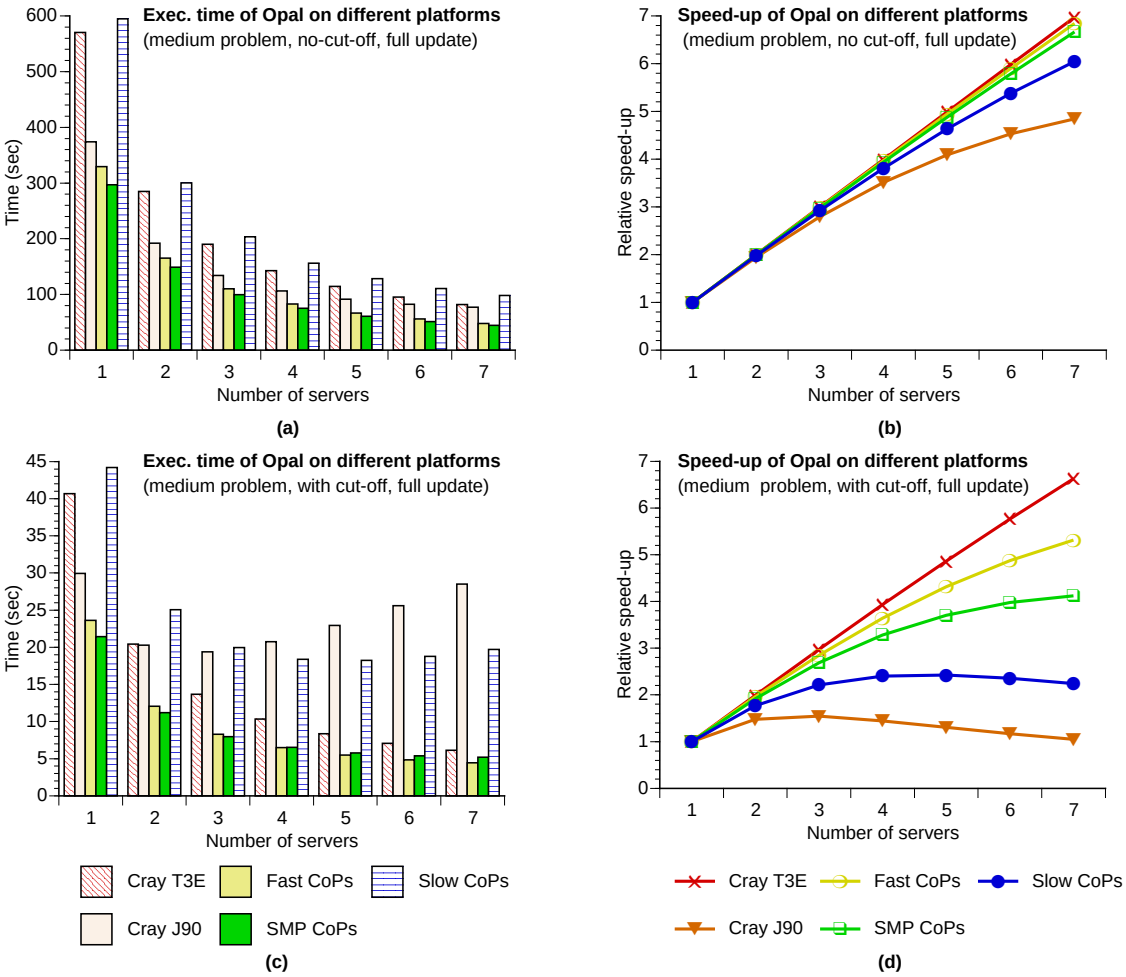

***Figure 6.6**: Predicted execution time for an Opal simulation of a medium problem size molecule.*

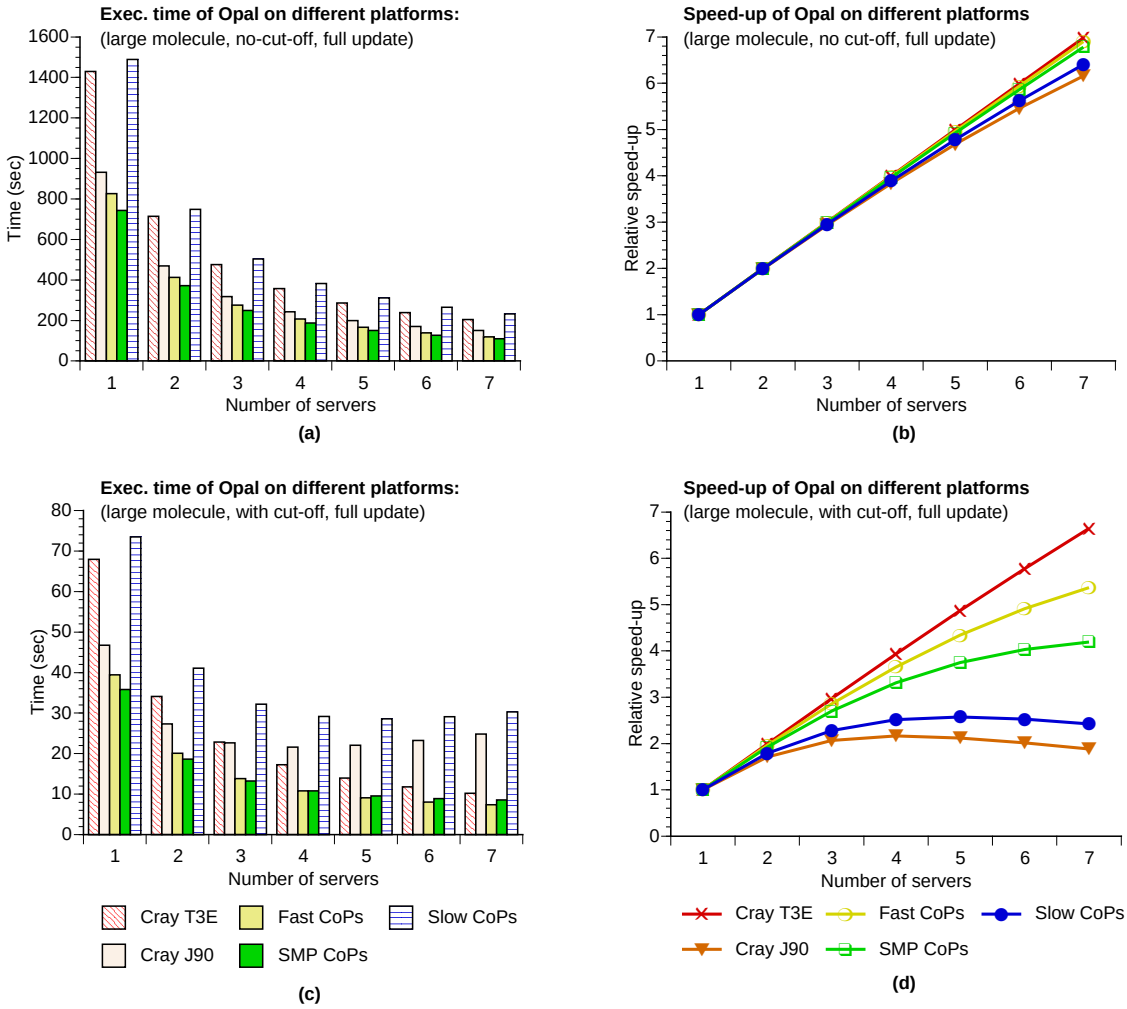

***Figure 6.7**: Predicted execution time for an Opal simulation of a large problem size molecule.*

6.4.2 Limits of the By-Hand Tuning Approach

However, our case study of Opal shows common problems with the performance instrumentation used and the by-hand tuning approach.

First of all, the application code and the middleware layers have to be instrumented with hooks and barriers for performance monitoring and the overlap of communication and computation had to be restricted slightly for the sake of a reliable accounting of execution times. The investigation with barriers is restricted to the time components but does not give any information about the usage of the most performance critical machine-resource that prevents good scalability and good performance. The by-hand tuning approach can be used as a performance method to indicate that performance is bad, but it cannot always state where and why the slow-down in performance happens. Moreover, instrumenting the middleware or the application source with performance monitoring hooks and introducing barriers has the disadvantage that this approach cannot be applied to complex layers, such as black box middleware systems whose source is unavailable or far too complex to be modified.

Sampling system performance data at a high level of abstraction provides a rough estimate for the compute rate in MFlop/s and is easy to use. However, the approach to measuring the compute rate has proven to be extremely complex to use while the information is extremely hard to understand in sufficient depth. Sampled computation rates are no substitute for the simple ratio of operations counted divided by the cycles used. The characteristic performance of Opal runs on different machines in Table 6.1 in Section 6.3.2 shows how difficult it is to measure accurate MFlop counts. The number of floating point operations to compute the same molecular simulation result significantly differs among the five different platforms. With a performance monitoring and tuning by-hand, we just believe the measured MFlop/s figures but we are not able to explain with certainty why the MFlop/s numbers did not make much sense. We can only guess that this is due to the vectorizing transformations and the different implementations for intrinsic functions like `sqrt()` and `exponentiate()`.

6.4.3 Motivation for an Alternative Approach to Performance Analysis

Our study of the performance of Opal on distributed computing systems shows some substantial drawbacks during the performance analysis using performance tuning by-hand which suggest the need for more accurate and reliable approaches to performance analysis.

Because distributed computing is considered for the sake of higher speeds of parallel execution, accurate and reliable performance engineering becomes a crucial problem. A precise understanding of performance and resource utilization is not just required for the tuning process but also for the prediction of scalability or viability of an application on new or alternative platforms. Most high performance computing applications need to

know whether they require the strong memory systems of traditional vector supercomputers or whether they can run on cheaper clusters of PCs. Similarly, database applications need to know if they only run on a symmetric multiprocessor with a single operating system image or if they can also run on a large farm of independent PCs. In the next two chapters we show how our inverted middleware framework is able to address these important performance questions even in distributed computing systems.

7

Predictions with MW^{-1} of Scientific Computation Applications

There exists a fair number of useful applications that were once brand-marked as embarrassingly parallel and therefore considered to be uninteresting. Indeed, the performance characteristics of these applications are quite boring if they are run on high-end supercomputers. Still, these applications do computations that are highly useful to the advancement of science once they can be deployed more widely and executed on cheaper resources. Furthermore, a closer look at some examples may quickly reveal that there are quite different levels of embarrassing parallelism and that not all of those codes can be migrated equally well to newer and more cost effective platforms.

In this chapter, we present a performance method to check in advance the viability of migrating an application from clusters of commodity PCs to a framework of widely distributed computing, by means of our inverted middleware instrumentation. We study the viability for a specific application, the highly parallel molecular biology code Dyana [52].

7.1 The Protein Structure Calculation Code Dyana

Dyana computes three-dimensional protein and nucleic acid structures by energy minimization in a simulated annealing process and is highly valuable to the investigation of Prion related diseases like the Bovine Spongiform Encephalopathy (BSE), which is more commonly known as the “mad cow disease”. The planned use of Dyana fits the idea of good cause computing which makes it a good candidate to exploit free compute resources donated by Internet users. The application code Dyana was written at the Institute of Molecular Biology and Biophysics of ETH Zurich [52]. Dyana calculates the protein and nucleic acid structures from distance constraints and torsion angle constraints collected by nuclear magnetic resonance (NMR) experiments [51]. The computation is based on simulated annealing driven by the fast torsion angle dynamics (TAD) algorithm [61].

The simulated annealing is repeated several times, each time starting with different

initial random values of the degrees of freedom, e.g. the torsion angles. This independent generation of many conformers introduces a high degree of inherent parallelism into the structure calculation with Dyana.

Dyana is written for a master-slave setting. The master coordinates the parallel distribution and calculation. The slaves run the simulations and return the results (i.e. the resulting conformer) to the master that collects and analyzes them.

7.1.1 Task Parallelism in the SMP Version

The original version of Dyana runs on shared-memory multiprocessors (SMPs). The parallelism in the computation of the conformers has been reached by means of multiple independent processes spawned by a UNIX `fork()` system call. The master process creates the slave processes calling `fork()` and implicitly sends the data to them, according to the standard UNIX semantics for process creation. Since no information is exchanged during the computation of conformers, no additional shared memory data structures are necessary.

Each slave computes exactly one conformer. Once a slave has finished its computation, it sends the conformer back to the master and exits. The slaves return their results to the master through the file system.

The master is informed by the operating system upon the termination of a slave (child-termination signal). Although the master has to know when a slave exits, it does not need to know which particular slave has finished. A counter, which increases upon forking and decreases upon receiving a child-termination signal tells the master when a parallel part of the computation is finished. Once all conformer simulations have been completed, the master gathers the data about protein structures by reading the corresponding files and continues with the sequential execution of a subsequent evaluation part.

As long as there are more structures to compute, the master spawns slaves. At the same time, the master ensures that the number of slaves does not exceed the number of available processors.

7.1.2 Managing Computation Steps and Tasks with INCLAN

The Dyana software system provides a collection of functions implementing the algorithms for NMR structure calculation. The user arranges these functions to obtain a sequence of commands that computes a protein structure from its experimental NMR data. This high degree of flexibility has been achieved by integrating INCLAN (INteractive Command LANguage) [51] into Dyana. INCLAN is an interpreted command language, offering control logics like if-then-else commands, loop constructs, ordinary variable assignment and arithmetic operations to control the search for the molecule structure with minimal energy. Therefore all common strategies for the generation of conformers (i.e. work-units or tasks in computer science terms) can implemented in Dyana.

The flexibility of INCLAN allows the computation to incorporate several different tasking and scheduling models. For this effort in modeling and performance evaluation we use a fairly rigid tasking model. Each computation step starts with an initial conformer structure and generates a series of random variations that are considered for energy minimization at this step. Each variation of the conformer is handled by INCLAN as a central work queue in the master.

7.1.3 Characteristics of the SMP Version

The `fork()` system call for task creation simplifies the code implementation, since the entire address space of the master is automatically replicated to the slave. The process on the slave only lives until its computation completes and there is no need to reset some state or keep the state of the master and the slaves consistent.

On the other hand, forking a process can become an expensive task even on shared memory machines, unless copy on write is used. If we look at the `fork()` system call as an implicit send command that transmits, the whole address space of a process the potential cost becomes obvious. Furthermore, the distributed operating systems on cluster nodes do not provide the automatic replication of an entire address space across different machines and the `fork()` call cannot be used for task creation. Therefore the original SMP version of Dyana needs some re-engineering to run on clusters or widely distributed platforms.

7.1.4 Migration of Dyana from SMPs to Clusters and to Distributed Platforms

Code migrations from vectorizable codes to message passing programs require some well-known transformations and a certain amount of re-engineering. In most cases, a successful migration involves some significant re-engineering of the code. The control and data transfer primitives need to be changed from multi-threading with shared memory primitives to calls for message passing libraries or even back to calls handling TCP/IP streams for widely distributed computing on the Internet. Unfortunately, these transformations are often done without a cost-benefit model in mind and without a prior performance characterization of the code.

For a migration of Dyana from SMPs to strongly interconnected clusters of PCs and further on to frameworks for widely distributed computing, we separate the control mechanism (i.e. the task activation and task synchronization) from the data transfer mechanism (i.e. the exchange of the initial molecular state and the computed 3D structure at the end). Control and data transfers are adapted separately to the new platform. The options chosen for each platform in the migration path are shown in Figure 7.1.

In the migration from an SMP to a cluster, we change the control mechanisms from `fork()` calls implemented in the command language INCLAN to MPI primitives for

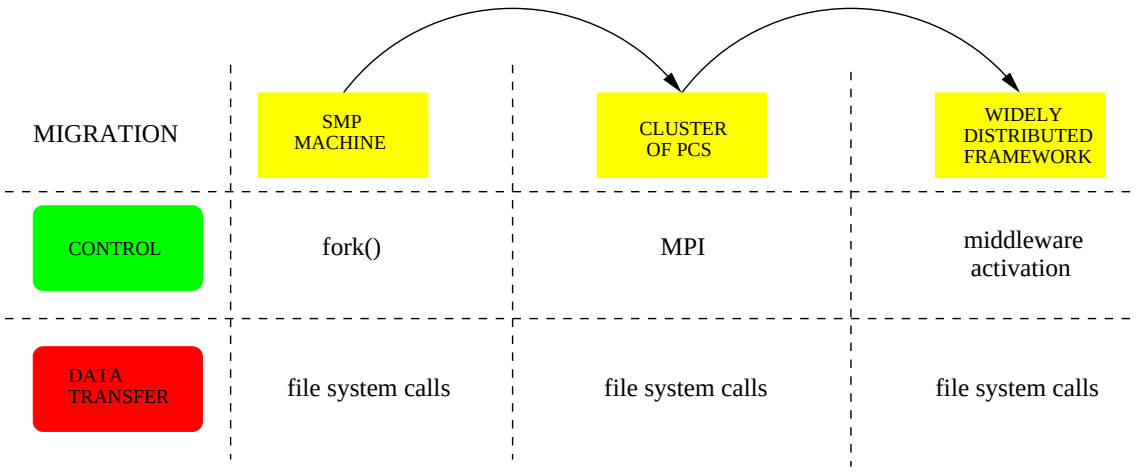

***Figure 7.1**: Control and data transfer mechanisms used in the migration path from SMP machine, to clusters of PCs and on to a framework for widely distributed computing. While in the migration we re-engineer the control mechanisms from `fork()` calls on SMPs, to MPI commands on clusters and middleware library calls on frameworks for widely distributed computing. We maintain the same data transfer mechanism (file system calls) for all architectures.*

task activation [94]. In frameworks for widely distributed computing, the task activation will be delegated to the appropriate middleware primitive using the library calls provided for this purpose. The additional synchronization primitives for the coordination of the initial and final data transfers are handled in a similar manner.

The inverted middleware framework shows irrelevant communication at the end of each computation of a conformer between master and slave. To manage the data transfers, we use the file system calls provided by the underlying memory mapped files on SMP machines, by the networked file system NFS for clusters of PCs or by the calls of the toolkit in widely distributed computing. Due to the small amount of data communicated, this communication can be done with a reasonable efficiency on all platforms considered. We acknowledge that there must be an appropriate security concept for each platform. Controlling the access to the slave computers and the integrity of data transfers is of a lesser interest to a performance study. Good solutions have been proposed in the past literature and we assume that the problem is solved in the middleware packages [4, 25].

For the migration to a widely distributed computing platform, the tasking model of a typical Dyana computation has to be reviewed critically. To provide the necessary parallelism the generation of new conformers must be done automatically and span more than one generation of alternatives. As with all optimization problems, the problem of getting trapped in local minima must be addressed carefully. The code needs to be adapted to a proper work-stealing paradigm as used in e.g. CILK [15].

The job queuing in the master and the work stealing scheduler can easily handle differ-

ent speeds of different machines and the model can also be extended to handle different classes of machines. For the total execution time only the total of compute power remains relevant, provided that the speed of a machine justifies the cost of networking it to a computational grid. Dyana computations are highly fault tolerant by their nature. Since variations of the conformers are generated by a random generator the infrequent loss of a task can be easily tolerated.

7.2 Performance Analysis of Dyana using MW^{-1}

In the performance analysis of Dyana using the inverted middleware, we no longer need a detailed knowledge of the middleware package as well as we do not act on the middleware by introducing any performance hooks like we did for the scientific computation code Opal in the previous chapter. The process of performance analysis of Dyana using the inverted middleware goes hand-in-hand with a systematic approach to the experimental design introduced in [62]. Two aspects have to be designed: the choice of the machine-resources which actively act on the application run-time and their proper representation by the response variables.

7.2.1 The Measuring Tools as Part of MW^{-1}

For our study of the performance predictions of Dyana on different platforms, we rely on the performance data a about the resource usage gathered by the inverted middleware during the successfully completed migration from SMPs to clusters of commodity PCs [90]. The performance data sampled at the operating system level by the inverted middleware framework is used to calibrate the detailed analytical model at the application-specific layer of our performance framework describing the precise performance characteristics of the code (CPU dependency, memory system usage, I/O requirements) for a variety of node architectures and networks, like uni-processor and dual-processor Pentium III nodes at 400, 800 and 1000 MHz. We integrate the model including several different memory systems (motherboards) and one completely different architecture using a DEC Alpha based cluster with a dedicated high speed interconnect [20]. Last but not least, we take into account different numbers of molecular structures: the Prion protein hPrP(R220K) and the ER2 protein.

7.2.2 The Analytical Performance Model as Part of MW^{-1}

As outlined in Section 7.1, the code Dyana simulates and evaluates the different alternatives of three-dimensional molecular structures, called conformers. For each of them, Dyana is responsible for:

- the creation of an initial structure with random values for the dihedral angles,

- the minimization of violations of the conformational constraints by simulating annealing using torsion angle dynamic,
- the evaluation of the resulting three dimensional structure by checking it against spectroscopic data.

For a run of n conformers on p processors, the total execution time, t_{dyana}, can be split into three terms:

$$t_{dyana} = t_{init} + t_{siml} + t_{comm} \tag{7.1}$$

where:

- t_{init} is the time spent to create and initialize the parameters needed during the whole simulation,
- t_{siml} is the time used to run the simulations of annealing using torsion angle dynamics,
- t_{comm} is the communication time during which the several slaves communicate with the master and vice versa.

Since t_{init} is constant for the whole simulation and irrelevant if compared with the other time components, we ignore its contribution.

The simulation is done in two phases. First, a random structure is created, then the simulation of annealing takes place. t_{siml} for n conformers on p processors is expressed as:

$$t_{siml} = \left\lceil \frac{n}{p} \right\rceil \left(t_{create\_structure} + t_{conf\_siml} \right) \tag{7.2}$$

where:

- $t_{create\_structure}$ is the time spent for creating the random structure,
- t_{conf_siml} is the time spent minimizing the violations of the conformational constraints by simulating annealing.

Since t_{conf_siml} is the most time consuming and $t_{create\_structure}$ is less than 5% of t_{siml}, we consider:

$$t_{siml} \approx t_{conf\_siml} \tag{7.3}$$

t_{conf_siml} is expressed as:

$$t_{conf\_siml} = \frac{FlOp_{conf}}{FlPt_{speed}} \tag{7.4}$$

where:

- $FlOp_{conf}$ is the total amount of floating point operations per conformer. $FlOp_{conf}$ is mostly determined by the protein structure considered, but it can also be a code factor and can depend on the compiler technology. Table 7.1 reports the amount of $FlOp_{conf}$, for both the molecules considered in our study, the Prion hPrP(R220K) and the ER2 proteins, on Pentium and Alpha architectures. The different number of floating point operations on the two platforms is related to the different compilers used and the different standard libraries called for transcendal functions.

Protein	Architecture	Compiler	$FlOp_{conf}$
hPrP	Pentium	PGI Compiler	6.5 GFlOp
hPrP	Alpha	DEC Compiler	5.3 GFlOp
ER2	Pentium	PGI Compiler	1.6 GFlOp
ER2	Alpha	DEC Compiler	1.4 GFlOp

***Table 7.1**: Total number of observed floating point operations ($FlOp_{conf}$) per conformer simulation for a Prion hPrP and a ER2 on Pentium and Alpha platforms. The different number of floating point operation on the two different platforms is due to the different compilers.*

- $FlPt_{speed}$ is the amount of floating point operations per second. It depends on architecture parameters like CPU rate and memory characteristics. The amount of floating point operations are sampled by means of our inverted middleware framework.

In our model, $FlPt_{speed}$ is approximated as:

$$FlPt_{speed} = \frac{CPU\_clock\_rate}{clocks\_per\_FlOp} \tag{7.5}$$

In Formula 7.5, the CPU_clock_rate is the frequency of the CPU. The clocks per floating point operation, $clocks\_per\_FlOp$, are further related to the CPU frequency and the memory characteristics:

$$\begin{aligned} clocks\_per\_FlOp &= f(CPU\_speed, \\ &\quad memory\_characteristics) \end{aligned} \tag{7.6}$$

The CPU is characterized simply by its clock speed. Despite all architectural differences like super-scalar execution, different handling of hazards and dependencies interfering with the pipelines, we observed that on most current microprocessors one floating point instruction can be successfully completed for every clock cycle. To characterize the memory system, we use the Extended Copy Transfer Characterization (ECT) [73, 72]. Our

model considers the average amount of cycles that are spent by one floating point operation when it has to fetch its data from memory. The total number of cycles used in an application run is estimated based on a detailed memory characterization (MBytes per second) taken from the ECT micro-benchmarks. We consider two different kind of memory accesses: contiguous and non-contiguous/strided access. We calculate the coefficients of the approximation by a linear regression model:

$$clocks\_per\_FlOp = a\alpha + b\beta + c\gamma \tag{7.7}$$

In Formula 7.7, α is the amount of cycles needed for a floating point operation without any memory access, β is the amount of cycles needed for a floating point operation which has to load a word from memory continuously, while γ is the amount of cycles needed for a floating point operation which loads a word from memory using strided access. Table 7.2 shows the values of α, β, γ estimated using the ECT model for the several platforms considered.

CPU Type	α	β		γ	
		CPUs		CPUs	
		1	2	1	2
Pentium III 933 MHz	1	13	18	33	75
Pentium III 800 MHz	1	17	21	26	50
Pentium II 400 MHz	1	11	14	29	40
DEC Alpha 667 MHz	1	7	7	34	34

***Table 7.2**: Values of α, β and γ estimated using the ECT memory system model for Pentium and Alpha platforms.*

The parameters *a*, *b* and *c* characterize Dyana and are chosen using the least-squares criterion [62] for all different machines, molecular structures and runs of Dyana. So on average, *a* represents the amount of operations which do not need any memory load, *b* is the number of floating point operations which access memory blocks continuously, while *c* is the amount which access the memory non-continuously.

At the end of each conformer simulation, the slaves send the computed three-dimensional molecular structure back to the master writing the data using file system calls and remain ready for a new simulation. In case of symmetrical networks, the time for the communication between master and slaves, t_{comm_conf}, depends on:

- the size of the data, $size_{data}$, that is exchanged,
- the number of processors, *p*,
- the bandwidth, $bandwidth_{master}$, that the master can sustain serving many slaves and

- the bandwidth, $bandwidth_{slave}$, that a single slave handle.

$$t_{comm\_conf} = max\left(\frac{size_{data}}{bandwidth_{slave}}, \frac{p \cdot size_{data}}{bandwidth_{master}}\right) \tag{7.8}$$

The time for the whole communication becomes:

$$\begin{aligned} t_{comm} &= \left\lceil \frac{n}{p} \right\rceil t_{comm\_conf} = \\ &\left\lceil \frac{n}{p} \right\rceil max\left(\frac{size_{data}}{bandwidth_{slave}}, \right. \\ &\left. \frac{psize_{data}}{bandwidth_{master}}\right) \end{aligned} \tag{7.9}$$

On strongly interconnected cluster platforms which normally consists of a small number of connected nodes and a large bandwidth (1000BaseT), the communication does not play any relevant role for the overall performance. On frameworks for widely distributed computing, it is either the available bandwidth of the slaves or the bandwidth of the master which limits the scalability of a parallel computation. The analytical model becomes:

$$\begin{aligned} t_{dyana} &= \left\lceil \frac{n}{p} \right\rceil \left(\frac{FlOp_{conf}}{CPU\_clock\_rate}(a\alpha + b\beta + c\gamma) + \right. \\ &\left. max\left(\frac{size_{data}}{bandwidth_{slave}}, \frac{psize_{data}}{bandwidth_{master}}\right)\right) \end{aligned} \tag{7.10}$$

The model can easily be adapted for asymmetric networks as found in home-computers connected to the Internet by asymmetric ADSL or CableTV links.

7.2.3 Model Validation

Since we have a running implementation of Dyana for SMPs and clusters, we can use application measurements as the most reliable way to validate the analytical model of the application-dependent layer in the inverted middleware. We compare the execution time measured with the estimated time allocated to the application run using the analytical model of the inverted middleware for a few selected scenarios and for different kind of molecular structures.

Figure 7.2(a) and Figure 7.2(b) compare respectively the execution time versus the estimated time allocated by the inverted middleware for the Prion protein on a Pentium II cluster (400 MHz) with one active processor per node (single processor) and two active processors per node (dual processors). Figure 7.3(a) and Figure 7.3(b) display the same comparisons with the ER2 protein. The graphs point out a good level of accuracy of the model. Figure 7.4 compares the execution time versus the time component allocated by the inverted middleware for the Prion on a cluster of 16 Alpha (667 MHz) dual processors.

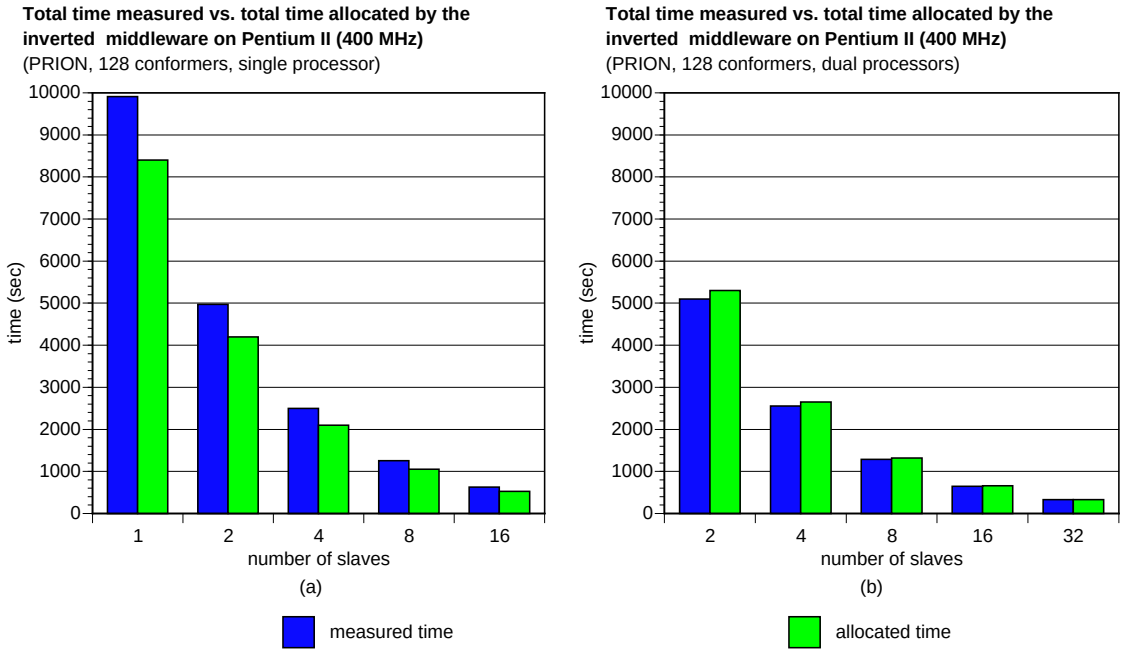

***Figure 7.2**: Comparison of the total time measured and the total time allocated by the inverted middleware to Dyana runs for a Prion protein on a cluster of dual Pentium II (400 MHz). A different number of slaves with one active processor per node (a) and with two active processors per node (b) is considered.*

7.3 Performance Prediction of Dyana using MW^{-1}

In the past sections, we established and calibrated an analytical performance model for Dyana based on the migration from SMP platforms to clusters of PCs. In this section we will use our model to extrapolate the performance of Dyana to different compute platforms at both ends of the performance spectrum:

- a framework for widely distributed computing (a cluster of 400 MHz Pentium Celeron Processors) interconnected by a highly heterogeneous network connecting home users with standard modems to a master with a high bandwidth connection,

- a strongly interconnected high-end cluster whose nodes, all 1 GHz Pentium III processors, are interconnected by high bandwidth networks (1000BaseT).

In Dyana, the master performs only short calculations when it receives a request by a slave. The amount of the data exchanged between master and slave depends on the molecule but in general it is not large. For a typical Prion protein, the communication size, including all the synchronization signals, is less than 20 KBytes.

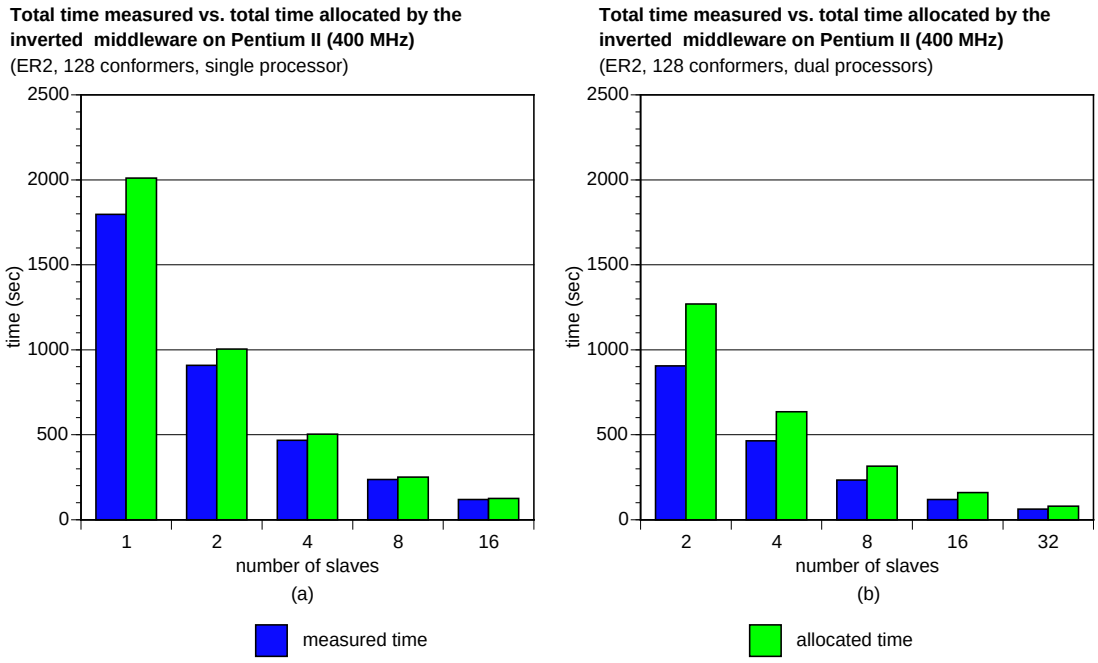

***Figure 7.3**: Comparison of the total time measured and the total time allocated by the inverted middleware to Dyana runs for a ER2 protein on a cluster of dual Pentium II (400 MHz). A different number of slaves with one active processor per node (a) and with two active processors per node (b) is considered.*

7.3.1 Performance Prediction of Dyana in Widely Distributed Computing

Although Dyana does not require much computation of the master when it receives a service request, this task may become a bottleneck as soon as the master cannot cope with an exceptionally large amount of requests. This is not an unrealistic scenario when thousands of computers on the Internet become involved. With our simple model and the profiling data of the application on clusters, we can try to estimate this cross-over point, when a master can no longer serve the amount of requests received without causing a limitation in performance.

On frameworks for widely distributed computing, the master normally resides on a strongly interconnected platform on the back-bone using fully switched 100BaseT or 1000BaseT Ethernet, while the slaves are usually connected to this framework by analog modems (max 7 KByte/s), ISDN (max 16 KByte/s) or ADSL (max 256 KByte/s) connection. We assume that most of the slaves are connected to the Internet by analog modems. This gives us a worst case speed of 7 KBytes/sec. With these numbers and our model, we can estimate the communication time:

$$t_{comm} = \left\lceil \frac{n}{p} \right\rceil max\left(\frac{size_{data}}{bandwidth_{slave}}, \frac{p \cdot size_{data}}{bandwidth_{master}}\right) \tag{7.11}$$

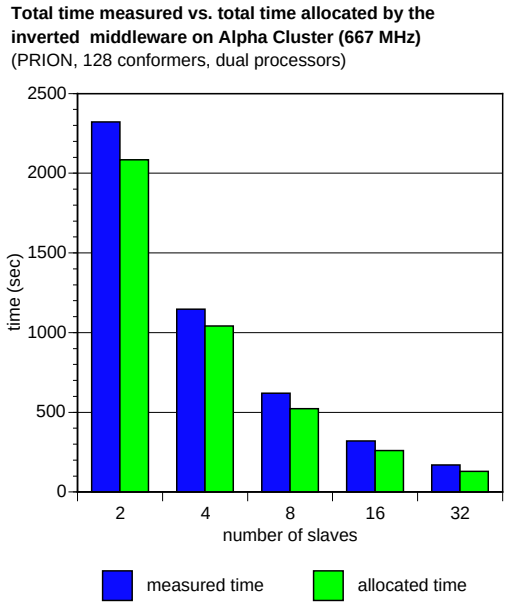

***Figure 7.4**: Comparison of the total time measured and the total time allocated by the inverted middleware to Dyana runs for the Prion protein on a cluster of DEC Alpha (667 MHz) for different number of slaves with two active processors per node.*

With communication time and duty cycle of the master (master computation time over slave computation time), we can easily derive the maximal number of slaves a single master can handle.

$$p > p_{max} = \frac{bandwidth_{master}}{bandwidth_{slave}} \approx 1500 \tag{7.12}$$

In the scenario in which the bandwidth of the slaves limits the communication, a request needs approximately 3 seconds to be served:

$$t_{comm\_conf} = \frac{size_{data}}{bandwidth_{slave}} \approx 3sec \tag{7.13}$$

According to our prediction for the case of a slave with a low-cost node (Pentium II 400 MHz), the computation of a conformer requires an average 85 seconds. During this time, the master can serve other requests and the duty cycle of Dyana becomes:

$$duty\_cycle = \frac{t_{conf\_siml}}{t_{comm\_conf}} \approx \frac{85}{3} \approx 28 \tag{7.14}$$

The total number of slaves per master that can be served on this platform is related to the maximum number of slaves which can be served at the same time, p_{max}, and the duty cycle of the application, $duty\_cycle$. In a situation of perfect load balancing, the maximal number of slaves becomes approximately:

$$p_{max} \cdot duty\_cycle = 1500 \cdot 28 \approx 42000 \tag{7.15}$$

For a good scalability beyond this point, the master must be replicated and a second level of parallelization must be added. Multiple masters can be used to do the communication phases and prevent related bottlenecks on computation involving more computers.

The problem is overcome by replication of the master in a multilevel hierarchical setup rather than running into peer-to-peer paradigms. Applications like Dyana are ill-suited for a peer-to-peer setting since they need a coordinator which gathers intermediate results and based on those results dynamically re-schedules the simulation tasks.

The parameter values observed in Dyana and similar codes indicate that the typical network latency of a few hundreds milliseconds in a computational grid is far less than the computation time of a task and even an order of magnitude less than the total communication time required for the execution of the task. The computations are therefore *communication bandwidth* and not *communication latency* limited.

Figure 7.5(a) shows the total time for the computation of 256 conformers. Figure 7.5(b) reports in logarithmic scale the prediction of the scalability in terms of conformer number per hour for the Prion protein on a framework of for widely distributed computing (commodity PCs - Pentium II 400 MHz Celeron) connected to the master from home by means of analog modems (max 7KByte/s). The chart shows the good scalability predicted for the Prion simulation using Dyana on such a widely distributed system.

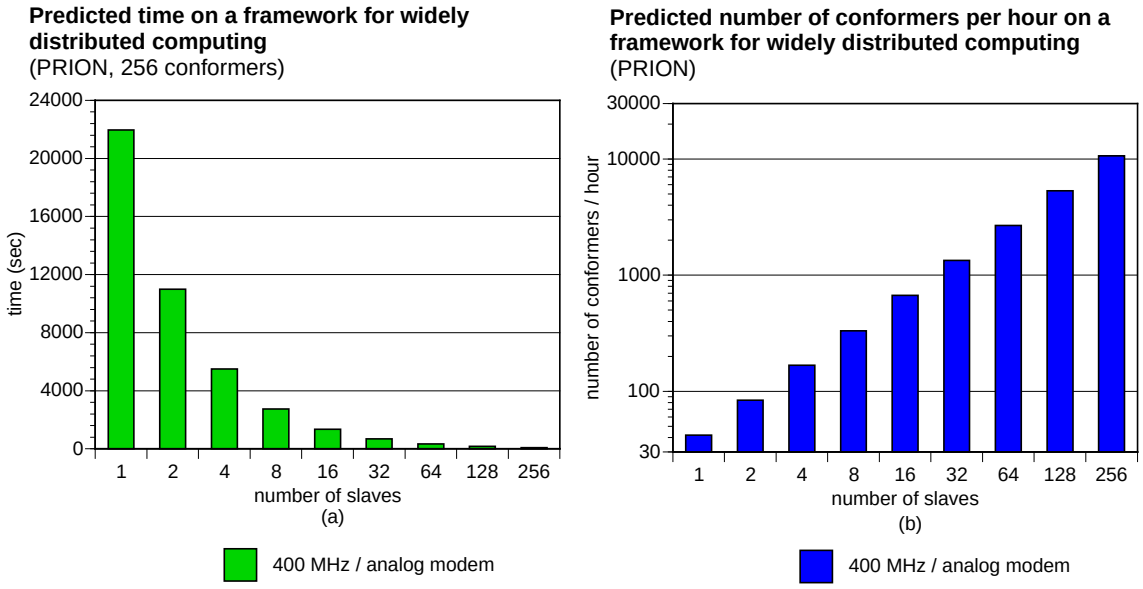

***Figure 7.5**: Predicted execution time of Dyana for the computation of 256 conformers (a) and predicted compute rate in conformers per hour (b) of Dyana for a Prion protein on a framework for widely distributed computing. The predictions are computed by our analytical model from basic performance characteristics of the new target platform.*

7.3.2 Performance Prediction of Dyana on Strongly Interconnected High-End Clusters

For the verification of the model we also consider new platforms with higher performance than the current Beowulf clusters. On a strongly interconnected high-end cluster, the communication rate requirements of Dyana are easily met. Therefore the communication time can be assumed zero. Figure 7.6(a) shows the total time for the computation of 256 conformers for a strongly interconnected high-end cluster whose nodes, Pentium III 1 GHz, are interconnected by high bandwidth networks, while Figure 7.6(b) show the prediction of the number of conformers per hour. As shown in Figure 7.6(b), also for

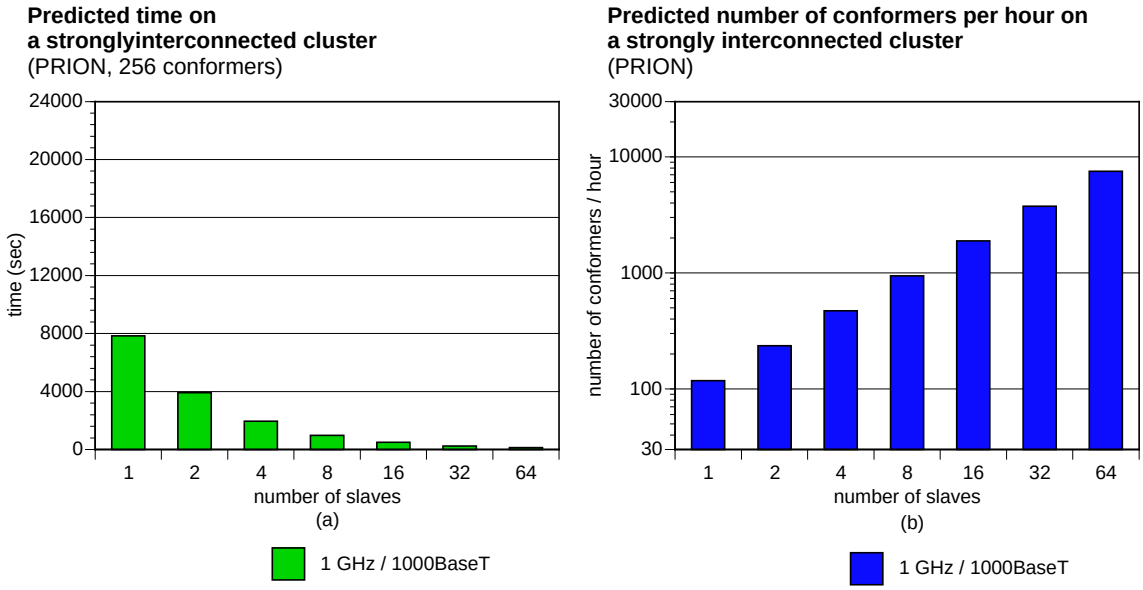

Figure 7.6*: Predicted execution time of Dyana for the computation of 256 conformers (a) and predicted compute rate in conformers per hour (b) of Dyana for a Prion protein on a strongly interconnected high-end cluster. The predictions are computed by our analytical model from basic performance characteristics of the new target platform.*

a strongly interconnected high-end cluster, we can predict good scalability for the Prion simulation. In contrast to Figure 7.5(b), Figure 7.6(b) shows a higher gain however on frameworks like the one considered in Section 7.3.1 we use commodity technology and the computation cost is therefore less while the good scalability remains.

7.4 Extending our Approach to other Codes and Future Work

An entire class of scientific codes (Opal, CHARMM, GAMESS, AMBER) can distribute their work-units among different processors in a cluster of PCs or on a computational

grid. Due to a highly similar master-slave setting, some computation intensive phases are typically followed by a communication phase.

In addition to Dyana, we used our inverted middleware framework to look at the scientific computation code CHARMM (Chemistry at HARvard Macromolecular Mechanics). The resource usage of CHARMM at the architectural and the operating system level are highly similar to Dyana. Therefore the modeling of CHARMM can be achieved by a similar set of formulas given in this chapter for modeling the performance of Dyana. The application parameters that determine the clock-per-floating-point-operation (i.e. a specialization better known as CPI) are also very similar. So the usage of the floating point units and of the memory system in CHARMM and Dyana are very alike. The total number of floating point operations per work-unit is different, but can easily be determined by benchmarking the sequential version of the code with our inverted middleware. The amount of communication to start and complete a work-unit can be characterize in the parameters used in our model. Therefore the same model can be extend and used to characterize the code and predict the performance of CHARMM for grid environments.

7.5 Conclusion about the Performance Prediction of Dyana using MW^{-1}

A successful migration of the molecular dynamics code Dyana from SMP workstations to a more cost-effective cluster of commodity PCs drastically increases the compute resources available to biologists for this kind of calculations. Because of a clean separation of control and data transfer, the amount of code re-engineering needed for the migration is kept at a minimum. While we readily adapt the control transfer mechanism to the best tasking paradigm for a particular platform, we leave all bulk data transfers with file system calls, using the appropriate underlying mechanism of shared memory in SMPs, NFS in clusters or the corresponding toolkit functions in grid computing environments. This programming shortcut is properly justified by the small impact of the data transfers on the overall performance of Dyana.

To further increase the number of computers available to Dyana, we study the viability of a next migration step from clusters to some newly developed frameworks for widely distributed grid computing on the Internet. The effectiveness of such a migration is studied with an analytic performance model of Dyana integrated into the inverted middleware framework. The model predicts performance on a broad range of different computing platforms including SMP servers, clusters of PCs and novel infrastructures for widely distributed computing on the grid. The model incorporates application parameters like protein size and platform parameters like CPU clock frequency, memory system type or network speed and the degree of parallelism used. The parameters are properly fitted to different experiments involving 1 to 32 processors as 32bit Intel Pentium IIIs and 64bit Compaq Alphas with clock rates between 400 MHz to 1 GHz.

The model in the inverted middleware is calibrated and validated during a previous migration from SMPs to clusters of PCs. The model reaches an excellent fit with a high accuracy in its predictions on average and a worst case of just of 19% deviation from experimental data. The per processor performance range considered is 82 MFlop/s to 180 MFlop/s per processor. The aggregate performance in our most parallel system reaches up to 4.9 GFlop/s.

After calibration, we use our analytical model to extrapolate the application runs to a widely distributed computing infrastructure incorporating thousands of processors on the Internet and are predicting excellent scalability up to approximately 42000 processors. At this number of slaves the communication to the master becomes the bottleneck. Replication of the master in a multilevel hierarchical setup can resolve this.

The performance methodology followed this study indicates that Dyana can be successfully migrated to frameworks for widely distributed systems on the Internet and is, therefore well suited for grid computing. Since Dyana is crucial to the ongoing research of Prion infections like BSE, the code would be an ideal candidate for a “good cause” computing campaign.

8

Workload Characterization with MW^{-1} of Database Applications

In applications that use multiple processors and distribute a large workload for the sake of higher execution speed, precise understanding of resource utilization becomes a crucial concern for finding the causes of performance and scalability problems. Such an investigation requires a proper engineering of the complex system. A precise understanding of resource utilization is not just required for fine tuning a running system, but also for the prediction of scalability or the study of the viability of new, alternative platforms such as clusters of low cost commodity PCs. Using the inverted middleware, we are able to look at the in-depth performance analysis of the distributed TPC-D benchmark (a standard OLAP application). The inverted middleware delivers many interesting insights about the most critical resources in the different queries otherwise not available just using the elaborate instrumentation for performance monitoring enclosed in standard DBMSs. In this chapter, we show that the data provided by our inverted middleware are a solid basis for architectural decisions and system optimization.

8.1 Distributing OLAP Workloads on Clusters of PCs

Our performance analysis method is strongly motivated by large OLAP (On-Line Analytic Processing) applications running in parallel on a cluster of PCs. Although the PowerDB project at ETH Zurich deals mainly with parallel OLTP (On-Line Transaction Processing, e.g. TPC-C benchmark [106]), we are more interested in OLAP workloads (e.g. TPC-D benchmark [107]). OLAP applications deal with large quantities of data in a single query that could potentially benefit from high speed interconnects in advanced PC clusters. On the other hand, high performance workloads in OLTP require an extremely large number of simultaneous queries to scale, and - in general - do not result in large data transfers. Therefore a large scale OLTP setup is much harder to generate than a large scale OLAP job, and therefore less interesting for computer architects. The accurate behavior

of database workloads for OLTP jobs is addressed for shared-memory multiprocessors in [84].

In this chapter, we show how the inverted middleware can be used to explain the precise resource usage and the scalability of the TPC-D benchmark running on clusters of PCs including disks and networks. We demonstrate that there is indeed a systematic way to filter the raw performance data provided by the operating system and turn the result into a more abstract performance picture that reflects the resource usage of the distributed application code. As cluster architects we can improve the configuration of future PC clusters based on this data about resource usage. Depending on the most performance critical resources we can improve the cost/performance ratio by using cheaper or more expensive CPUs, disks, memory systems (motherboard) or interconnects between the nodes. If we can use cheaper components without deterioration in performance, we can afford a larger number of nodes in a cluster.

Other studies address the problem of understanding which hardware component leads to which part of the total execution time. In [2] simple queries have been studied for a memory resident database and different kind of commercial DBMSs running on an Intel Xeon and Windows NT 4.0. The study focuses on processor and memory interactions excluding effects of I/O subsystems and data partitioning.

Since cluster computing is always about the best use of commodity components, we look at parallel- and distributed systems built entirely with commodity hardware and commercial software components. In particular, we combine the open source operating system LINUX with the proprietary single node DBMS of ORACLE and a lean experimental software layer for the process of distribution of the queries. The resulting high performance database system for decision support workloads can be classified as a shared-nothing architecture. Since we are interested in using clusters for very large data sets exceeding the disk capacities of a single node, our data distribution scheme relies on disjoin partitioning rather than on full replication.

8.1.1 The TPC-D Benchmark

The TPC-D benchmark [107] consists of a broad range of decision support applications that require complex, long running queries against large data structures. Although this benchmark became obsolete in 1999, we still use it as a representative of OLAP applications for historical reasons but we could easily migrate our approach to TPC-H or TPC-R and the qualitative aspects of our study would remain the same if these more recent benchmarks were used. Furthermore, we are not interested in publishing new results for this benchmark, but our aim is rather to demonstrate how our particular performance analysis framework may help in detecting bottlenecks and architectural problems. Some of the more recent work in the PowerDB project also deals with TPC-H and TPC-R including concurrent updates, but this is not relevant to the statements in this thesis.

The TPC-D benchmark contains 6 dimension tables (Customer, Nation, Region, Supplier, Part, PartSupp) and 2 fact tables (Order, LineItem). It consists of 17 queries with different characteristics.

The empirical scalability of the different TPC-D queries, chosen as a typical example of OLAP applications on PC clusters presents a picture of unpredictable performance and speed-up numbers that seem to be highly sensitive to the nature of the workload. Figure 8.1 shows a typical study case for the TPC-D benchmark distributed with TP-Lite (a software tool for distributing queries) on three nodes of a cluster of PCs. We expect a speed-up of almost three (shown with the upper line in the picture) however, not all the queries achieve speed-up, some of them get a slow-down (i.e query 2, query 9 and query 10) and some of them (i.e. query 5 and query 7) do not even end in a reasonable time due to some inefficiency in parallel processing.

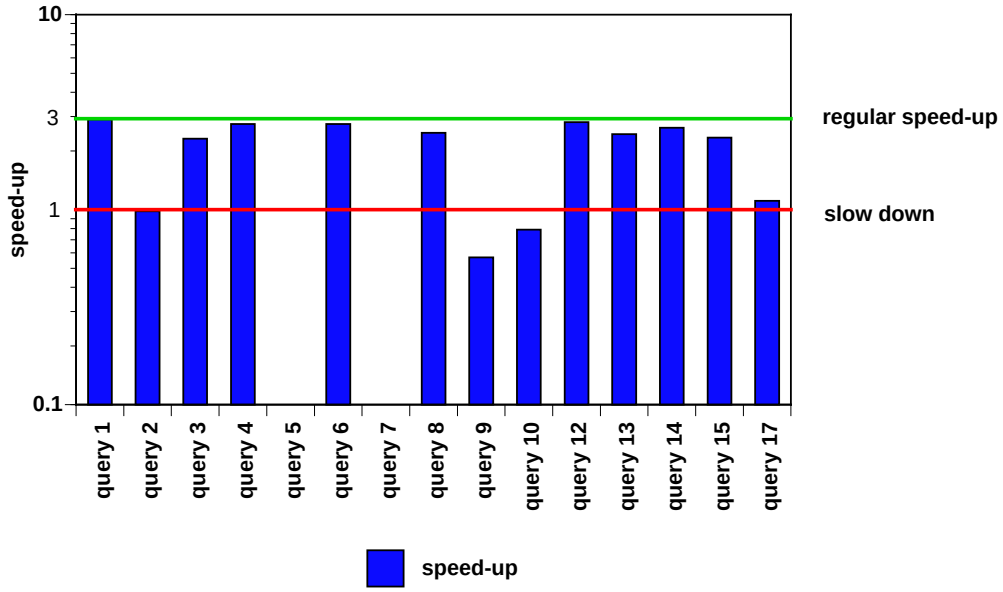

***Figure 8.1**: Scalability for the TPC-D benchmark distributed across three nodes of a cluster of commodity PCs with TP-Lite*

To study the reason for suboptimal performance as seen in Figure 8.1, most DBMS like ORACLE incorporate elaborate instrumentation for performance monitoring and tuning [53, 40]. However, such instrumentation normally works on the basis of data collection inside the DBMS and not by mapping the operating system state back onto an abstraction level that is more appropriate for a user. In particular, the performance monitoring instrumentation of ORACLE only accounts for the total count of operations and does not allow for efficient sampling of performance information at certain intervals nor for this information to be efficiently collected from a large number of processing nodes in a cluster. While some of the logical or table access counts would certainly be interest-

ing, most performance information of the database management system does not directly relate to the usage of physical resources in a distributed system and is therefore hard to use in a framework that aims at predicting the execution time based on application and platform parameters.

8.1.2 TP-Lite Approach

In principle, a parallel implementation of a database on a cluster of PCs works as follows: clients send their requests (i.e. SQL-transactions) to a so-called coordinator. The coordinator analyzes these requests, partitions them into several independent sub-transactions, and routes the sub-transactions to the nodes within the cluster. The goal of the partitioning and the routing is to minimize response times of queries and to maximize the throughput of SQL transactions. While the general approach is rather complex due to update operations and concurrency, we focus our attention on a query-only environment as this is sufficient for most work in OLAP. In this case, we do not have to deal with replication, concurrency, dynamic partitioning of data, and crash recovery (see [88, 89, 49] for more details). In our simplified approach based on a *shared-nothing architecture*, we use a static and disjoint partitioning of the data in the cluster. Queries are sent to all nodes and the union of the resulting tuples is composed into the overall answer of the query. The partitioning is such that each node has to touch about the same amount of data for query evaluation.

With TP-Lite, a query is executed in a master-slave setting using an instance of ORACLE8 on each node, its proprietary database links, and PL/SQL [29]. The coordinator runs both the master and slaves in independent transactions and we use ORACLE pipes as their communication primitive. A SQL-query is executed as follows: the master sends a message to all slaves over ORACLE pipes to initiate query execution. Each slave executes the SQL-query against the database of a dedicated node in the cluster using a database link. The result tuples are sent back to the master as messages over a shared ORACLE pipe. TP-Lite can be seen as a poor man's implementation of a parallel database.

Although Böhm et al. [17] reported that TP-Lite is easily outperformed by using a TP-monitor or a proprietary coordination layer, we have used the TP-Lite implementation in conjunction with OLAP applications on clusters of commodity PCs to demonstrate the ability of our performance monitoring framework to detect and identify performance bottlenecks. Our performance method is able to document and explain why the TP-Lite approach does not always scale well when the number of nodes is increasing to a larger amount.

8.2 Performance Analysis of OLAP Application with MW^{-1}

ORACLE provides many means to estimate and measure the costs for executing SQL operations. For instance, a number of internal system performance tables record how many resources have been consumed by the current session, e.g. the number of disk accesses, CPU consumption, or the number of requested locks. Our framework complements this information with aggregated performance measures of the nodes in the cluster. In contrast to the performance tables in ORACLE, we not only know who consumed how many resources during the execution of a query, but also when and where these resource consumptions occurred. An interesting question at this point is: how can we exploit this knowledge in order to optimize the application.

In the optimal case, the inverted middleware would be able to tune the application automatically so that changes of hardware or query characteristics would immediately lead to a better (hopefully optimal) configuration of the application. However, having a database as the middleware, automatic tuning of the application is not entirely feasible due to the complexity of the SQL interface. Commodity databases like ORACLE can only adjust some specific performance settings: e.g. ORACLE uses a cost-based query planner to find an optimal execution strategy taking statistical information about the data distribution and performance characteristics of the operating system into account. However, important aspects like physical design (e.g. what indexes, how much normalization, how much replication, how to cluster tables) or partitioning of data are far beyond what commodity database systems are able to optimize. Therefore we assume that the application-specific layer further incorporates some experts who are able to tune the application based on the footprints generated by the monitoring tool.

The query execution plan describes the algorithm that the database uses in order to evaluate the given SQL query. In most cases, one could apply a number of different plans to a single query. Choosing the optimal one, however, is not an easy task. Given the performance data of the lower levels of the inverted middleware, an expert could identify bottlenecks in resource allocation and could relate them to the operations performed by the database according to the execution plan. In order to remove these bottlenecks, the expert might suggest creating additional indexes or partitioning data in a different way (among other possibilities).

8.2.1 Experimental Setup

Factors and their levels

We look at the execution of a parallel query on top of multiple instances of ORACLE running on nodes of a cluster of PCs under the LINUX operating system.

The total size of the tables in the TPC-D database benchmark is 10GB. Out of the 17 queries of the TPC-D benchmark, we mainly used query 1, 3, 4, 8 and 12 for the

experiments since they read large parts of the fact tables. As mentioned above, our implementation of a parallel database uses static partitioning of data. While the dimension tables are fully replicated on all nodes, the data of the fact tables is disjointly distributed over the nodes of the cluster. Since the queries under consideration run against large parts of the fact tables, we achieve a speed up with the cluster simply due to the reduced volume of data touched by each node.

An application-run in our experiment depends on several parameters controlling the workload and the execution environment. The parameters under investigation are called factors. Most factors are external and under the control of the application writer (*application factors*). For the experiments, we used two different partitioning schemes. The first scheme ($1-part$) partitions only the LineItem fact table, the second one ($2-part$) partitions both fact tables. Further, note that our partitioning scheme and the selection of the queries guarantee that the union of the result tuples generated by the nodes in the cluster is equal to the result set of the query against the entire database. However, it is beyond the scope of this study to discuss this issue and related aspects in more detail.

Our approach also deals with parameters that affect performance but are not directly visible to the application writer such as the set of factors related to the characteristics of the platform (i.e. the architectural parameters of a PC cluster), which we define as the *platform factors*. Among them we consider the factors at the following levels:

- the clock rate of the CPU (t_{clock_cycle}): 400 MHz Pentium II (Deschutes) or 1000 MHz Pentium III (Coppermine),
- the average disk read performance[1] ($disk\_rate$): 22.0 MB/s (slow disk) or 30.5 MB/s (fast disk)
- the average disk access time (t_{rand_access}): 7.3 ms (slow disk) or 6.8 ms (fast disk),
- the speed of the network interconnect ($network\_speed$): 100 Mbit/s (Fast Ethernet) or 1000 Mbit/s (Gigabit Ethernet),
- the number of slaves in the cluster (ns): 1, 3 or 6 processing nodes[2].

Besides the platform factors (parameters under investigation), some parameters are held at constant level for this investigation. For the sake of clarity, they are written as

[1]The disk characteristics in detail are: slow disks, Seagate SCSI ST318203LW: with a min/avg/max throughput of 14.5/22.0/26.9 MB/s and access time 7.3 msec, the fast disks are Seagate SCSI disks ST318404LW with a throughput of 22.8/30.5/36.3 MB/s and 6.8 msec access time.

[2]The power database project aims at the scalability to a larger number of nodes (16, 32, or 64 processors), however, measuring such task requires very large data sets. With our current experiments, we use 10 GB of data which is adequate for 1, 3, 6 processors. We plan to extend our work to the new 128 node "Xibalba" database cluster as soon as we manage the engineering challenge of generating and distributing data sets of 100 GB to 10 TB size with LINUX.

variable in the analytical performance model of the performance monitoring framework. Among them are the clock cycle per instruction (CPI = 1) linking CPU clock-frequencies to integer performance, the size of a memory block (blk = 512), which is an important constant in the I/O buffering system of LINUX, the number of disks per node (3 disks: one for the system information, one for the indexes and a third for the data), the physical capacity of each disk ($disk\_size$ = 18 GB), and the memory size (512MB).

Response variables

The performance data (response variables) we collect from the nodes comprises:

- the number of instructions (i.e. user instructions $IC_{user}(j)$ and system instructions $IC_{sys}(j)$) on the coordinator and nodes CPUs. j ranges from 1 to $ns+1$ where ns is the factor expressing the number of nodes.

- the number of sequential disk accesses ($seq(j)$) and non sequential disk accesses ($no\_seq(j)$) to the disk of the coordinator and each node (j ranges from 1 to $ns+1$).

- the average stride $avg\_stride(j)$ for the disk access in each node and the coordinator. By average stride, we mean the average movement of the disk head between two random accesses to a disk. Sequential disk accesses show up as reads with a stride of one while non sequential accesses show up as larger strides [1].

- the amount of traffic transferred over the network interconnect (i.e. Fast Ethernet, Gigabit Ethernet). We distinguish between the amount of *bytes received* ($size\_rec(j)$) by each node and the coordinator on the network device, and the amount of *bytes sent* ($size\_trans(j)$) from each node as well as from the coordinator on the network device. Again, j ranges from 1 to $ns+1$.

This information is sampled as a user-defined rate over the time of an execution. In our case, because of the length of the query taken into account, the rate is of one ample each second.

8.2.2 Modeling Data Parallelism

The application-specific layer of the inverted middleware should suggest optimizations or provide feedback for the application based on the performance data gathered by the distribution-specific layer and the system-specific layer. We develop our analytical model starting from the constituent components which reflect application performance activities.

[1] In our model, the block index actually increments by two since LINUX always reads two blocks (blk) at a time.

The run-time of a query t_{run_tot} is expressed as:

$$t_{run\_tot} = \max_{j=1..ns} t_{slave}(j) + k \tag{8.1}$$

where *ns* is the number of slaves, $t_{slave}(j)$ is the time spent for running the query on the slave j (j ranges from 1 to *ns*), k is the time spent by the master for any reorganization of the data required in the query (ie. sorting).

The time for running a query on a single slave is the sum of several time contributions:

$$t_{slave}(j) = t_{CPU}(j) + t_{DISK}(j) + t_{NET}(j) + t_{MW}(j) \tag{8.2}$$

$t_{CPU}(j)$ is the time contribution due to the CPU computation on the slave j, $t_{DISK}(j)$ is the time related to the disk accesses (sequential and non sequential access), $t_{NET}(j)$ is the time spent for the communication between the slave j and the master. Last but not least, $t_{MW}(j)$ is the time contribution related to the middleware.

Estimation of the CPU usage: t_{CPU}

By monitoring resources at the operating system layer, we are able to catch and model the time components related to the distributed system. The CPU time $t_{CPU}(j)$ on the slave j can be expressed as:

$$t_{CPU}(j) = IC(j) * CPI * t_{clock\_cycle}(j) \tag{8.3}$$

The instruction counter or *IC*(j) is the counter of the instructions executed on the slave j. It is the sum of the user instructions $IC_{user}(j)$ and the system instructions $IC_{system}(j)$:

$$IC(j) = IC_{user}(j) + IC_{system}(j) \tag{8.4}$$

CPI is the average number of clock cycles per instruction. *CPI* depends on the organization and instruction set architecture. On our framework, we assume $CPI = 1$. $t_{clock\_cycle}(j)$ is the clock cycle time related to the hardware technology.

Estimation of the disks usage: t_{DISK}

The time spent reaching the data on the disks ($t_{DISK}(j)$) is the sum of the time spent for sequential access to disks ($t_{sequential}(j)$) and the time spent for non sequential access to disks ($t_{no\_sequential}(j)$).

$$t_{DISK}(j) = t_{sequential}(j) + t_{no\_sequential}(j) \tag{8.5}$$

The time spent for sequential accesses to disk ($t_{sequential}(j)$) is expressed as:

$$t_{sequential}(j) = \sum_{i=1..3} \frac{seq(j,i) * (2 * blk)}{disk\_rate(j,i)} \tag{8.6}$$

The factor i ranges from 1 to 3, or more in detail from the system data disk, to the index disk and to data disk, $seq(j,i)$ is the amount of sequential accesses to the disk i, blk is the size of a block (512 Bytes) and $disk\_rate(j,i)$ is the rate of the disk i on the slave j.

The total time for non sequential access to the disks on the slave j ($t_{no\_sequential}(j)$) is expressed as:

$$t_{no\_sequential}(j) = \sum_{i=1..3} (no\_seq(j,i) * t_{avg\_stride}(j,i)) \tag{8.7}$$

$no\_seq(j,i)$ is the amount of sequential accesses on the disk i (again we consider three different disks for the system information, the indexes and the data). The average stride for a non sequential access $t_{avg\_stride}(j,i)$ on the disk i is expressed as:

$$t_{avg\_stride}(j,i) = \frac{(blk * avg\_stride(j,i))}{disk\_size(j,i)} * t_{random\_access}(j,i) \tag{8.8}$$

The disk size $disk\_size(j,i)$ and the average time for a random access $t_{random\_access}(j,i)$ are given as well as the block size blk (512 Bytes) is known. The average stride $avg\_stride(j,i)$ can be computed as the difference between the total stride size ($total\_stride(j,i)$) and the total sequential stride size ($2 * seq(j,i)$) over the total amount of non sequential accesses measured $no\_seq(j,i)$:

$$avg\_stride(j,i) = \frac{total\_stride(j,i) - 2 * seq(j,i)}{no\_seq(j,i)} \tag{8.9}$$

Estimation of the network usage: t_{NET}

For the several queries considered, the communication does not depend on the speed of the network but on the size of the data received and transmitted and the communication rate of the master:

$$t_{NET}(j) = (size\_rec(j) + size\_trans(j)) * \frac{(rate\_rec(master) + rate\_trans(master))}{ns} \tag{8.10}$$

where both the amount of data (in bytes) transmitted ($size\_trans(j)$) and received ($size\_rec(j)$) from and to the slave j and the communication speed of the master ($rate\_rec(master)$ and $rate\_trans(master)$) has been measured by means of the monitoring tool. The maximal communication speed of the master is 1500 KByte/s.

$$rate\_rec(master) + rate\_trans(master) = 1500 KByte/s \tag{8.11}$$

8.2.3 Model Simplifications

There are two major simplifications in the model. First, although we make use of three disks in parallel for each node (one disk for the system information, one for the indexes and one for the data), in our model we summate the time allocated to the accesses from/to

the disks as if the accesses were sequential. The presence of three three disks per node should assure parallel access to the information and reduce the access time. However, we have not noticed any relevant reduction of the time for the queries considered due to the parallelism introduced by the disks.

In our analytic model, we have not included the memory effect. We do not include the memory effect because we assume that the database size is so large that its data cannot be completely stored in the memory during a query-run. The disk access time becomes larger than the memory access time thus overwhelming the memory effect. Our assumption is supported by the major motivation behind the migration of OLAP database work to clusters of PCs which is to run huge databases of terabyte-scale on PC clusters that are equipped with a large number of disks. This aspect also explains why in our study we look at a reduced number of TPC-D queries. The method is extendible to the whole set of queries of the TPC-D benchmark as long as the data of the queries does not completely fit in the memory of the nodes. For faster experimentation with different cluster configurations we have scaled the problem size down to 10GB and have artificially limited memory size on each node to keep the original balance in the storage hierarchy. However, some queries like query 2 and query 17 are not considered in our study because they have such a reduced number of accesses to the disk and all of their data can easily reside in the memory.

8.2.4 Modeling Validation

We can use application measurements as the most reliable way to validate the analytical model in the inverted middleware. We compare the execution time measured with the estimated total time allocated to the resource usage using the analytical model of the inverted middleware for a few selected queries (i.e. query 1, 3, 4, 12) and for different levels of factors. The maximal error measured is up to 20%. In the following, we look at some of the validated cases.

Figure 8.2 and Figure 8.3 show the comparison of the total time measured and the total time allocated to the machine resources by the inverted middleware for query 1. For the validation, we consider a 3D space of factors (see Figure 8.2) whose components are: the CPU clock rate, the number of nodes among which the data of the tables is disjointly distributed and the speed for the disk accesses. The levels for the CPU clock rate are two: 1GHz and 400 MHz. Three different levels for the number of nodes have been chosen: one, three, six. Last but not least, there are two different access speeds to the disks: fast and slow access to the disks.

Figure 8.4 and Figure 8.5 display the comparison of the total time measured and the total time allocated to the machine resources by the inverted middleware for query 4. For the validation of query 4, we consider a 3D space of factors whose components are both application and platform factors (see Figure 8.4). We take into account: the speed to

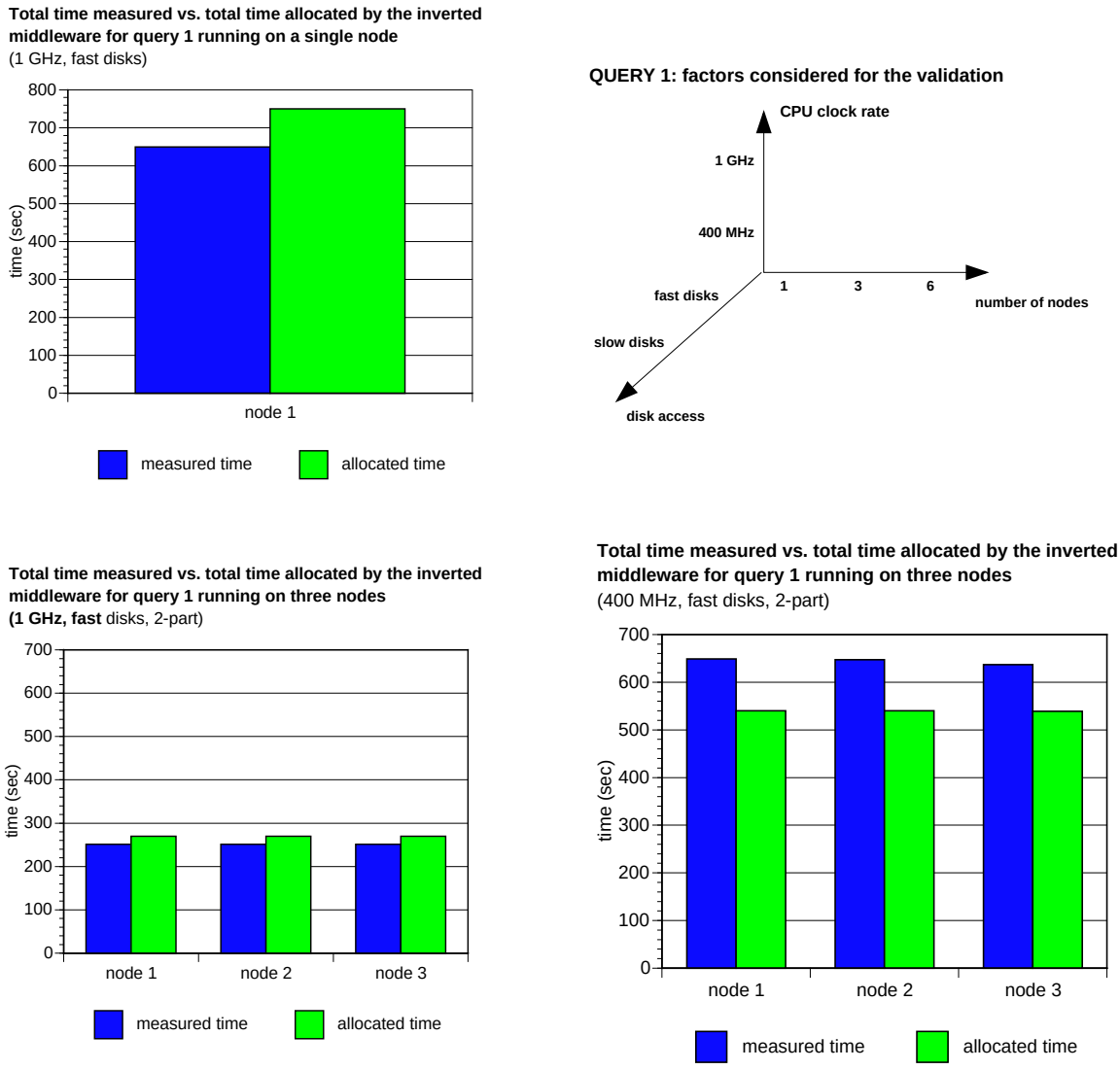

***Figure 8.2**: Validation of the analytical model for query 1 and for different level of CPU clock rates, speeds for disk access and number of nodes. (Part 2. in Figure 8.3)*

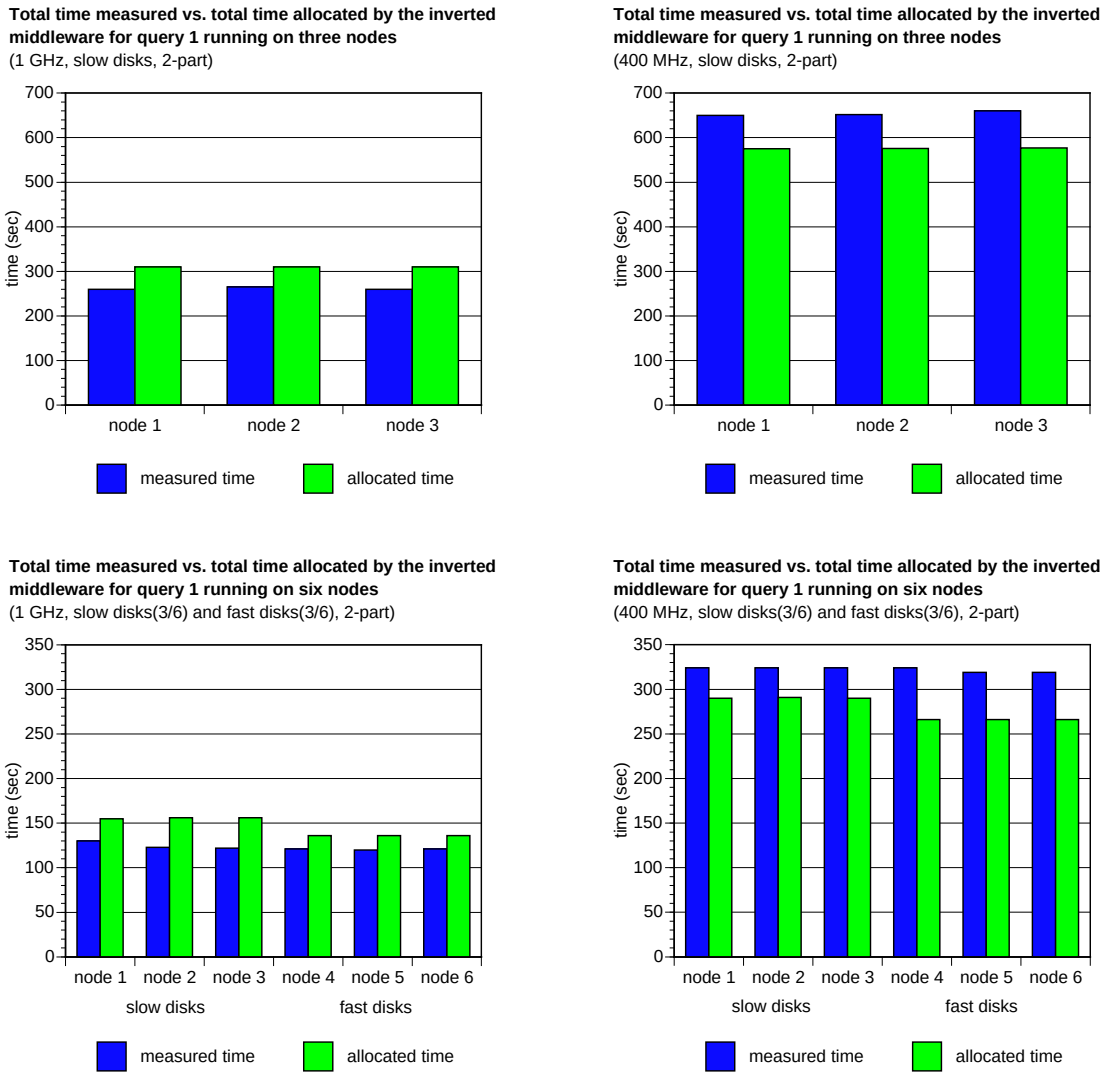

***Figure 8.3**: Validation of the analytical model for query 1 and for different level of CPU clock rates, speeds for disk access and number of nodes.(Part 1. in Figure 8.2)*

access the disks, the number of nodes among which the data of the tables is disjointly distributed and the partitioning schema of the data. Again, there are two kinds of disk access: fast and slow access to the disk. The data is distributed on a single node, three and six nodes. Moreover, we look at two data partitioning schema: the first scheme ($1-part$) partitions only the LineItem fact table, the second one ($2-part$) partitions both fact tables (i.e. LineItem and Orders).

8.3 Using MW^{-1} to Classify Workload according to the Resource Usage

In this section, we look at three important performance issues: the workload characterization of the TPC-D queries, their scalability and their dependency on the network interconnect in a cluster of commodity PCs. With a rigorous performance analysis based on the use of the inverted middleware framework, we present some solid explanations and remedies for the issues and problems found.

8.3.1 Workload Characterization of Distributed TPC-D

The TPC-D benchmark uses 17 different queries that fall into various categories. As database specialists, we would typically classify them into categories based on the number and the extent of joint operations which are needed to work through the database relation tables. Our current approach as system architects and performance evaluation specialists is quite different; we propose an alternative classification of the queries, based on the most critical machine resources (i.e. CPU, disks, and network) used during their execution. We look at one single query at a time and run it in parallel on a shared-nothing architecture in which each machine is comprised of a processor, memory and three disks. The nodes communicate to each other by means of a high-speed interconnection network [92].

The several machine resources in our distributed system (local or remote) differ considerably in speed and availability. In our performance analytical model that is part of the performance monitoring framework, we identify three different machine resources that can be critical limiters to faster parallel execution of a query: the CPU usage, the disk usage and the network usage for inter-node communication in the case of parallel processing. We consider CPU, disk and network usage because of their direct significance to the database applications. Moreover, we add a fourth category for queries whose performance cannot be accurately modeled using the classical resources under investigation. For these queries, instead of a performance critical resource, we have a significant inefficiency in the middleware layer.

The general approach and the related set of resources presented in this chapter is certainly not limited to database applications. We successfully use a similar method for the analysis and modeling of parallel scientific codes (in Chapter 7).

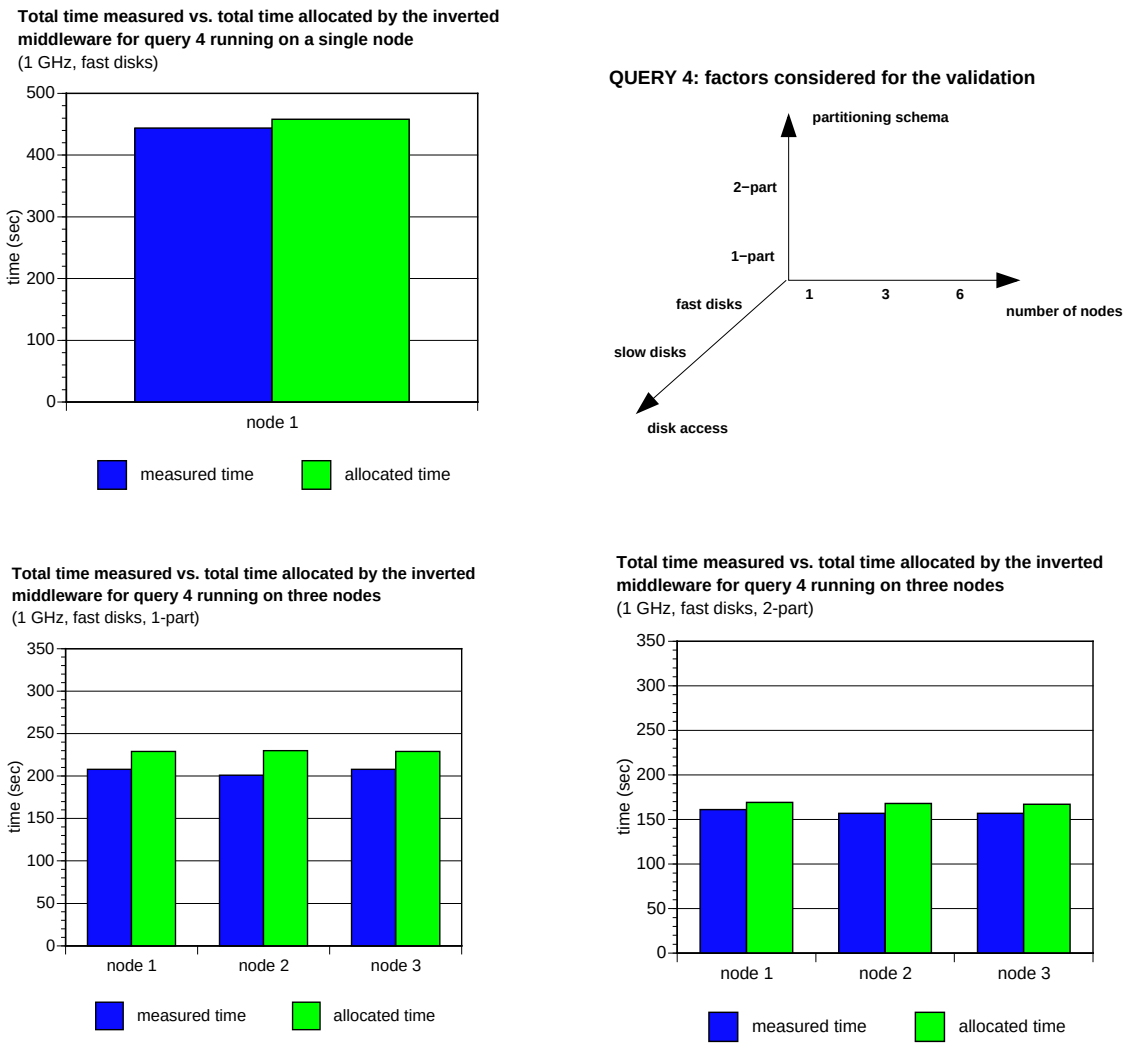

***Figure 8.4**: Validation of the analytical model for query 4 and for different level of CPU clock rates, speeds for disk access and number of nodes. (Part 2. in Figure 8.5)*

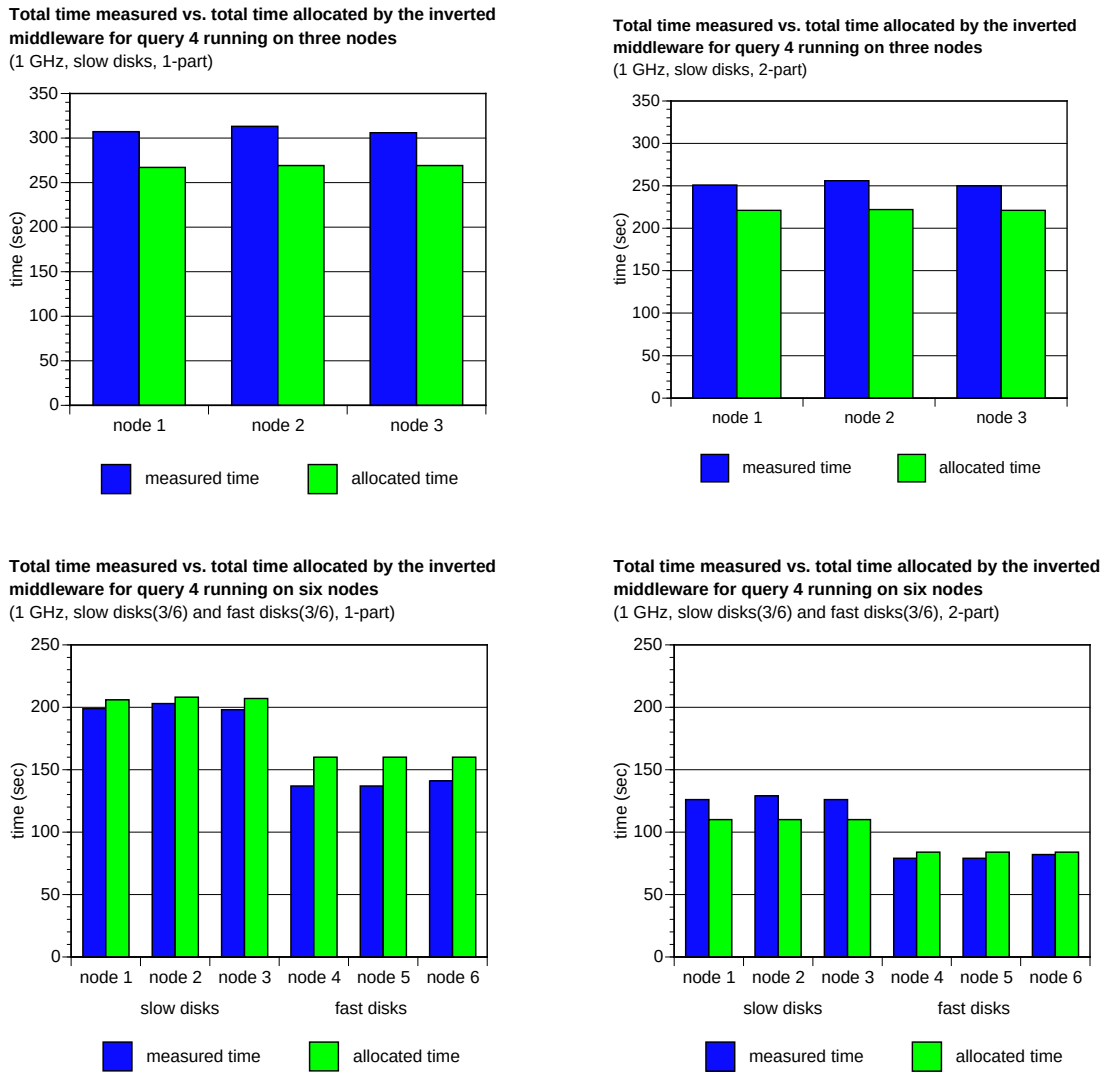

***Figure 8.5**: Validation of the analytical model for query 4 and for different level of speeds for disk access, data partitioning and number of nodes.(Part 1. in Figure 8.4)*

A typical example of a query limited by CPU usage is query 1, a typical example of a query limited by disk usage is query 4. Query 3 exhibits an interesting dependency on the communication subsystem when executed on multiple nodes and is therefore a good example for a query limited by network usage. Finally, query 8 shows the limitation of our performance evaluation techniques since its resource usage appears to be totally inconclusive. Unlike all other cases where the processed output of our monitoring tools and our analytic model explain the execution time within an average error of less than 10% [2], in this case most of the execution time remains without direct allocation to machine resources in the system.

Figure 8.6(a) breaks down the usage time allocated to the specific machine resources

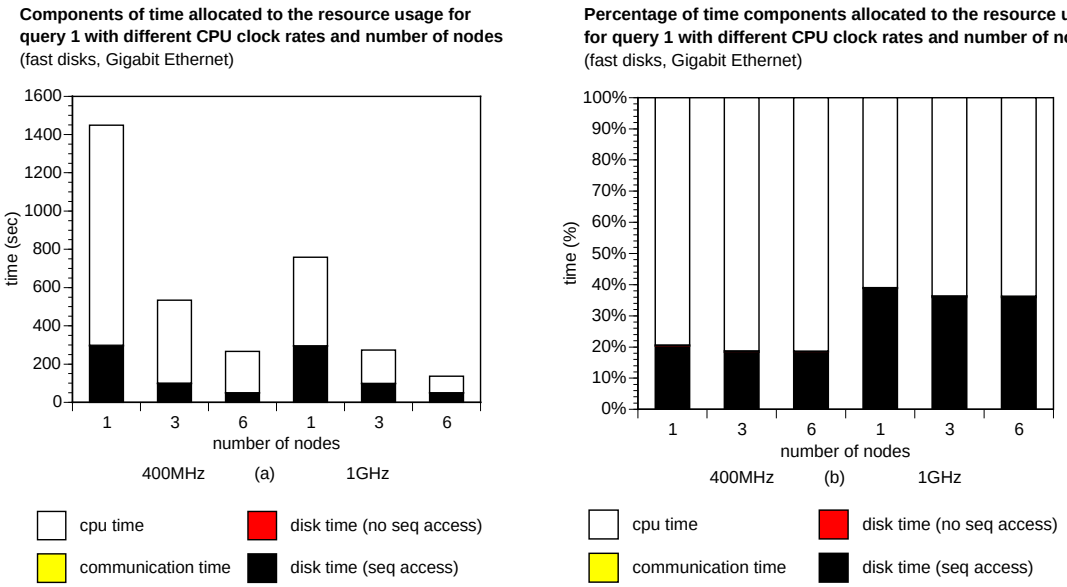

Figure 8.6*: Components of the execution time allocated by the performance modeling framework to the resource usage (CPU, disk and network) for query 1 in seconds (a) and as percentages (b) for different CPU clock rates and different number of nodes. CPU usage for query 1 accounts for 80% to 60% of the execution time, depending on the clock rates of the CPUs used. Network usage and non sequential accesses to the disks are negligible.*

by the performance modeling framework for query 1 with two different generations of PC clusters that differ in the CPU clock rates. We break down the execution time into four components: CPU usage, sequential access and non sequential access to the disk subsystem and inter-node communication. Figure 8.6(b) shows the percentage of these components. The performance modeling framework shows that the CPU usage accounts

[2] We did calibrate the analytical model with numerous measurements and we do have the data to show its accuracy, but we had to omit the figures due to space constraints.

for 80% to 60% of the execution time, depending on the clock rates of the CPUs used. The component of the execution time attributed to CPU usage diminishes by a factor of 2.5 when we upgrade the CPUs from 400MHz to 1 GHz proving that query 1 is largely CPU limited. Looking at the scalability with a growing number of nodes, we state that the fractions of execution times stay the same and that the limiting factor remains the CPU even for larger clusters. The precise extent of scalability will be discussed in the next subsection.

Figure 8.7(a) breaks down the usage time allocated to the specific machine resources by

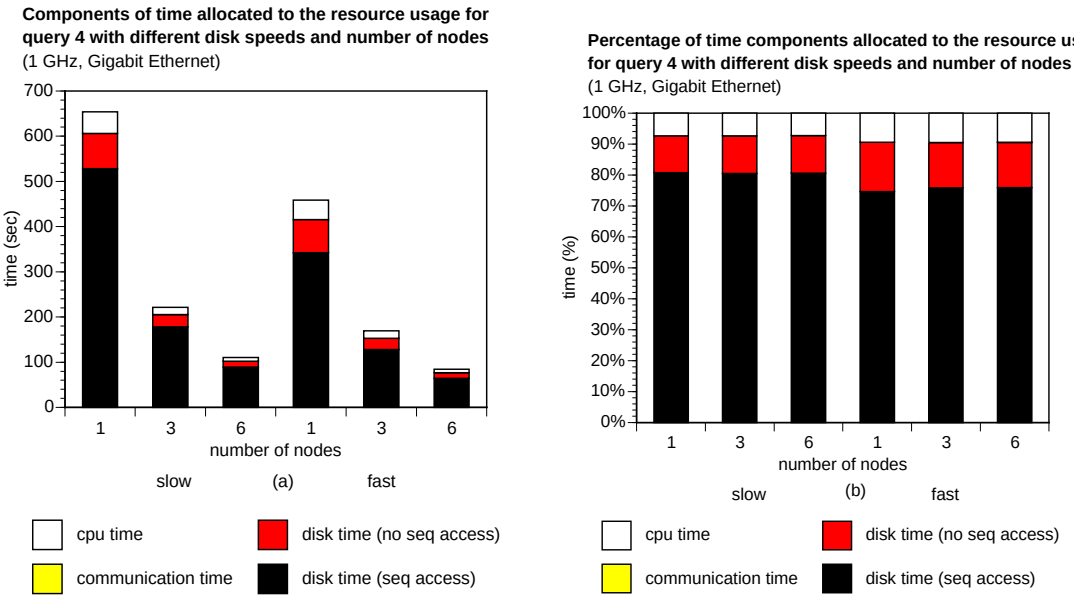

***Figure 8.7**: Components of the execution time allocated by the performance modeling framework to the resource usage (CPU, disk and network) for query 4 in seconds (a) and as percentages (b) for different disk speeds and different number of nodes. Disk usage accounts for more than 90% of the execution time in both cases of disk subsystems.*

our performance modeling framework for query 4 with two different disk types used in the nodes of our clusters. Again, Figure 8.7(b) shows the percentage of the time components. The data provided by the performance modeling framework indicates that the disk usage accounts for more than 90% of the execution time in both cases of disk subsystems. The performance monitoring framework is further capable of distinguishing sequential from non sequential (random) read disk accesses. As expected for OLAP workload, the emphasis is on sequential read access with a ratio of 90%/10% for the slower disks and 85%/15% for the faster disks. Going to faster disks significantly decreases the execution time as can be seen in Figure 8.7(a). The fraction of execution time allocated to disk operations by the performance monitoring framework decreases with faster disks as expected. As for scalability with 1, 3 or 6 nodes, the fractions of CPU-, disk- and network usage

stay the same for all distributed configurations. We identified query 3 as a representative

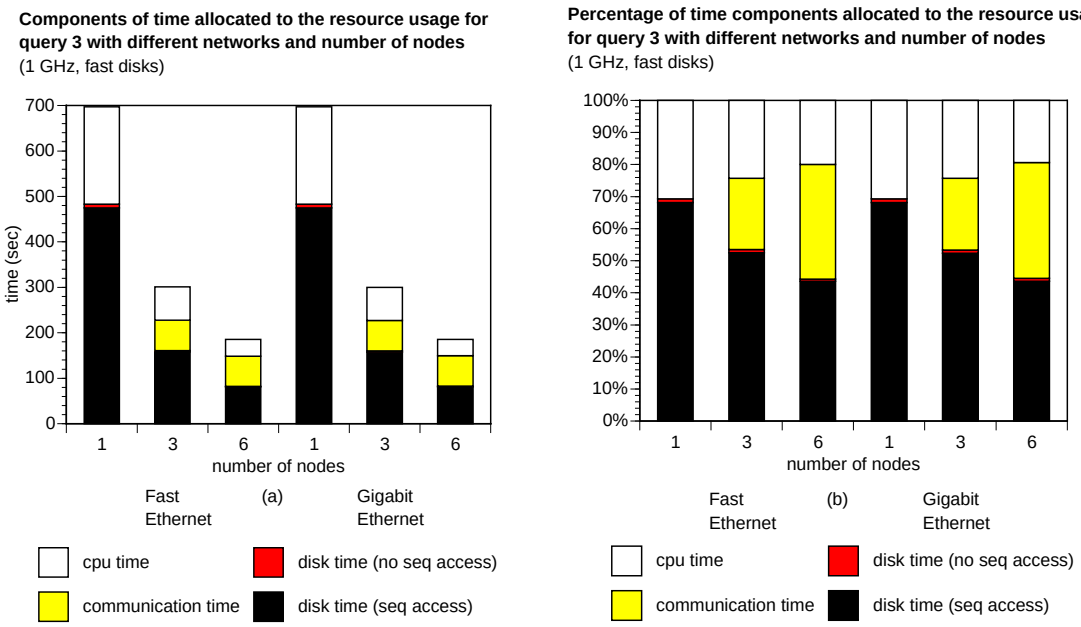

Figure 8.8*: Components of the execution time allocated by the performance modeling framework to the resource usage (CPU, disk and network) for query 3 in seconds (a) and as percentages (b) for different networks (i.e. Fast- and Gigabit Ethernet) and different number of nodes. The fraction of communication time increases with the number of nodes (bad scalability). After deduction of the network part, the 30/70 ratio between CPU and disk remains invariant regardless of the size of the cluster.*

of a network-limited query. Figure 8.8(a) shows the usage time allocated to the specific machine resources by the performance modeling framework for query 3 (i.e. CPU, disk and network usage) with two different network interconnects commonly found in clusters of PCs (i.e. Fast Ethernet and Gigabit Ethernet). As expected, there is no inter-node communication in the uniprocessor case (1 processor) and the communication only becomes visible as we distribute the workload to multiple nodes (3 and 6 nodes) in the PC cluster. The distribution of the resources without the network is about 30% CPU and 70% disk and stays that way for larger clusters pointing at almost linear scalability for the CPU and disk components and at the good explanation of the non scalability by the communication work. Surprisingly, there is no improvement with the addition of Gigabit Ethernet that is 10 times faster than Fast Ethernet. The scalability and the impact of the network will be discussed below - for now we just state the evidence that query 3 is a network-limited query in parallel OLAP applications on clusters of PCs. We classify those queries whose total usage time allocated to the classical machine resources is much shorter than the total execution time measured (less than 15% in average) as DBMS inefficient or DBMS-

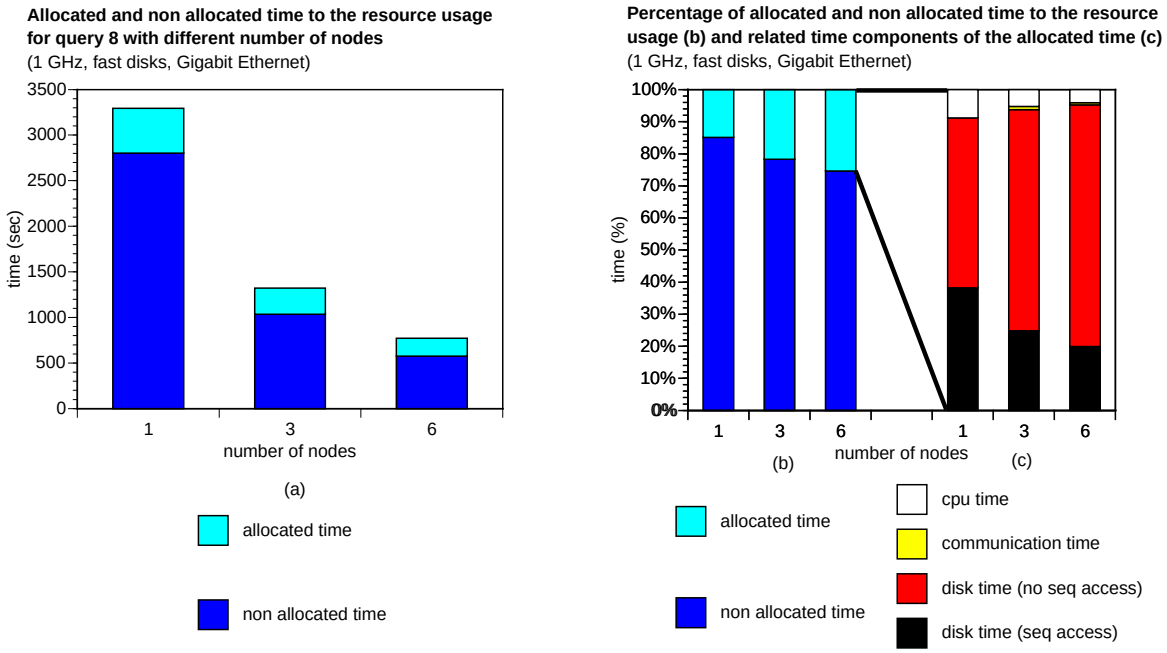

Figure 8.9*: Execution times that* can *be allocated to the use of specific machine resources and time that* cannot *be allocated by our performance monitoring framework in query 8. The time is given in seconds (a) and percentages (b). The allocate-able part is further broken down into resource usage as observed by the performance monitoring framework (c). None of the monitored resources are intensively used. This may be due to ill-conceived resource demands by the DBMS (i.e. the sequence and the amounts in which the different resources are demanded).*

dependent. The performance monitoring framework reports that for such queries, none of the monitored resources are intensively used. The lack of a critical machine resource indicates an internal problem in the DBMS. The non-allocated time is probably due to an ill-conceived sequence in which the different resources are demanded by the DBMS. Certainly, there might be an additional non-monitored resource or a combination of resources that could be the bottleneck, but we did not find any and it remains unlikely that the entire set of monitoring tools of a modern operating system would not record any indication. In parallel scientific codes, we found such behaviors due to load imbalance and due to ill fated synchronization algorithms [103].

Query 8 is characterized by such behavior. Figure 8.9(a), indicates the time allocated to the resource usage by the performance monitoring framework and the non allocated time for this query. We can see that the phenomenon is present to the same extent in the centralized case (1 processor) as well as in the parallel cases (3 or 6 nodes). The time spent based on the profiled workload of the CPU, disks and network covers less than 20% for the query running on a single node and increases only slightly to over the 25% for six

nodes as indicated in Figure 8.9(b). A further breakdown of the time explained by the performance monitoring framework in query 8 is rather inconclusive. The part of query 8 we can explain seems to be disk limited with a large emphasis of random disk accesses. In Figure 8.9(c), it also seems that the sequential accesses to the disk and the CPU usage scale with a larger number of nodes while the non sequential disk accesses do not scale (right figure last three bars). Such a situation indicates a high potential of optimization through tuning parameters of the application code or the DBMS. In fact, we have access to a few intrinsic parameters of the system that affect the performance. In the TP-Lite approach the distribution of the relation tables can be set to a more or less aggressive strategy for partitioning vs. replication. We can use our performance monitoring framework to determine the best partitioning by parameter variation. In general, the standard

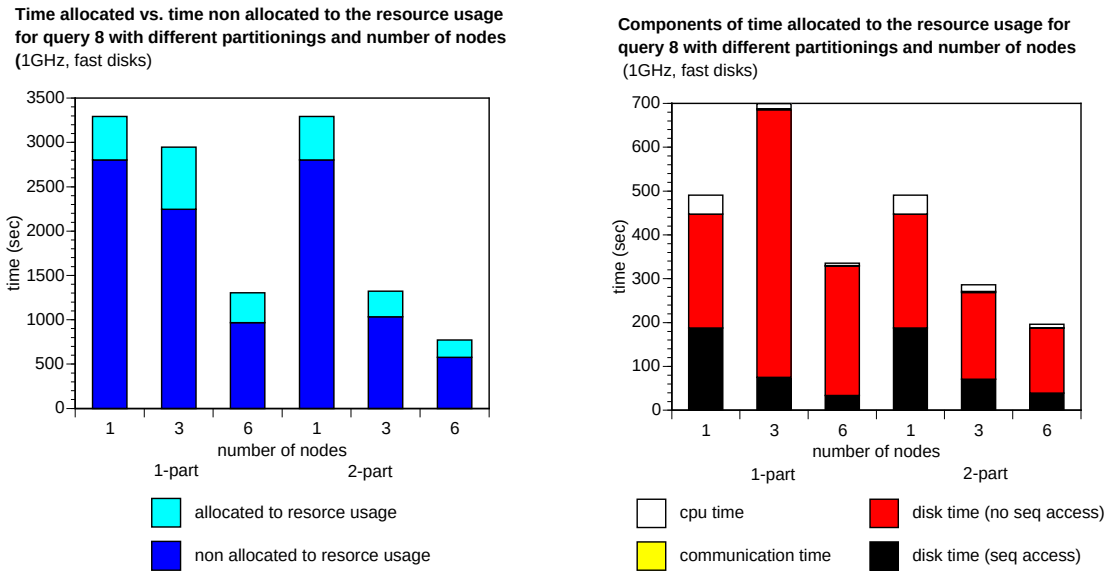

***Figure 8.10**: Execution times that* can *be allocated to the use of specific machine resources and time that* cannot *be allocated by our performance monitoring framework in query 8 in seconds (a) with different partitioning and number of slaves. The allocateable part is broken down into resource usage as observed by the performance monitoring framework (b). The non sequential access to the disks is the most time consuming task during the run of the query.*

parameter setting of the partitioning scheme is a 2-part scheme which results in better scalability. In query 8, we did try another partitioning scheme and report on this case as 1-part schema. The 1-part case resulted in even higher instability in terms of predictable performance. Figure 8.10(a) indicates the execution time allocated to the resource usage by the inverted middleware with the non allocated time for this query with two different internal partitioning schemes. The scalability of the execution time for the 1-part scheme worsened with increasing number of nodes compared to the scalability of the execution

time for the 2-part scheme. Looking at the percentage of allocatable resource usage in Figure 8.10(b), we see that in the case of 1-part schema among three nodes the number of non sequential disk access and the corresponding time component explode and offset any scalability, while in the case of six nodes things improve. This is a fairly typical picture of loss of control, stability and performance of distributed application that relies heavily on layered middleware. Observations like this one prompted us to start our project on more systematic performance engineering in distributed systems using middleware.

The inverted middleware framework, with the collected performance monitoring information combined with an analytical model of resource usage, permits us to characterize the workload of TPC-D benchmark more closely through a classification of the queries according to their individual resource usage. With our framework, we can classify the queries as CPU-limited, disk-limited, communication-limited or inefficient due to middleware limitations.

8.3.2 Scalability of Distributed TPC-D in a Cluster of PCs

Clusters of commodity PCs have become a highly popular platform for distributed computing since simple PC workstations have one of the best price performance ratios in the market for computing equipment. The most important purpose of migrating OLAP applications to larger clusters of PCs is to permit extremely large workloads to execute well on an extremely common and cost effective platform. We chose OLAP as our application since we are more interested in scaling up the problem size than in a large number of clients issuing simultaneous queries. Therefore the scalability of the different TPC-D queries is of a particular interest. We carefully optimized the physical design of our PC cluster in order to achieve minimum response times and a maximum speed-up when scaling to larger clusters. Table 8.1 shows the speed-up of some of the chosen queries.

number of nodes	1	3	6
query 1 (2-part schema)	1	2.6/3	5.4/6
query 4 (2-part schema)	1	2.7/3	5.3/6
query 3 (2-part schema)	1	2.3/3	3.9/6
query 8 (2-part schema)	1	2.4/3	4.1/6
query 8 (1-part schema)	1	1.1/3	2.5/6

***Table 8.1**: Speed-up measured for some of the TPC-D queries chosen.*

Queries 1 and 4 expose almost perfect scalability while queries 3 and 8 do not scale that well. The precise reasons for the scalability or non scalability are explained by the performance analysis framework of inverted middleware. Again, the detailed monitoring information about resource usage is collected and enhanced with an analytical model to give better explanations of what happens in each case.

Figure 8.6(a) and Figure 8.6(b) show that CPU usage is the most important component in the processing of query 1 and that this component scales almost perfectly with a growing number of nodes involved in a distributed processing of this query. The second most important resource limitation is disk usage. Since we use a shared-nothing architecture to distribute the workload, the number of disks increases with the number of nodes involved. The internode communication stays negligible in this query since very small amount of data is transferred. Figure 8.7(a) and Figure 8.7(b) confirm a similar picture for query 4. Processing this query results in heavy disk usage. Again this is invariant to changes in the number of nodes because the total disk performance scales with an increasing number of nodes. The second most used resource is CPU usage which scales perfectly to larger systems.

For query 3, the network usage no longer scales well with the increase of the number of slaves, while the other resources (i.e. CPU usage and sequential access to disk) have good scalability (see Figure 8.8(a) and Figure 8.8(b). The reason for the limited scalability is a growing percentage of execution time due to internode communication, involving 3 and 6 nodes. Furthermore, the time components scale exactly in the same way for 1000 BaseT and 100 BaseT, and therefore the loss of scalability seems to be independent of the bandwidth and speed of the interconnection network.

The amount of communication required to process a distributed query is not necessarily a reason for bad scalability. In our cluster of PCs, the nodes are interconnected with a full crossbar switch and therefore the network resources available to the distributed processing of OLAP workload could actually grow linearly with the number of processing nodes involved. Furthermore, the total amount of communication data reported by the monitors and collected by the inverted middleware framework is fairly small and does not point directly at a bottleneck. Even a slow, shared medium Fast Ethernet (hub) would be able to provide the necessary raw bandwidth to move the data. Therefore a proper communication performance study requires more than the total amount of data transfered.

The global sampling that is accurately synchronized by a virtual wall clock time permits us to identify bursty and unbalanced network traffic in the distributed application. Figure 8.11 on the left shows the communication activity on the coordinator node for query 3 while coordinating the processing of the query with three/six nodes (coordinator/3 and coordinator/6). At the same time the figure displays on the right the communication activity for the same query on the three/six individual processing nodes (node n/3 and node n/6). The data in the figures is gathered at the distribution-specific layer by the performance monitoring framework.

In the top-right charts we consider the communication on each of the three nodes for the processing of the query with three nodes, while in the bottom-right charts we look at the communication on the first and last nodes for the processing of the query with six nodes. The charts related to the processing on three nodes (top) depict the transferred number of bytes per second as a function of time where is denotes the start of query ex-

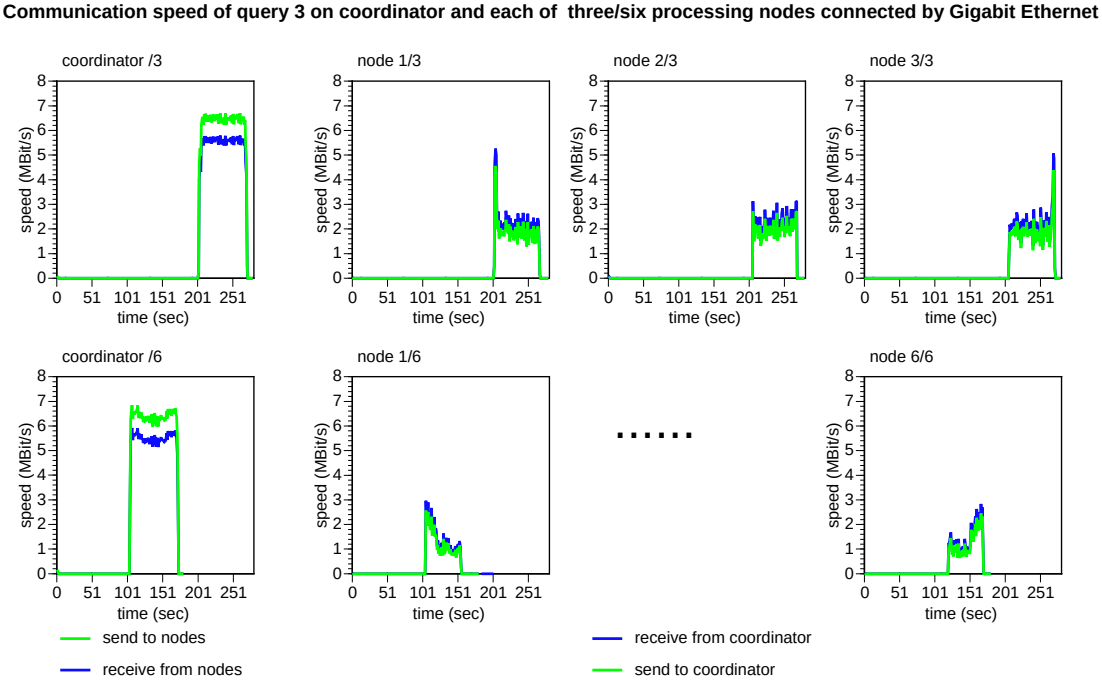

***Figure 8.11**: Communication activity during the the processing of query 3 for three/six nodes on the coordinator node (left) and on each of the three/six processor nodes (right).*

ecution and 280s the end of query execution for three nodes. The same query finishes earlier i.e. roughly at time 170s when the work is distributed among six nodes (see charts on the bottom). In the left charts we look at the communication work on the coordinator and distinguish between “sends to the nodes” and “receives from the nodes”. In the right charts we distinguish between “sends to the coordinator” and “receives from the coordinator”. The graphs show a network traffic that is quite bursty and unbalanced and that most of the traffic is concentrated on the coordinator, while the processing nodes clearly communicate at roughly one third or one sixth of the speed of the coordinator.

The graphs refuted our initial assumption that communication is highly asymmetrical due to the algorithm in TP-Lite that processes partial queries on each node and finally gathers the results in the coordinator. In reality, the communication traffic is quite symmetrical and during the data transfer the coordinator sends almost as many bytes of request as the nodes sends for its answers. Furthermore, the peak communication rates in the coordinator is determined to be just 6.4 MBit/s in a network that can sustain one Gigabit/s (i.e. 160 times more) under good conditions.

For query 8, the degree of scalability is strongly correlated with the partitioning scheme as shown in Figure 8.10(a) and Figure 8.10(b).

The inverted middleware framework provides precise allocation of total execution

time to each resource usage and offers the possibility to record resource usage for arbitrary sampling intervals. This helps in finding and isolating the cause for the loss of scalability in the processing of non scalable queries. An accurate sampling of all communication activity over the entire execution time is required to discover and deal with performance limitation due to bursts and hot spots in the communication patterns.

8.3.3 Performance Impacts of the Network Interconnect Speed

During the evaluation, procurement and installation of a large cluster for database research, we studied the question of the least interconnect technology that is sufficient for the planned work on distributed databases. For a Beowulf type cluster, a commodity Fast Ethernet switch would be sufficient. For enhanced clusters, we would opt for a Fast Ethernet switch that provides full bisection bandwidth (or non blocking ports in networking terminology). For an advanced cluster, we consider high performance interconnects like Gigabit Ethernet or Myrinet. Depending on the performance networking, a cluster node can cost between 100 and 1000 per node installed. Our study should answer the question of whether clusters with higher interconnect speeds are worthwhile for distributed OLAP processing or not. For our measurements, we equipped our cluster of PCs with Gigabit Ethernet and a fully non-blocking 16 port switch in addition to non-blocking Fast Ethernet.

The study of the benefit of a higher speed interconnect is limited to query 3, an example query that obviously becomes communication bound in the distributed configurations of 3 or 6 nodes. As seen in the Figure 8.11, the communication activity is bursty and takes place towards the end of the query. Furthermore, the communication takes place between the coordinator and all the nodes at the same time (two peaks, one for the 3 node experiment and another for the 6 node experiment), but most importantly the speed of communication is just 6-7 MBit/s which is two order of magnitudes below the 100 MBit/s that is possible with Fast Ethernet and three order of magnitudes slower than Gigabit Ethernet. It is not visible from the chart if the inefficiency is due to further burstyness or packet collisions within the sampling interval or if there is software reason for this bad performance and a separate experiment must be set up to verify if there is any benefit at all in using a faster networking technology (which would also handle bursty traffic faster). Figure 8.12(a) and Figure 8.12(b) depict the trace of communication activity on a node (transfers from and to a coordinator) for query 3 in a system of three nodes of cluster connected by Fast Ethernet and Gigabit Ethernet respectively. The two similar traces with completely equal peak and average performance prove that communication behavior is exactly the same for a 100 MBit and a 1 GBit network and that there is no benefit at all in purchasing a network with a higher throughput. Although Fast Ethernet and Gigabit Ethernet differ significantly in the throughput available, the latency and overhead for small packets are not that different.

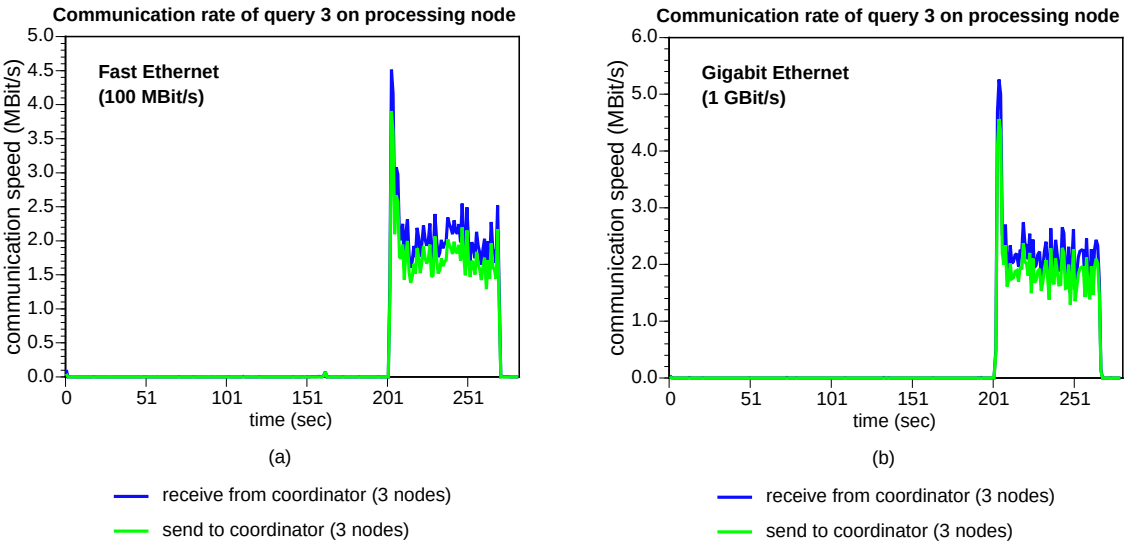

***Figure 8.12**: Slave communication rates of query 3 for Gigabit Ethernet (a) and for Fast Ethernet (b) on a cluster with three slaves and CPUs with 1 GHz clock rate.*

In our first approach for a distributed processing of OLAP database workloads, we used ORACLE database links to ship data from the nodes to the coordinator in the cluster. These database links are based on the ORACLE NET8 transport protocol and use 2PC (two phase commit) to ensure transactional guarantees. Our experiments clearly show that ORACLE database links are not efficient enough to work on large communication loads since they require a lot of slow synchronization. So finally, the latency of the network or the processing overhead in the middleware might become the cause of the slow-down of the communication while the bandwidth and the throughput no longer has any influence on the performance. Moving the application on low latency networks (like Myrinet) could still fail to solve the communication problems because of the high overheads in the middleware. Even if it helps slightly, it is not expected to be cost effective since the bandwidth of Myrinet still remains unused most of the time.

Again, the inverted middleware with its accurate sampling and its global representation of resource usage in the distributed system permits an in-depth analysis of communication problems and properly explains the inefficient use of the network while processing queries like query 3 in TPC-D. With our performance analysis framework, we are finally able to explain why the TP-Lite approach behaved so much worse than the other approaches in [17].

8.4 Architectural Implications of the Resource Usage Observed

Following the classification of the queries reported in the previous section, we can calculate the CPI value from the measured performance data for the different classes of TPC-D queries and find numbers that are quite interesting. CPU limited queries are indicative for the basic CPI value of OLAP workloads, but since we are on a parallel and distributed system we are experiencing several modifications to those basic values. If the nodes processing a query spend a lot of time waiting for "peripherals" such as disks (SCSI cards) or the network (Ethernet cards), the cycles are not attributed to processing instructions but to idle loops in the operating system kernel. Therefore the CPI is inflated when disk or network do most of the work and the CPU is waiting for them.

For strictly CPU-limited queries (e.g. query 1), we measure a CPI of about 1.5 on the older and the newer types of nodes, regardless of clock rate of either 400MHz or 1GHz. Researchers of UCB and UIUC looked at CPI figures for TPC-C and TPC-D on large symmetric shared-memory multiprocessors (SMPs). We are not aware of any studies that deal with massively parallel supercomputer nodes or entire clusters of PCs up to this date.

Keeton *et al.* [67] look at the CPU usage of the TPC-C benchmark using Informix DBMS running on an Quad Pentium Pro, shared-memory, shared-disk configuration. The basic CPI including memory usage is 2.90, which is much higher than the CPI values found in computational benchmarks like SPEC95. Those values are for OLTP and so the difference to our values of 1.4 to 1.5 for OLAP is not surprising. Cao *et al.* [23, 22] look at the characterization of the TPC-D benchmark with Microsoft's SQL Server and Windows NT on top of SMP server with Quad Pentium Pro, shared-memory, shared-disk configuration. They report a basic CPI of 1.27. This slightly lower figure is for a high-end SMP with moderate parallelism, while our work deals with cluster nodes in a massively parallel setting.

We carefully study the dependency of the CPI on the different queries, i.e. the workload. For disk-limited queries (e.g. query 4), the CPU is frequently waiting for the disk accesses, which inflates the overall CPI considerably. The CPI measured on clusters with the faster disks is about 10 while the CPI for slower disks is about 15. Communication-limited queries are unique for parallel and distributed computing platforms. The resulting CPI values of 3 to 4 reflect the idle loops encountered for waiting for data from the network. Finally, the DBMS software limited queries show extremely high CPI values that can only be interpreted by a failure of the monitoring environment to track down the resource usage properly. Query 8 is characterized by a CPI of about 90.

The good values of the CPI for CPU and disk limited queries indicate that a cluster architect should focus on purchasing fast disks and purchasing fast CPUs. This is consistent with the view of processor architects that see database workloads as a major driver for ever better higher MIPS ratings in microprocessors with higher clock rates and more ILP in those database workloads.

Distributing OLAP queries to a large number of nodes with partitioned data can result in large amounts of inter-node communication for certain queries. Unfortunately, the communication is completely dominated by software overheads within the DBMS and adding a faster network does not help. More precisely neither better latencies nor better bandwidth result in much improvement. The precise analysis of the network usage over time indicates that there is a bottleneck due to communication, which takes place between the coordinator and all the nodes at the same time. The bottleneck can be removed by improving the scheduling of the partial queries and by balancing the load communication more carefully. As expected for a database application, the performance of the disks is highly critical to overall performance. Using a distributed file-system with RAID capability based on remote disk accesses may improve the OLAP performance beyond the limitations of simple parallelization.

8.5 Conclusion about the Workload Characterization of TPC-D using MW^{-1}

Performance analysis in parallel databases running on clusters of commodity PCs remains a highly difficult task, since we are still lacking many of the tools and instrumentation that could give us the performance data we need to understand parallel- and distributed systems executing OLAP workloads in standard DBMSs. As a contribution to address this important problem, we use the inverted middleware framework for collecting and filtering the raw performance data given by the operating system in a distributed high performance database system built from common commodity PCs and commercial DBMSs.

Together with a simple analytical model of overall resource usage, we can assess the high level performance indicators at a level of abstraction that is appropriate for an application writer. In particular, the inverted middleware is able to give some new insights about the use of specific machine resources in the system. It is able to capture CPU usage (CPI), two kinds of disk usage (sequential and non sequential) and the communication system usage. Unlike the built-in performance monitors of most database management systems, our operating system based monitoring solution is fully operational in a cluster setting with distributed processing and accounts for accumulated resource usage as well as for a time-variant resource usage by sampling and collecting performance data at arbitrary intervals. Information about peak resource usage and temporary bottlenecks can be derived from the time-variant resource usage traces.

We investigate parallel high performance databases executing OLAP workloads on clusters of commodity PCs and successfully analyze a highly complex hardware-software system that relies primarily on standard hardware and commercial software components that were provided to us by a database research group. The configuration evaluated includes an ORACLE DBMS for SQL processing at each node, the LINUX operating system to manage the node's resources, as well as different Ethernet switches to take care

of inter-node communication. The experimental software shell to distribute the queries to different nodes (TP-Lite) is taken from a database research project.

In our measurement effort, we execute a 10GB TPC-D benchmark and precisely characterize the workload according to the resource usage encountered in the cluster. We gain significant insight into the question of the impact of high performance networking on this sort of application. A most interesting result is that we can classify the queries of the TPC-D benchmark into: CPU-limited, disk-limited, communication-limited or inefficient due to DBMS software limitations. Knowing the critical resource, we can properly explain scalability (or lack thereof) for each query on a cluster with three and six nodes. Surprisingly, the inter-node communication is not the key to better scalability unless the communication facilities of standard DBMS software are drastically improved. In particular, we have shown that the Gigabit high speed network in our high-end cluster of PCs does not improve scalability of the application. This is due to inefficiencies in the commercial DBMS software that is used in conjunction with TP-Lite. Using a simple analytic model that breaks up the total execution time into components attributed to a resource, we can even estimate scalability for larger number of nodes.

9 Conclusions

9.1 Summary and Contributions

Performance analysis in distributed high performance computing systems like clusters of commodity PCs remains a highly difficult task, since we are still lacking tools and instrumentation that could give us the appropriate performance information about a running application program. We need this information to gain insight about the demands and the requirements of the application in respect to the most performance critical machine resources. The lack of control on efficiency in distributed computing systems is mainly due to existing middleware packages which do not provide proper instrumentation for performance analysis related to the distribution of computation on the system. Most middleware packages provide a high level of abstraction of the computing system to the user but, as a result of this, obstruct the detection of performance bottlenecks and the analysis of architectural problems in the distributed systems.

As a contribution to the solution of this essential problem in high performance computing, we present the concept of inverted middleware. The inverted middleware is a performance monitoring framework for performance analysis in distributed computing systems based on the theory of the near-complete decomposability as stated by Courtois.

Due to the near-complete decomposability, we can break up the complex systems into hierarchical layers so that we can either examine interactions within the layers in isolation, i.e. as if interactions among layers would not exist, or interactions among layers without considering the interactions within the layers. Our system contains three hierarchical layers: the operating and network system layer, the middleware layer and the application layer. We prove that our decomposition of the system into layers follows the criteria of Holt and therefore is likely to belong to the group of near-decomposable systems.

So far, the concept of inverted middleware is based on the ideas of modeling and inverting the functionality of the middleware layer rather than instrumenting the middleware or the application source code with additional performance monitoring hooks. Our approach has the advantage that it can be applied to black box middleware systems without source codes, for example an entire ORACLE DBMS, whose source is unavailable, and if it were available, would be far too complex to be modified. Our conceptual approach

to performance analysis by means of the inverted middleware is based on mapping the operating system state with its performance information backwards onto the higher level of abstraction provided by the corresponding middleware package.

In this dissertation, we propose an internal structure of the inverted middleware, we look at the software system services that it provides and at the problems encountered in a first prototype implementation. Our framework for performance monitoring combines monitoring instruments at the operating system level with an effective strategy for collection and interpretation of this monitoring information at the overall system levels. Using some simple analytical models of overall resource usage, the inverted middleware assesses the high level performance indicators at a level of abstraction that is most appropriate for an application writer. In the existing inverted middleware prototype, we characterize the performance behavior in terms of dependency on the most critical machine resources i.e. the CPU, the memory system usage, the disk usage and communication system usage. Unlike the built-in tools for performance monitoring in common database management systems, our monitoring solution is fully operational in a cluster of PCs setting and our framework can account for accumulated resource usage as well as for a time-variant resource usage by sampling and collecting performance data at arbitrary intervals. Information about peak resource usage and temporary bottlenecks are derived from the time-variant resource usage traces.

In our inverted middleware framework, we also address problems related to accuracy and reliability of performance information of the application running on a distributed system. Accurate and reliable performance information can only be delivered in distributed systems by keeping the monitoring intrusions to a minimum. Keeping monitoring intrusions low a priori is much better than removing such intrusions a posteriori by corrections. For keeping a low communication intrusion a priori, we use the UDP/IP protocol rather that the TCP/IP protocol for the performance monitoring traffic in the inverted middleware since this protocol avoids any mutual blocking of the different system components due to flow control. In case some performance information gets lost we treat it as sampling errors. A loose notion of time based on hardware cycle counters in the distributed system allows us to reduce scheduling and execution intrusions.

We demonstrate the viability of the inverted middleware approach for performance analysis and prediction in molecular dynamics. In Chapter 7, we look at the scientific code Dyana for computing protein and nucleic acid structures from distance constraints and torsion angle constraints collected by nuclear magnetic resonance (NMR) experiments. After some calibration, we derive the analytic performance model for Dyana and integrate this model in the inverted middleware framework to extrapolate the application runs to future compute platforms like a widely distributed computing infrastructure incorporating thousands of processors on the Internet (desktop computational grid). The inverted middleware is able to predict excellent scalability for Dyana of up to approximately 42000 processors. At this large number of slaves the communication to the master

becomes a bottleneck. The problem can be overcome by a replication of the master in a multilevel hierarchical setup as a simple alternative to switching over to peer-to-peer paradigms. Applications like Dyana are ill-suited for a peer-to-peer setting since they need a coordinator which gathers intermediate results and based on those results dynamically re-schedules the simulation tasks.

In a second study, we show that our approach to the performance analysis with the inverted middleware is indeed more general and not limited to the domain of scientific computing. In fact, we use our inverted middleware framework to investigate parallel high performance databases executing OLAP workloads on clusters of commodity PCs. We successfully analyze a highly complex hardware-software system that relies primarily on standard hardware and commercial software components provided to us by the database research group at the ETH Zurich and by well-known vendors of DBMSs. The experimental software solution to distribute the queries to different nodes (TP-Lite) is taken from the database research project. The configuration evaluated includes an ORACLE DBMS for SQL processing at each node, the LINUX operating system to manage the node's resources as well as different Ethernet switches to take care of inter-node communication.

In our performance study in Chapter 8, we execute a 10GB TPC-D Benchmark and characterize the workload in terms of precise resource usage encountered in clusters of commodity PCs. As an interesting consequence, we can classify most of the 17 queries of TPC-D into CPU-limited, disk-limited or communication-limited queries leaving a few examples to a fourth class of workload. The queries in this latter class show some inefficiencies which cannot be explained by machine resource limitations but are due to DBMS software limitations. Moreover, because of the information collected and processed by the inverted middleware, we gain significant insight about the impact of high performance networking on the database application.

Furthermore, we properly explain the scalability (or lack thereof) on clusters of PCs with three and six nodes. Based on a simple analytic model that properly factorizes the total execution time into several time components, each attributable to one machine resource, we can estimate scalability for each workload to larger numbers of nodes. The biggest surprise is that the inter-node communication technology in hardware does not determine better or worse scalability unless the communication facilities of standard DBMSs in software are drastically improved. In particular, we have shown that the Gigabit high speed network in high end clusters of PCs does not at all improve scalability of the application and that this problem is due to overwhelming inefficiencies in the commercial DBMS software used in conjunction with TP-Lite.

With the inverted middleware we are able to make proper architectural decisions about the construction of PC clusters for distributed database systems. We also promote distributed database applications by stressing and classifying the multilevel resource demands for such applications on clusters of commodity PCs. Such a classification would

not be possible without the novel approach to the performance analysis pursued by the inverted middleware.

9.2 Further Work

An improved performance analysis in distributed systems can show which resources are potential bottlenecks and can identify further opportunities for system optimization.

In this work, we show that the inverted middleware can be of great help with architecture decisions for clusters of PCs. The framework deals with many important aspects of distributed systems such as realistic networks with contention, sampling accuracy and system perturbation. The framework can be extended to widely distributed systems in grid computing, but a few more scientific challenges must be faced during this research effort.

Moving towards widely distributed systems and assuming complete information about performance and resource usage is still quite unrealistic because of the large amount of data normally involved on computational grids. An adequate global performance assessment of a running application is only possible by interpolating samples of performance data in a statistical rather than a fully deterministic framework. There are some issues that have to be taken into account when adapting and extending the inverted middleware to performance analysis of widely distributed systems like desktop computational grids. First, in widely distributed systems the loss of monitoring data must be accepted. Rebuilding a global performance picture of the system by treating lost performance data as sample errors requires the interpolation of the data in a statistical framework. Second, a flexible, global notion of time is of fundamental importance for the correct interpretation of the partially sampled data. However, the complexity and heterogeneity of the grid architecture work against an easy and effective implementation of the notion of time on such systems. These issues and the resulting impact on the inverted middleware are still open questions.

9.3 Concluding Remarks

As more and more existing application codes in high performance computing will move from expensive supercomputers to more cost effective platforms like clusters of commodity PCs, novel middleware packages could make such a transition possible and easy. These packages will allow us to migrate the applications to the most cost effective systems of the future. But as long as a computation is distributed for the sake of exploiting parallelism and reaching ever higher speeds, the activity of performance analysis, -optimization and -prediction will remain highly critical to the success of these distributed systems. The inverted middleware presented in this thesis addresses this problem by helping the application writers to retain control over performance related issues despite the increasing

complexity of distributed systems using modern middleware. In fact, in this dissertation we prove that our inverted middleware approach leads to more accurate and reliable performance analysis, -modeling and engineering on distributed computing systems than previous methods.

Although our inverted middleware framework certainly has its limitations, a first prototype resulted in many highly interesting insights about possible causes for unstable performance and architectural problems in clusters of commodity PCs. The information provided by the inverted middleware to an application writer helps to re-engineer existing high performance computing applications for new compute platforms in a effective way.

Thanks to the clear insights delivered by the inverted middleware, we can conclude that distributed computer systems based on commodity components are highly successful platforms for high performance computing application in the scientific computation domains (e.g. protein folding, molecular dynamics applications). However, we also have to emphasize as a results of our effort that much more research is needed to make distributed computing systems like clusters of commodity PCs well-suitable for parallel, distributed databases.

We can conclude that the idea of inverted middleware constitutes a solid basis for a priori architectural decisions and a posteriori system optimization in distributed computing systems like clusters of commodity PCs today and the various computational desktop grid architectures of the future.

Bibliography

[1] A. Adl-Tabatabai and T. Gross. Source-Level Debugging of Scalar Optimized Code. In *Proc. of the ACM SIGPLAN'96 Conf. on Prog. Language Design and Implementation*, Philadelphia, Pennsylvania, May 1996.

[2] A. Ailamaki, D.J. DeWitt, M.D. Hill, and D.A. Wood. DBMSs on a Modern Processor: Where does time go? In *Proc. of the 25th International Conference on Very Large Data Bases (VLDB)*, Edinburgh, UK, September 1999.

[3] G. Aloisio, M. Cafaro, C. Kesselman, and R. Williams. Web Access to Supercomputing. *Computing in Science & Engineering*, 3(6):66–72, November/December 2001.

[4] D. Anderson and et al. United Devices - Building the Worlds Largest Computer, one Computer at a Time. http://www.ud.com/.

[5] P. Arbenz, M. Billeter, P. Güntert, P. Luginbühl, M. Taufer, and U. von Matt. Molecular Dynamics Simulations on CRAY Clusters Using the Sciddle-PVM Environment. In *Proc. of EuroPVM 96*, Berlin, Germany, October 1996.

[6] P. Arbenz, W. Gander, H.P. Lüthi, and U. von Matt. Sciddle 4.0: Remote Procedure Calls in PVM. In *Proc. of High-Performance Computing and Networking: International Conference and Exhibition*, pages 820–825, Brussels, Belgium, April 1996.

[7] P. Arbenz, C. Sprenger, H.P. Lüthi, and S. Vogel. Sciddle: A Tool for Large Scale Distributed Computing. *Concurrency: Practice and Experience*, 7:121–146, 1995.

[8] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawake, and C.V. Packer. Beowulf: a Parallel Workstation for Scientific Computation. In *Proc. of 1995 ICPP Workshop on Challenges for Parallel Processing*, Urbana-Champain, Illinois, August 1995.

[9] A. Beguelin and et al. Visualization and Debugging in Heterogeous Environment. *Computer*, 26(6):88–95, June 1993.

[10] G. Bell and J. Gray. High Performance Computing: CRAYs, Clusters, and Centers. What Next? Technical report, Microsoft Research, August 2001.

[11] G. Bell and J. Gray. What's Next in High-Performance Computing? *Communications of the ACM*, 45(2), February 2002.

[12] Beowulf Project Homepage. http://www.beowulf.org/.

[13] F. Berman and R. Wolski. Scheduling from the Perspective of the Application. In *Proc. of 5th IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[14] P.A. Bernstein. Middleware: a Model for Distributed System Services. *Communications of the ACM*, 39(2), February 1996.

[15] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leisserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. 5th ACM Symp. on Principles and Practice of Parallel Prog. (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[16] M. Blumrich, R.D. Alpert, Y. Chen, D. Clark, S. Damianakis, C. Dubnicki, E.W. Felten, L. Iftode, M. Martonosi, and R.A. Shillener. Design Choices in the SHRIMP System: An empirical study. In *Proc. 25th Intl. ACM Symp. on Computer Architecture (ISCA)*, pages 330–341, Barcelona, Spain, June 1998.

[17] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the Coordination Overhead of Synchronous Replica Maintenance in a Cluster of Databases. In *Proc. of the 6th International Euro-Par Conference*, pages 435–444, Munich, Germany, August 2000.

[18] C.A. Bohn, G.B. Lamont, J.K. Little, and R.A. Raines. Asymmetric Load Balancing on a Heterogeneous Cluster of PCs. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Monte Carlo Resort, Las Vegas, Nevada, June 1999.

[19] R. Brand. Pentium II Performance Counting Library. Technical report, Swiss Federal Institute of Technology, Zurich, Switzerland, 1999.

[20] S. Brauss, M. Lienhard, J. Nemecek, A. Gunzinger, M. Näf, M. Frey, M. Heimlicher, A. Huber, P. Müller, and R. Paul. An Efficient Communication Architecture for Commodity Supercomputers. In *Proc. of the SC99 Conference*, Portland, Oregon, November 1999.

[21] H. Brunst, H.-Ch. Hoppe, W.E. Nagel, and M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach . In *Proc. of ICCS2001*, San Francisco, California, May 2001.

[22] Q. Cao, J. Torrellas, and H. Jagadish. Unified Fine-Granularity Buffering of Index and Data: Approach and Implementation. In *Proc. of IEEE International Conference on Computer Design (ICCD'00)*, pages 175–186, Austin, Texas, September 2000.

[23] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed Characterization of a Quad Pentium Pro Server Running TPC-D. In *Proc. of International IEEE Conference on Computer Design (ICCD '99)*, pages 108–117, Austin, Texas, October 1999.

[24] K. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.

[25] A. Chien and et. al. Entropia - High Performance Internet Computing. http://www.entropia.com.

[26] F. Chism. Communication Latency and Bandwidth on the CRAY Research T3E. In *Proc. 10th Intl. Parallel Processing Symposium*, pages Slides, Vendor Presentation, Honolulu, HI, April 1996. IEEE.

[27] D.E. Comer. *Internetworking with TCP/IP. Volume I. Principles, Protocols, and Architecture*. Third Edition. Prentice Hall, 1995.

[28] D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP. Volume III. Client-Server Programming and Applications*. Second Edition. Prentice Hall, 1996.

[29] Oracle Corporation. Oracle8. http://www.oracle.com/, 1997.

[30] P.J. Courtois. *Decomposability—Queueing and Computer System Applications*. Academic, London, U.K., 1977.

[31] P.J. Courtois. On Time and Space Decomposition of Complex Structures. *Communications of the ACM*, 28(6), June 1985.

[32] F. De Ferreira Rezende and K. Hergula. The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways. In *Proc. of the Twenty-Fourth International Conference on Very-Large Databases*, pages 146–57, San Francisco, California, 1998.

[33] L.A. De Rose and D.A. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proc. of the 1999 International Conference on Parallel Processing (ICPP'99)*, pages 311–318, Aizu-Wakamatsu, Japan, September 1999.

[34] E.W. Dijkstra. The Structure of the THE Multi-programming System. *Communications of the ACM*, (11), 1968.

[35] E.W. Dijkstra. Complexity Controlled by Hierarchical Ordering of Function and Variability. In Peter Naur and Brian Randell, editors, *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, pages 181–185. NATO Scientific Affairs Division, January 1969.

[36] E.W. Dijkstra. Notes on Structured Programming. In *Structured Programming*. Academic Press, 1969.

[37] E.W. Dijkstra. Hierarchial Ordering of Sequential Processes. *Acta Informatica, Springer Verlag (Heidelberg, FRG and NewYork NY, USA) Verlag*, 1(2), October 1971.

[38] J. Dongarra. Performance of various Computers using Standard Linear Equations software in a Fortran Environment. In *Proc. of the Third Conference on Multiprocessors and Array Processors*, San Diego, California, January 1987.

[39] J. Dongarra, S. London, K. Moore, P. Mucci, and D. Terpstra. Using PAPI for Hardware Performance Monitoring on LINUX Systems. In *Proc. of Linux Clusters: The HPC Revolution – A conference for high-performance Linux cluster users and system administrators*, National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana, Illinois, June 2001.

[40] C. Dye. *Oracle Distirbuted Systems*. 1st Edition. O'Reilly, 1999.

[41] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[42] G.C. Fox. What have we learnt from using Real Parallel Machines to solve Real Problems? In *Proc. of the 3rd conference on Hypercube concurrent computers and applications*, pages 897–955, Pasadena, California, January 1988.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional Computing Series, 1995.

[44] J.A. Gannon, K.J. Williams, M.S. Andersland, J.E. Lummp, Jr., and T.L. Casavant. Using Perturbation Tracking to Compensate for intrusion in Message-Passing Systems. In *Proc. of the 14th International IEEE Conference on Distributed Computing Systems*, pages 414–423, Los Alamitos, California, June 1994.

[45] W.J. Gehrin, Y.Q. Quian, M. Billeter, K. Furukubo-Tokunaga, A.F. Schier, D. Resendez-Perez, M. Affolter, G. Otting, and K. Wthrich. Hydration and DNA Recognition. *Cell*, pages 211–223, 1994.

[46] A. Geist. PVM on Pentium Clusters Communication spanning NT and Unix. http:www.scl.ameslab.gov/workshops/Talks/Geist/sld001.htm.

[47] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[48] J. Gosling et al. *The Java Language Specification*. GOTOP Information Inc., 1996.

[49] T. Grabs, K. Böhm, and H.-J. Schek. High-level Parallelisation in a Database Cluster: a Feasibility Study Using Document Services. In *Proc. of ICDE*, Heidelberg, Germany, April 2001.

[50] Portland Group. *PGI Workstation User's Guide*. http://www.pgroup.com/ppro_docs, /pgiws_ug/pgiwsu.htm.

[51] P. Güntert. Structure Calculation of Biological Macromolecules from NMR Data. *Quart. Rev. Biophys.*, 31:145–237, 1998.

[52] P. Güntert, C. Mumenthaler, and K. Wüthrich. Torsion Angle Dynamics for NMR Structure Calculation with the new Program Dyana. *J. Mol. Biol.*, 273:283–298, 1997.

[53] M. Gurry. *Oracle SQL Tuning Pocket Reference*. O'Reilly, 2001.

[54] M. Heath. Recent Developments and Cases Studies in Performance Visualization Using ParaGraph. In *Performance Measurement and Visualization of Parallel Systems*, pages 175–200. Elsevier Science Publishers, Amsterdam, The Netherlands, 1993.

[55] M. Heath and J. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

[56] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgen Kaufmann Publishers, 1995.

[57] J. Hoeflinger, B. Kuhn, W.E. Nagel, P. Petersen, H. Rajic, S. Shah, J. S. Vetter, M. Voss, and R. Woo. An Integrated Performance Visualizer for MPI/OpenMP Programs. In *Proc. of WOMPAT2001*, West Lafayette, Indiana, July 2001.

[58] J. K. Hollingsworth. Finding Bottlenecks in Large Scale Parallel Programs. Technical report, Ph.D. Thesis, University of Wisconsin - Madison, 1994.

[59] J.K. Hollingsworth and P.J. Keleher. Prediction and Adaptation in Active Harmony. *Cluster Computing*, 2:195–205, 1999.

[60] R.C. Holt. Structure of Computer Programs: A Survey. In *Proc. IEEE 63*, pages 876–893. IEEE, 1975.

[61] A. Jain, N. Vaidehi, and G. Rodriguez. A Fast Recursive Algorithm for Molecular Dynamics Simulation. *J. Comp. Phys.*, 106:258–268, 1993.

[62] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, 1996.

[63] H. Jakiela. Performance Visualization of a Distributed System: A Case Study. *Computer*, 28(11):30–36, November 1995.

[64] D. Jefferson and H. Sowizral. Fast Concurrent Simulation With The Time Warp Mechanism. In *Proc. of 1985 SCS Multiconference on Distributed Simulation*, January 1985.

[65] N.R. Jennings. An Agent-based Approach for Building Complex Software Systems - Why agent-oriented approaches are well suited for developing complex, distributed systems. *Communication of the ACM*, 44(4):35–41, Apr. 2001.

[66] KAI Software, a division of Intel Americas. GuideView Performance Anlayzer. http://www.kai.com/parallel/kapro/guideview, 2001.

[67] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proc. of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 15–26, New York, June 1998.

[68] K. Kennedy et al. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *Proc. of the International Parallel and Distributed Processing Symposium Workshop (IPDPS NGS)*, San Francisco, CA, April 2002.

[69] T. Kistler and M. Franz. The Case for Dynamic Optimization: Improving Memory-Hierarchy Performance by Continuously Adapting the Internal Storage Layout of Heap Objects at Run-Time. Technical Report 99–21, University of California, Irvine, May 1999.

[70] T. Kistler and M. Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[71] A. Krall, A. Ertl, and M. Gschwind. JavaVM Implementation: Compilers versus Hardware. In *Proc. of 3rd Australasian Computer Architecture Conference (ACAC'98)*, Perth, Western Australia, Australia, February 1998.

[72] Ch. Kurmann and T. Stricker. ECT - Extended Copy Transfer Characterization. http://www.cs.inf.ethz.ch/CoPs/ECT/.

[73] Ch. Kurmann and T. Stricker. Characterizing Memory System Performance for local and remote Accesses in high-end SMPs, low-end SMPs and Clusters of SMPs. In *7th Workshop on Scalable Memory Multiprocessors ACM Trans. Computer Systems. Held in conjunction with ISCA98*, Barcelona, Spain, June 1998.

[74] D.S. Linthicum. Middleware Performance. *DBMS*, 11(9):22, August 1998.

[75] K. London, J. Dongarra, S. Moore, P. Mucci, and T. Seymour, K.and Spencer. End-user Tools for Application Performance Analysis, Using Hardware Counters. In *International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix (Mesa), Arizona, April 2001.

[76] P. Luginbühl, P. Güntert, and M. Billeter. *Opal: User's Manual Version 2.2*. ETH Zürich, Institut für Molekularbiologie and Biophysik, Zurich, Switzerland, 1995.

[77] A.D. Malony and D.A. Reed. Models for Performance Perturbation Analysis. In *Proc. of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, pages 15–25, Santa Cruz, California, December 1991.

[78] H. Mathkour. An Intelligent Tool for Boosting Database Performance. *Journal of King Saud University (Computer and Information Sciences)*, 10:81–106, 1998.

[79] J.A. McCann and K.J. Manning. Tool to Evaluate Performance in Distributed Heterogeneous Processing. In *Proc. of the Sixth IEEE Euromicro Workshop on Parallel and Distributed, Processing (PDP'98)*, Los Alamitos, California, January 1998.

[80] W. E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996.

[81] OMG. Common Object Request Broker Architecture (CORBA/IIOP). http://www.omg.org, May 2002.

[82] D.L. Parnas. On a Buzzword: Hierarchical Structure. In *Proc. of IFIP Congr. 1974*, pages 336–339, Amsterdam, The Nederland, 1974.

[83] J. Postel. User Datagram Protocol. Technical Report RFC 768, ISI, August 1980.

[84] P. Ranganathan, K. Gharachorloo, S. Adve, and L. Barroso. Performance of Database Workloads on Shared-Memory Systems with out-of-order Processors. In *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.

[85] F. Rauch, Ch. Kurmann, and T. Stricker. Partition Repositories for Partition Cloning - OS Independent Software Maintenance in Large Clusters of PCs. In *Proc. of the IEEE International Conference on Cluster Computing 2000*, Chemnitz, Germany, November 2000.

[86] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. of the Scalable Parallel Libraries Conference (SPLC'93)*, October 1993.

[87] R.L. Ribler, H. Simitci, and D.A. Reed. The Autopilot Performance-Directed Adaptive Control System. *Generation Computer Systems, Special Issue (Performance Data Mining)*, 18(1):175–187, 2001.

[88] U. Röhm, K. Böhm, and H.-J. Schek. OLAP Query Routing and Physical Design in a Database Cluster. In *Proc. of the International Conference on Extending Database Technology (EDBT)*, Konstanz, Germany, March 2000.

[89] U. Röhm, K. Böhm, and H.-J. Schek. Cache-Aware Query Routing in a Cluster of Databases. In *Proc. of the International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001. IEEE Computer Society.

[90] G. Roos. Computation of Protein Structure with Dyana on a Cluster of PCs. Technical report, ETH Zurich, 2000.

[91] S. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proc. 7th. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Boston, MA, Oct 1996. ACM.

[92] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill International Editons, 1997.

[93] H.A. Simon and A. Ando. Aggregation of Variables in Dynamic Systems. In *Econometrica 29*, pages 876–893, 1961.

[94] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI - The complete Reference*. MIT Press, 1997.

[95] P.G. Sobalvarro, S. Pakin, E.W. Weihl, and A.A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proc. of the International Parallel Processing Symposium (IPPS '98)*, March 30-April 3 1998.

[96] P. Stähli. Extention of a Performance Counter Library for Pentium Series. Technical report, Swiss Federal Institute of Technology, Zurich, Switzerland, 2001.

[97] T. Sterling, D. Becker, J. Dorband, D. Savarese, U. Ranawake, and C. Packer. Beowulf: A Parallel Workstation for Scientific Computation. In *Proc. of the 24th International Conference on Parallel Processing*, pages 11–14, Oconomowoc, WI, 1995.

[98] R.W. Stevens. *TCP/IP Illustred, Volume I. The Protocols*. Addison-Wesley Professional Computing Series, 1999.

[99] F.J. Suarez, J. Garcia, J. Entrialgo, D.F. Garcia, S. Graffa, and P. De Miguel. A Toolset for Visualization and Analysis of Parallel real-time Embedded Systems based on fast Prototyping Techniques. In *Proc. of the Sixth Euromicro Workshop on Parallel and Distributed Processing (PDP'98)*, pages 186–194, Los Alamitos, California, 1998.

[100] X.H. Sun, M. Pantano, T. Fahringer, and Z. Zhan. SCALA: a Framework for Performance Evaluation of Scalable Computing. In *Parallel and Distributed Processing. 11th IPPS/SPDP'99 Workshops. Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 49–62, Berlin, Germany, 1999.

[101] M. Taufer. Parallelization of the Software Package Opal for the Simulation of Molecula Dynamics. Technical report, Diploma Thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 1996.

[102] M. Taufer, E. Perathoner, A. Cavalli, A. Caflish, and T. Stricker. Performance Characterization of a Molecular Dynamics Code on PC Clusters. Is there any easy parallelism in CHARMM? In *Proc. of IPDPS, IEEE and ACM International Parallel and Distributed Processing Symposium*, Fort Lauterdale, Florida, Apr 15-19 2002.

[103] M. Taufer and T. Stricker. Accurate Performance Evaluation, Modelling and Prediction of a Message Passing Simulation Code based on Middleware. In *Proc. of the SC98 Conference*, Orlando, Florida, November 1998.

[104] M. Taufer, T. Stricker, G. Roos, and P. Güntert. On the Migration of the Scientific Code Dyana from SMPs to Clusters of PCs and on to the Grid. In *Proc. of the CCGRID 2002, IEEE International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.

[105] M. Taufer, T. Stricker, and R. Weber. Scalability and Resource Usage of an OLAP Benchmark on Clusters of PCs. In *Proc. of the Symposium of Parallel Algorithms and Architectures, ACM SPAA '02*, Winnipeg, Manitoba, Kanada, August 2002.

[106] The Transaction Processing Performance Council. TPC-C. http://www.tpc.org, 2001.

[107] The Transaction Processing Performance Council. TPC-D. http://www.tpc.org, 2001.

[108] B. Topol and J. Stako. Integrating Visualization Support Into Distributed Computing Systems. Technical report, Graphics, Visuaization, and Usability Center, Giorgia Inst. Technology, Atlanta, October 1994.

[109] J.P. Tsai, K-Y. Fang, and H-Y. Chen. A Noninvasive Architecture to Monitor Real-Time Distributed Systems. *Computer*, 23(3):11–23, March 1990.

[110] J.S. Vetter and D.A. Reed. Managing Performance Analysis with Dynamic Statistical Projection Pursuit. In *Proc. of the SC99 Conference*, Portland, Oregon, November 1999.

[111] VMware – Virtual Machine Software. http://www.vmware.com/.

[112] F. Vraalsen, R. A. Aydt, C. L. Mendes, and D. A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior. In *Proc. of the 2nd International Workshop on Grid Computing/LNCS (GRID 2001)*, Denver, Colorado, November 2001.

[113] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. of the 1998 Conference on Supercomputing*, Orlando, FL, November 1998.

[114] U. K. Wiil and P. J. Nürnberg. Evolving Hypermedia Middleware Services: Lessons and Observations. In *Proc. of the 1999 ACM Symposium on Applied Computing (SAC '99)*, pages 427–436, San Antonio, Texas, February 1999.

[115] W. Wu, R. Gupta, and M. Spezialetti. Experimental Evaluation of On-line Techniques for Removing Monitoring Intrusion. In *Proc. of the 2nd SIGMETRIC Symposium on Parallel and Distributed Tools*, pages 30–39, Welches, Oregon, August 1998.

[116] W. Wu, M. Spezialetti, and R. Gupta. On-Line Avoidance of the Intrusive Effects of Monitoring on Runtime Scheduling Decisions. In *Proc. of IEEE-CS 16th International Conference on Distributed Computing Systems*, pages 216–223, Hong Kong, China, May 1996.

[117] W. Wu, M. Spezialetti, and R. Gupta. On-line Avoidance of Communication Intrusion in Token Ring Networks. In *Proc. of the 9th IASTED Int'l Conf. Parallel and Distributed Computing and Systems*, pages 429–436, Washington D.C., Virginia, October 1997.

[118] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. of the 1996 Conference on Supercomputing*, Pittsburgh, Pennsylvania, November 1996.

Curriculum Vitae

23. April 1971	Born in Feltre (BL), Italy Citizen of Imer (TN), Italy Daughter of Pia Gaio and Giovanni Taufer
1990–1996	Studies in Computer Science University of Padua, Italy
1996	Research visit at the Swiss Center for Scientific Computing, Swiss Federal Institute of Technology, Zurich, Switzerland
1996	Diploma degree in Computer Science, University of Padua, Italy
1996–2002	Research and teaching assistant at ETH Zurich, Institute for Computer Systems, Laboratory for Parallel and Distributed Systems, in the research group of Prof. Thomas M. Stricker