EFFECTS OF COMMUNICATION PROTOCOL STACK OFFLOAD
ON PARALLEL PERFORMANCE IN CLUSTERS

By

Boris Vladimirovich Protopopov

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2003
EFFECTS OF COMMUNICATION PROTOCOL STACK OFFLOAD
ON PARALLEL PERFORMANCE IN CLUSTERS

By

Boris Vladimirovich Protopopov

Approved:

Anthony Skjellum
Associate Professor of Computer Science and Engineering
(Major Professor)

Donna S. Reese
Associate Professor of Computer Science and Engineering
(Committee Member)

Susan M. Bridges
Professor of Computer Science and Engineering
(Committee Member)

Arkady Kanevsky
Adjunct Associate Professor of Computer Science and Engineering
(Committee Member)

Ioana Banicescu
Associate Professor of Computer Science and Engineering
(Committee Member)

A. Wayne Bennett
Dean of the College of Engineering
Name: Boris Vladimirovich Protopopov

Date of Degree: August 2, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: EFFECTS OF COMMUNICATION PROTOCOL STACK OFFLOAD ON PARALLEL PERFORMANCE IN CLUSTERS

Pages in Study: 166

Candidate for Degree of Doctor of Philosophy

The primary research objective of this dissertation is to demonstrate that the effects of communication protocol stack offload (CPSO) on application execution time can be attributed to the following two complementary sources. First, the application-specific computation may be executed concurrently with the asynchronous communication performed by the communication protocol stack offload engine. Second, the protocol stack processing can be accelerated or decelerated by the offload engine. These two types of performance effects can be quantified with the use of the degree of overlapping $D_o$ and degree of acceleration $D_{accs}$. The composite communication speedup metric $S_{comm}(D_o, D_{accs})$ can be used in order to quantify the combined effects of the protocol stack offload.

This dissertation thesis is validated empirically. The degree of overlapping $D_o$, the degree of acceleration $D_{accs}$, and the communication speedup $S_{comm}$ characteristic of the system configurations under test are derived in the course of experiments performed for
the system configurations of interest. It is shown that the proposed metrics adequately describe the effects of the protocol stack offload on the application execution time.

Additionally, a set of analytical models of the networking subsystem of a PC-based cluster node is developed. As a result of the modeling, the metrics $D_o$, $D_{accs}$, and $S_{comm}$ are obtained. The models are evaluated as to their complexity and precision by comparing the modeling results with the measured values of $D_o$, $D_{accs}$, and $S_{comm}$.

The primary contributions of this dissertation research are as follows. First, the metric $D_{accs}$ and $S_{comm}$ are introduced in order to complement the $D_o$ metric in its use for evaluation of the effects of optimizations in the networking subsystem on parallel performance in clusters. The metrics are shown to adequately describe CPSO performance effects. Second, a method for assessing performance effects of CPSO scenarios on application performance is developed and presented. Third, a set of analytical models of cluster node networking subsystems with CPSO capability is developed and characterised as to their complexity and precision of the prediction of the $D_o$ and $D_{accs}$ metrics.
DEDICATION

To my mother and father
ACKNOWLEDGMENTS

I would like to take this opportunity to express my deep feeling of gratitude to the many people whose involvement and facilitation has made the completion of this dissertation research possible. I would like to acknowledge Dr. Anthony Skjellum for his academic and scientific advice throughout my graduate studies and especially during the dissertation research. I would also like to thank Dr. Susan Bridges for her thorough review of the proposed research materials and her insightful comments and suggestions. My sincere thanks to Dr. Donna Reese for her help in focusing the scope of the research thesis. I would like to thank Dr. Arkady Kenevsky for his comments on the scope and the applicability of the performance analysis techniques used in the research, and I would like to thank Dr. Ioana Banicescu for her reviews and practical remarks on the formulation of the research results.

I would like to thank Dr. Rossen Dimitrov for his time spent discussing the details of the research, sharing his views and the research tools he previously developed. I would also like to thank my employer, Mercury Computer Systems Inc., for the financial support of my study. I would like to thank Mr. Randall W. Dean with Mercury Computer Systems, Inc. for his support and facilitation in the practical matters of setting up video conferences. I would like to thank my colleagues at the French Development Office, Mercury Computer Systems, Inc. for their help with Linux TCP/IP zero-copy patch.
Finally, I would like to thank my fiancée Sarah Otterson. Without her arduous proof-reading, continued support, and understanding, this work would hardly see the day of completion.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1 System under study</td>
<td>4</td>
</tr>
<tr>
<td>1.1.2 Performance metrics and workload definitions</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Problem statement</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Dissertation thesis statement</td>
<td>9</td>
</tr>
<tr>
<td>1.4 Justification of study</td>
<td>11</td>
</tr>
<tr>
<td>1.5 Scope of the dissertation research</td>
<td>12</td>
</tr>
<tr>
<td>1.6 Dissertation research methodology</td>
<td>13</td>
</tr>
<tr>
<td>1.7 Organization of the dissertation</td>
<td>14</td>
</tr>
<tr>
<td>II. PERFORMANCE ANALYSIS OF NETWORKING SUBSYSTEMS</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Approaches to reducing TCP/IP stack overhead</td>
<td>17</td>
</tr>
<tr>
<td>2.1.1 Quantifying the TCP/IP protocol stack overhead</td>
<td>18</td>
</tr>
<tr>
<td>2.1.2 End-system optimizations</td>
<td>20</td>
</tr>
<tr>
<td>2.1.3 Communication protocol stack optimizations</td>
<td>22</td>
</tr>
<tr>
<td>2.1.4 Communication protocol stack offload</td>
<td>25</td>
</tr>
<tr>
<td>2.2 Networking subsystem performance optimizations</td>
<td>31</td>
</tr>
<tr>
<td>2.2.1 Revising hardware/software system interfaces</td>
<td>33</td>
</tr>
<tr>
<td>2.2.2 Minimizing overhead of runtime environments</td>
<td>35</td>
</tr>
<tr>
<td>2.2.3 Application-level latency hiding techniques</td>
<td>37</td>
</tr>
<tr>
<td>2.2.4 End-to-end argument</td>
<td>38</td>
</tr>
<tr>
<td>2.2.5 Exokernels</td>
<td>39</td>
</tr>
<tr>
<td>2.2.6 Resource Kernels</td>
<td>41</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>2.2.7</td>
<td>QoS-based resource management</td>
</tr>
<tr>
<td>2.3</td>
<td>Performance analysis of computer systems</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Queueing networks</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Stochastic Petri Nets</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Hierarchical stochastic models</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Deterministic models</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Simulation</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Performance measurements</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary and conclusions</td>
</tr>
<tr>
<td>III.</td>
<td>APPLICATION AND SYSTEM LEVEL METRICS</td>
</tr>
<tr>
<td>3.1</td>
<td>Characterization of the target application domain</td>
</tr>
<tr>
<td>3.2</td>
<td>Analysis of application execution time</td>
</tr>
<tr>
<td>3.3</td>
<td>Extension of the notion of the protocol stack offload</td>
</tr>
<tr>
<td>3.4</td>
<td>Application and system metrics</td>
</tr>
<tr>
<td>3.5</td>
<td>Use of the metrics for studying CPSO performance effects</td>
</tr>
<tr>
<td>3.6</td>
<td>Summary</td>
</tr>
<tr>
<td>IV.</td>
<td>RESEARCH METHODOLOGY</td>
</tr>
<tr>
<td>4.1</td>
<td>The thesis validation procedure</td>
</tr>
<tr>
<td>4.2</td>
<td>The CPSO scenarios of interest</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Checksum offload</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Data copy avoidance</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Large MTU size</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Complete CPSO</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Complete CPSO with data copy avoidance</td>
</tr>
<tr>
<td>4.3</td>
<td>Models for CPSO performance effects</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary</td>
</tr>
<tr>
<td>V.</td>
<td>EXPERIMENTAL WORK</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.2</td>
<td>The experimental platform description</td>
</tr>
<tr>
<td>5.3</td>
<td>The experimental work on the dissertation thesis validation</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Benchmark description</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Experimental results</td>
</tr>
<tr>
<td>5.3.3</td>
<td>The comparison of the calculated and measured values of $S_{comm}$</td>
</tr>
<tr>
<td>5.4</td>
<td>The experimental work on characterization of the networking subsystem</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>DoBench pseudo-code</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>$D_o$ measured in DoBench</td>
<td>91</td>
</tr>
<tr>
<td>5.3</td>
<td>$D_{accs}$ measured in DoBench</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>$D_o$ standard deviation</td>
<td>93</td>
</tr>
<tr>
<td>5.5</td>
<td>$D_{accs}$ standard deviation</td>
<td>94</td>
</tr>
<tr>
<td>5.6</td>
<td>$S_{comm}$ measured in DoBench</td>
<td>95</td>
</tr>
<tr>
<td>5.7</td>
<td>$S_{comm}$ delta</td>
<td>95</td>
</tr>
<tr>
<td>5.8</td>
<td>COV $S_{comm}$</td>
<td>96</td>
</tr>
<tr>
<td>5.9</td>
<td>Memory copy latency</td>
<td>101</td>
</tr>
<tr>
<td>5.10</td>
<td>Checksum latency</td>
<td>101</td>
</tr>
<tr>
<td>5.11</td>
<td>MpiBench pseudo-code</td>
<td>104</td>
</tr>
<tr>
<td>5.12</td>
<td>MpiBench CPU utilization</td>
<td>111</td>
</tr>
<tr>
<td>5.13</td>
<td>MpiBench NIC interrupt rates</td>
<td>111</td>
</tr>
<tr>
<td>5.14</td>
<td>MpiBench communication bandwidth</td>
<td>112</td>
</tr>
<tr>
<td>5.15</td>
<td>MpiBench communication latency</td>
<td>112</td>
</tr>
<tr>
<td>5.16</td>
<td>$D_{accs}$ measured in MpiBench</td>
<td>113</td>
</tr>
<tr>
<td>5.17</td>
<td>$D_{accs}$ delta</td>
<td>114</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>6.1</td>
<td>$D_o$ for the Open QN model</td>
<td>127</td>
</tr>
<tr>
<td>6.2</td>
<td>$D_o$ for the Open QN model, continued</td>
<td>128</td>
</tr>
<tr>
<td>6.3</td>
<td>$R$ for the Open QN model</td>
<td>129</td>
</tr>
<tr>
<td>6.4</td>
<td>$D_{acs}$ for the Open QN model</td>
<td>130</td>
</tr>
<tr>
<td>6.5</td>
<td>$S_{comm}$ for the Open QN model</td>
<td>130</td>
</tr>
<tr>
<td>6.6</td>
<td>The MVA algorithm pseudo code</td>
<td>133</td>
</tr>
<tr>
<td>6.7</td>
<td>$R$ for the Closed QN model</td>
<td>133</td>
</tr>
<tr>
<td>6.8</td>
<td>$D_o$ for the Closed QN model</td>
<td>134</td>
</tr>
<tr>
<td>6.9</td>
<td>$D_o$ for the Closed QN model, continued</td>
<td>135</td>
</tr>
<tr>
<td>6.10</td>
<td>$S_{comm}$ for the Closed QN model</td>
<td>136</td>
</tr>
<tr>
<td>6.11</td>
<td>$R$ for the Deterministic model</td>
<td>137</td>
</tr>
<tr>
<td>6.12</td>
<td>$D_o$ for the Deterministic model</td>
<td>138</td>
</tr>
<tr>
<td>6.13</td>
<td>$D_o$ for the Deterministic model, continued</td>
<td>139</td>
</tr>
<tr>
<td>6.14</td>
<td>$D_{acs}$ for the Deterministic model</td>
<td>140</td>
</tr>
<tr>
<td>6.15</td>
<td>$S_{comm}$ for the Deterministic model</td>
<td>140</td>
</tr>
<tr>
<td>6.16</td>
<td>Delta $D_o$ for the adjusted Open QN model</td>
<td>142</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

This dissertation addresses a research topic in analysis of the performance effects resulting from optimizations of networking subsystems in clusters of workstations. The term "networking subsystem" refers to a collection of hardware and software components that provide messaging services to applications. Specifically, networking subsystem includes communication protocol stack and communication middleware software components.

The dissertation focuses on a specific class of networking subsystem optimizations commonly referred to as "communication protocol stack offload" (CPSO). This class of networking subsystem optimizations is represented by a set of system design and implementation techniques that allow one to reduce the volume of the stack processing that is traditionally performed by the node compute engine. A fraction of this processing is offloaded and processed by the node’s networking subsystem.

The primary objective of the research is to advance the understanding of the impact of the protocol stack offload on application performance. Specifically, it is desirable to develop a method that allows one to quantify the effect of the CPSO phenomenon on the application execution time.
The system configuration that exhibits the CPSO phenomenon is characterized by a certain breakdown of the communication protocol processing workload between the cluster node CPU and the network co-processor. This breakdown is referred to as the protocol stack offload scenario in the rest of the dissertation.

An important related application characteristic that modulates the performance effects of the CPSO scenarios is the application’s capacity to hide (tolerate) communication latency by overlapping it with application-specific computation. This capacity determines the degree to which the application can take advantage of the performance gains offered by the CPSO scenarios.

In order to achieve these research objectives, a set of metrics is proposed and used to quantify the performance effects of CPSO scenarios. These metrics are derived from experimental data for a set of CPSO scenarios of interest. It is shown that the metrics adequately describe the performance effects of the CPSO scenarios. The metrics can be used in order to identify the scenario that minimizes the application execution time for a class of applications characterized by a certain latency hiding capacity.

Additionally, the CPSO scenarios are studied with the use of several analytical models. The purpose of this study is to evaluate the predictive power and the precision of the models. The results of this study are used to determine the degree of applicability of these models to the analysis of the CPSO performance effects.

Although the research methodology presented in this dissertation is applicable in a wider setting, the research focuses on a concrete class of computer systems. The computer...
system under consideration is a PC-based workstation commonly used as a compute node in a cluster. The communication protocol stack considered for offload is the ubiquitous MPI/TCP/IP stack also commonly used in clusters as a general data transport. The subject application domain is parallel scientific applications. These choices are designed to make the results of the proposed research immediately relevant in the context of parallel scientific applications that present an important and growing class of application workloads for clusters.

The subsequent sections of this chapter provide an overview of the proposed research. Section 1.1 reviews the relevant concepts that provide proper context for the discussion that follows. Section 1.2 discusses the problem area addressed by the proposed research, and Section 1.3 formulates the dissertation thesis. Section 1.4 provides necessary justifications for the proposed research. Section 1.5 defines the scope of the research, and Section 1.6 briefly reviews the research methodology. Finally, Section 1.7 describes the organization of the rest of the dissertation.

1.1 Background

Dynamics of developments in computer-related technologies and economics of scale have brought clusters of PC-based workstations, also known as Networks of Workstations (NOWs), into the spotlight during the past decade [72]. Increased processing power of COTS CPUs, as well as improved designs of memory subsystems, allowed for significant increases in the computational power of cluster nodes. Advances in switched networking
technologies [14, 16, 23, 40, 41, 57, 71, 73, 103] and I/O subsystem design resulted in cluster networking solutions with scalable bandwidth and low latency previously available only in MPP systems. Emergence of freely available Linux OS as a viable alternative to proprietary operating systems offered opportunities for customization of the operating system environment, as well as for easy incorporation of kernel support for new features not available with proprietary operating systems.

Increased computational power and improved scalability, as well as availability of free OS environment and development tools, transformed clusters into attractive platforms for a wide range of applications such as high-performance transaction processing, streaming video, real-time image processing in entertainment and medical imaging, and noise reduction and increased channel selectivity in 3G wireless applications. The distributed nature of clusters comprising multiple similar computing nodes matched well the scalability and high availability demands of many important applications for clusters. Similarly, growing computational power of cluster nodes, combined with increased network performance, made clusters attractive platforms for many applications of scientific computing [4, 6, 72].

1.1.1 System under study

These observations have motivated the choice of a typical PC-based cluster node as an interesting and important system under study. The system architecture of a PC-based node is standardized, and its hardware and software components are well-defined and ubiquitous. The cluster node CPU and host memory subsystem, including memory and cache,
memory interconnect, and the related bridges and controllers form a tightly coupled sub-
system that is collectively referred to as cluster node compute engine. The subsystem is
responsible for the processing of the application-specific computation and the processing
of system-level tasks, including the communication protocol stack processing.

The I/O system interconnect and memory-to-I/O-interconnect bridge is another logi-
cally cohesive set of components collectively referred to as cluster node I/O fabric. It's
primary role is to provide scalable bandwidth and low latency communication between the
system compute engine and I/O devices.

The network co-processor is one such I/O device. The device performs data transfer
requests issued to it by the system compute engine. Some network co-processors a.k.a. in-
telligent NICs (iNICs) are capable of offloading a fraction of the communication protocol
stack processing from the system compute engine. Such NICs are referred to as protocol
stack offload engines.

Although the complexity of the data transfer requests issued by the system compute
engine to the NIC varies depending on the NIC’s “intelligence,” all NICs perform some
non-zero portion of the communication processing. Therefore, in this dissertation, the
notion of the protocol stack offload engine includes NICs of all levels of intelligence,
as a matter of notational convenience. Finally, the network physical media and network
switches form yet another cohesive subset of system components collectively referred to
as the cluster network fabric.
The hardware system components discussed above do not directly interact with application programs. They are abstracted and virtualized to a varying degree by the operating system, communication protocol stacks and middleware. Since the dissertation studies application performance as it is affected by the CPSO scenarios, it is also necessary to introduce the terminology for the system components in direct interaction with the applications.

The cluster node *compute subsystem* is a term that refers to the system compute engine virtualized by the cluster node operating system. The cluster node *networking subsystem* is a complementary term that refers to the CPSO engine, system I/O fabric, and cluster network fabric virtualized by the protocol stack and the communications middleware.

### 1.1.2 Performance metrics and workload definitions

In order to analyse the performance effects of CPSO scenarios, it is necessary to define the performance metrics and the workloads under consideration. The primary application performance metric used in this dissertation is elapsed application execution time $T$ measured in seconds. The CPSO scenarios are evaluated and quantified as to their effect on the application execution time.

The system components discussed in Section 1.1.1 process the application-level and system-level tasks that are in turn driven by the application to a significant degree. The system-level tasks of specific interest in this dissertation are the communication protocol stack processing. These tasks and their corresponding workloads are described in terms of
their processing demands $D_i$ on the system components discussed earlier. The processing demands reflect the time measured in seconds that a component needs in order to fulfill a given request. This approach to the workload description is used in Queueing models discussed further in Chapter II.

In order to analyze the CPSO performance effects, it is necessary to distinguish between the application-specific portion of the workload and the communication stack processing workload. This task is simplified by the fact that this dissertation considers a class of parallel scientific applications characterized by two major activities: application-specific computation and communication. The former activity constitutes the application-specific workload. The latter activity drives the node’s networking subsystem, and therefore defines the communication stack processing workload. The latter workload is studied in detail, both empirically and in the course of analytical modeling discussed in this dissertation.

The detailed study of the application-specific computational workload is outside the scope of this dissertation. The only metric that describes this workload is the total application-specific computation time $T_{comp}$ measured in seconds. Additionally, certain aspects of the application need to be considered since they determine to a significant degree the profile of the stack processing workload.

The following two metrics are used to describe these aspects: the total time $T_{comm}$ spent by the application while performing communication and the ratio $R = \frac{T_{comp}}{T_{comm}}$ that characterizes the relative amount of computation and communication performed by the
application. As discussed in the subsequent sections of the dissertation, the latter aspect is important in determining the degree to which the application can take advantage of the concurrency of the communication and computation processing under the conditions of the protocol stack offload. As was shown in [26], many parallel applications can be designed and implemented in a way that allows them to take advantage of the asynchronous communication engines and to effectively hide communication latency by overlapping communication and computation. This latency hiding capacity is mainly determined by the application algorithm and application implementation techniques.

1.2 Problem statement

The growing gap between raw CPU processing capacity, network link speeds and memory subsystem bandwidth has been observed over a number of years [56, 83]. CPU processing power has been growing at an average rate of 50% a year during 1985-2001, while network throughput grew 40% a year, and sustained memory bandwidth grew 35% a year on average during 1975-2001 [83]. Such disproportionate growth of CPU and network speeds over the sustained memory bandwidth invalidated many tradeoffs made in the design and implementation of operating systems and communication protocol stacks most of which were developed two decades ago [42]. The assumption that network node’s memory bandwidth was significantly larger than network link speed, and that it was acceptable to perform significant per-byte processing was no longer valid. For this reason, the legacy OS support and processing of the communication protocol stacks consumed an
increasingly excessive amount of the node’s computational resources in order to utilize the network at high speeds.

Various approaches to alleviating this I/O bottleneck [21] have been developed. One of the approaches proposed to offload the communication protocol stack processing to a specialized intelligent protocol offload engine. The notion of the stack offload implied that communication would happen asynchronously with respect to the computation performed by the host compute engine. Therefore, the application performance gains could come from freeing up a certain percentage of the node’s compute resources and overlapping the computation with asynchronous communication performed by the offload engine.

Despite the significant engineering and research efforts in the area, there have not been many attempts to undertake a systematic study of the protocol stack offload phenomenon. To the best of the author’s knowledge, no general methods and metrics have been proposed for quantification of the CPSO performance effects, no studies that use such methods have been conducted. No systematic evaluation of analytical models as to their applicability to the performance analysis of the CPSO scenarios has been conducted either. The research effort discussed in this dissertation addresses this apparent gap.

1.3 Dissertation thesis statement

The following statement formulates the dissertation thesis. The effects of communication protocol stack offload (CPSO) on application execution time can be attributed to the following two complementary sources. First, the application-specific computation may
be executed concurrently with the asynchronous communication performed by the communication protocol stack offload engine. Second, the protocol stack processing can be accelerated or decelerated by the offload engine. These two types of performance effects can be quantified with the use of the degree of overlapping \( D_o \) and degree of acceleration \( D_{accs} \) metrics. The composite communication speedup metric \( S_{comm}(D_o, D_{accs}) \) can be used in order to quantify the combined effects of the protocol stack offload.

This dissertation thesis is validated empirically. The degree of overlapping \( D_o \), the degree of acceleration \( D_{accs} \), and the communication speedup \( S_{comm} \) characteristic of the system configurations under test are derived in the course of experiments performed for the system configurations of interest. It is shown that the proposed metrics adequately describe the effects of the protocol stack offload on the application execution time. A detailed discussion of the research methodology can be found in Chapter IV.

In addition to the experimental work conducted in order to validate the dissertation thesis, a set of analytical models of the networking subsystem of a PC-based cluster node is developed. As a result of the modeling, the metrics \( D_o, D_{accs}, \) and \( S_{comm} \) are obtained. The models are evaluated as to their complexity and precision by comparing the modeling results with the measured values of \( D_o, D_{accs}, \) and \( S_{comm} \).

The primary contributions of this dissertation research are as follows. First, the metric \( D_{accs} \) and \( S_{comm} \) are introduced in order to complement the \( D_o \) metric in its use for evaluation of the effects of optimizations in the networking subsystem on parallel performance in clusters. The metrics are shown to adequately describe CPSO performance
effects. Second, a method for assessing performance effects of CPSO scenarios on application performance is developed and discussed. Third, a set of analytical models of cluster node networking subsystems with CPSO capability is developed and characterised as to their complexity and precision of the prediction of the $D_o$ and $D_{accs}$ metrics.

1.4 Justification of study

In the course of the dissertation research, the dissertation thesis is validated. As a result, a method for evaluating performance effects of the CPSO scenarios is developed and validated. This method can be used as a valuable tool for characterization of communication performance in clusters. The method can also be used to evaluate the performance benefits of CPSO approaches that can be utilized by applications characterized by specific design and implementation strategies.

The developed analytical models of the networking subsystem are evaluated, and the conclusions as to their applicability in the performance analysis of the CPSO effects are made. These models can be used within the identified limitations when the empirical evaluation is not feasible.

The proposed research considers the system architecture of a modern PC-based workstation that is commonly used in cluster computer systems designed to perform parallel scientific computations. The ubiquity of such compute engines warrants the relevance of the research in the broader context of parallel applications for clusters of workstations.
The role of clusters as increasingly powerful computing platforms for a growing range of parallel and distributed applications has been widely recognized [72]. The superior price-performance characteristics of clusters and their increasing potential continue to attract significant efforts in application development and research. Given the significant effect of the characteristics of the cluster network subsystem on the application performance, research in performance analysis of cluster networking subsystem presents an important area of research in computer science.

1.5 Scope of the dissertation research

The scope of the proposed research is limited to the performance analysis of the cluster node networking subsystems under conditions of the protocol stack offload as discussed earlier. This work does not include a detailed application performance analysis under conditions of overlap of computation and communication. The existing research results discussed in [26] in this area can be used for that purpose.

Thorough study of aspects of the communication protocol stack offload phenomenon other than performance implications, such as system scalability and adaptability, is also outside the scope of this research. These and other related areas present interesting opportunities for future research.
1.6 Dissertation research methodology

As mentioned above, the primary research objective is to adequately describe the CPSO scenarios of interest with the use of the metrics $D_o$, $D_{accs}$, and $S_{comm}$, as well as to demonstrate the application of the metrics in the course of the comparison of their performance effects. The adequate description of the CPSO performance effects is instrumental in achieving a reasonable understanding of the performance implications of the protocol stack offload for a given class of parallel applications characterized by their ability to hide communication latency.

The primary research objective is achieved as follows. The procedures for measuring the metrics are defined, and the appropriate experiments are designed. In the course of these experiments, $D_o$ and $D_{accs}$ are measured, and the measurements’ precision is characterized. Then, the communication speedup $S_{comm}$ is calculated and compared to the communication speedup that is measured directly. Based on the agreement of the measurements and calculations, a conclusion is made as to the value of the metrics in quantifying the CPSO performance effects.

An additional research goal is to evaluate a set of models in their ability to predict the metrics for the CPSO scenarios of interest. Three models based on Queueing Network (QN) system representation are developed and studied. In this study, the metrics $D_o$, $D_{accs}$, and $S_{comm}$ are derived in the course of modeling. The derived metrics are compared with the values obtained during the experimental thesis validation. Based on this comparison, the conclusions are made as to the models’ precision and their limitations.
The approaches to improving the models are proposed. A detailed investigation of these approaches may present interesting future research opportunities.

1.7 Organization of the dissertation

The dissertation text is organized as follows. Chapter II presents a review of relevant research materials published to date. A review of research on quantification of TCP/IP protocol stack overhead is followed by the discussion of approaches to reducing the protocol stack overhead, including end-system optimizations, communication protocol stacks optimizations, and communication protocol stack offload.

These efforts are placed in the broader context of the relevant more general research that is focused on increasing performance of networking subsystems. Several approaches are reviewed, including revision of the hardware/software system interfaces, minimization of the overhead of runtime environments, and application-level latency hiding. Several general system design and architectural ideas such as end-to-end argument, exokernels, resource kernels, and QoS-based resource management, are also evaluated as to their applicability to the problem of increasing networking subsystems’ performance.

Chapter II also reviews modern approaches to computer system performance analysis. Analytical modeling frameworks such as Queueing Networks, and Stochastic Petri nets, hierarchical stochastic models, and deterministic models are reviewed. Simulation and performance measurements techniques are also discussed. The chapter concludes with a
brief summary of relevant results and the justification of the choice of QN-based models as the analytical framework for the modeling process.

Chapter III presents the terms and definitions used in the rest of the dissertation. Chapter IV discusses the proposed research methodology. Chapter V presents the experimental work on the thesis validation. Chapter VI discusses the analytical models of the cluster node networking subsystem. Finally, Chapter VII summarizes the dissertation research and Chapter VIII discusses directions for future work motivated by the research effort described in this dissertation.
CHAPTER II

PERFORMANCE ANALYSIS OF NETWORKING SUBSYSTEMS

This chapter presents a review of relevant research materials published to date. The techniques for reducing the communication protocol processing burden on the cluster node compute subsystem, including communication protocol stack offload, are discussed. These techniques are placed in the broader context of the relevant more general research focused on increasing performance of computer systems. Recent revision of semantics of hardware/software interfaces and several interesting approaches to minimization of the overhead of runtime environments, with implications on performance analysis, are reviewed.

The chapter then offers a review of several performance analysis techniques: analytical modeling techniques represented by Queueing Networks, Petri Nets, hierarchical stochastic models, and deterministic models, as well as simulation modeling techniques, and performance measurements. The chapter discusses the choice of QN-based models for performance analysis of cluster networking subsystems. The chapter concludes with a summary of the reviewed material.
2.1 Approaches to reducing TCP/IP stack overhead

The distributed nature of cluster system architectures emphasizes the need for performance and scalability of cluster data transfer services that rely on scalable and high-performance network architecture. The need for high-performance network fabrics with scalable bi-section bandwidth has been largely addressed by increased network link signaling rates and advances in packet-switched network technologies [16, 23]. However, the ability to utilize the increasingly powerful networks has come at an increasingly high price in terms of the end system compute resources\(^1\) dedicated to network-related processing.

The growing gap between raw CPU processing capacity, network link speeds, and memory subsystem bandwidth has been observed over a number of years [56, 83]. CPU processing power has been growing 50% a year on average during 1985–2001, while network throughput has been growing 40% a year, and sustained memory bandwidth has been growing 35% a year on average during 1975–2001 [83].

This disproportionate growth of CPU and network speeds over the sustained memory bandwidth has invalidated many tradeoffs made in the design and implementation of I/O system interfaces and communication protocol stacks most of which have been developed two decades ago [17]. Specifically, the assumptions that the end-system memory bandwidth was significantly higher than network link speed, and that it was acceptable to perform significant per-byte processing, were no longer valid. For this reason, the legacy

\(^1\)The compute resources in question include both CPU time and memory bandwidth consumed by the network stack processing.
OS support and processing of the communication protocol stacks has consumed increasing amounts of the end node’s computational resources in order to utilize the network at high speeds.

Several approaches to alleviating this I/O bottleneck [21] have been developed [17, 20, 27, 99]. These approaches can be classified as end-system optimizations, communication protocol optimizations, and communication protocol stack offload.

### 2.1.1 Quantifying the TCP/IP protocol stack overhead

The ubiquitous TCP/IP stack has received significant attention due to its generality and the amount of legacy software developed for use in systems with TCP/IP. It was determined that the primary sources of the overhead in TCP/IP-based networking subsystems could be attributes to the legacy I/O system interfaces, operating system overheads, interrupt processing, multiple data copies, and checksum computation.

Clark et al [20] conducted a detailed study of the sources of the TCP/IP processing overheads. The researchers drew a clear distinction between the intrinsic protocol processing and other source of the overhead such as the operating system support for timers, interrupt processing, I/O buffer management, and thread scheduling, as well as multiple data copies and checksum computations. Their experiments were focused on a common case scenario of the half-duplex inter-process bulk data transfers.

The reported instruction counts on both send and receive sides turned out to be dominated by the above-mentioned overheads attributed to the operating system environment,
as well as to data touching operations such as the above-mentioned data copies and checksum computations. In their measurements, the per-byte overheads (data copies and checksums) constituted 64% of the total CPU overhead, as compared to the 36% of per-packet overhead that included 20% operating-system-related overhead, 8% TCP/IP processing overhead, and 8% Ethernet adapter driver overhead for 1,460 byte Ethernet packets.

The main conclusion drawn from the experiments was that TCP/IP processing itself did not constitute a bottleneck, and it could support very high transfer rates if properly implemented. In order to address the identified sources of the overhead, the authors proposed to optimize the relevant operating system facilities, to minimize the number of data copies, and to combine checksum computation with data copies.

Clark et al [20] also considered moving the protocol processing from the host compute engine outboard, onto a specialized controller. According to their views, the controller could run a lean operating system with optimized TCP/IP stack. The controller would need to have high-performance memory subsystem and optimized I/O bus interface.

Another alternative considered in [20] was a relatively simple network controller with interrupt batching capabilities that could be synchronously driven by the host CPU. In this case, the host CPU could copy data directly from the user buffers to the controller, while performing checksumming in the same loop.

The authors also stated their skepticism regarding the hardware-based implementations of TCP/IP stack. In their opinion, inflexibility of the hardware-based solutions could not
be justified by the performance increases because the major sources of the overhead lay outside the protocol stack processing.

2.1.2 End-system optimizations

A number of other researchers published similar findings as to the sources of TCP/IP stack processing overhead [19, 94, 99]. Chase et al. [19] reported on the analysis of the system resource utilization by applications using TCP/IP for high-throughput communication. The authors showed that modern TCP implementations were capable of very high throughput, and the achievable networking performance was limited by the end systems. The causes of limited performance were per-packet operating-system-related and interrupt processing overheads, as well as per-byte overheads for data buffering and checksum calculation, rather than the network hardware or the TCP protocol processing itself. The measurements conducted in [19] showed that increasing MTU size to 8 Kbytes, avoiding data copies and offloading checksum processing, resulted in bandwidth improvements up to 70%, relative to the standard 1500 bytes Ethernet MTU size. The corresponding reduction in CPU utilization was from 95% to 45% for 8 Kbytes MTU and sustained network throughput of about 1 Gbit/sec [19].

The approaches to reducing per-packet processing overheads discussed in [19, 41, 57] included network device driver optimizations for batching interrupt processing and network buffer management and MTU size increase. Increased MTU size lead to reduction of the total packet count and also contributed to reduction of the interrupt processing
overhead. These approaches complimented the techniques for reduction of the per-byte end-system processing overhead.

An interesting review of the techniques for reduction of the per-byte processing overheads was presented in [17]. The authors discussed the effects of data buffering performed by the OS on performance of the system I/O services at the application level. The paper showed that system-level data buffering resulted in additional data copies and incurred significant overheads that adversely affected the system I/O performance. The major source of these adverse effects in the case of network I/O was the disparity between rapidly growing CPU and network throughput on one hand, and the slower growing host memory throughput on the other hand.

The paper offered a novel taxonomy of hardware and software approaches to decreasing these overheads while preserving the copy semantics of the UNIX-style I/O system interfaces. The novel optimization techniques of copy avoidance, input-disabled pageout, transient output copy-on-write, and input alignment, were discussed. These techniques were used in an optimized buffering approach called emulated copy.

The authors noted that the degree to which the emulated copy techniques were applicable depended on the sophistication of the subject I/O controllers implementing early-multiplexing, pooled, or outboard buffering semantics. Overall, the emulated copy technique had higher latency for small transfer sizes, but higher throughput (average 50% increase) and smaller CPU utilization (from 26% to 10–12%) for all message sizes. The
authors noted that this did not constitute a problem since emulated copy and traditional copy could be used in combination in their respective optimal ranges of the transfer sizes.

The work presented in this paper generalized earlier results obtained for specific copy avoidance techniques and specific devices [39, 94]. The related Fbufs work [27] used a similar approach with variations in the user buffer virtual page attributes that resulted in slightly different implications of access to the application-level data buffers when the transfer was in progress.

The main conclusion made in the paper was that the presented emulated copy technique could be used to re-implement UNIX-style I/O system interfaces in application-transparent fashion while achieving significant I/O performance increases.

2.1.3 Communication protocol stack optimizations

One of the issues with TCP/IP protocol stack that is often referred to when discussing TCP/IP performance is its generality. Since TCP/IP protocol stack is designed to be ubiquitous, it cannot not take advantage of any assumptions about the characteristics of the underlying media and data link protocols in order to optimize its performance.

In contrast, the characteristics of many physical and data link networking technologies commonly used in cluster environments include high link speeds, very low bit error rates, and provisions for hardware flow control. Therefore, cluster networks allow for many simplifying assumptions in the design and implementation of transport protocols [16, 23, 71]. Many light-weight protocols designed specifically for cluster environments take advantage
of these cluster networks characteristics in order to reduce their processing overheads and increase application-level performance [73, 88].

Some lightweight communication protocols are further optimized based on a limited set of abstractions they provide to the upper-layer software. In [88], port-multiplexed connection-based byte stream abstraction implemented in TCP is replaced by connectionless reliable packet delivery with no multiplexing in the name of efficiency and low overhead. This and other similar protocols can be executed entirely by NICs, and they allow for further reduction of CPU processing overhead and elimination of the OS from performance-critical paths.

Another interesting example of transport protocol optimization is software-assisted modular implementation of the VIA specification [65]. This implementation is designed to accommodate networking hardware with little or no support for VIA-specific features, such as many Ethernet adapters. In order to support RDMA operations, M-VIA implements a lightweight reliable data transport on top of Ethernet data link functionality. Additionally, M-VIA takes advantage of the VIA memory registration and receive buffer pre-posting semantics in order to avoid data copying overheads. VIA-style control over interrupt generation upon completion of data transfer requests allows for tight control over the CPU overhead.

Other examples of standards-based technologies that have a very light transport protocol stack include VIA and InfiniBand [16, 23]. These technologies are reviewed in section 2.2.1, as they present an interesting example of high-level communication semantics di-
rectly supported in the hardware. The two common high-throughput data communication primitives provided by both VIA and InfiniBand are RDMA read and write primitives. In effect, they allow network-connected computer systems to access each other’s memory in a protected fashion without explicit application-level participation of the remote party. As a result, communication processing overhead is reduced. Many application-level standards such as SDP and DAT [22, 91], take advantage of these primitives.

In order to deliver high-performance data transport services, both VIA and InfiniBand rely on sophisticated specialized hardware support. A more general solution that provides high-performance RDMA communication primitives is described in the iWARP specification suit [24, 82, 86]. The specification suit takes advantage of the existing reliable underlying transport service such as TCP and SCTP [42, 95]. The specification consists of three standards, RDMAP, DDP, and MPA [24, 82, 86]. The explicit goal of the specification is to provide a general platform-independent solution for efficient data communications facilities capable of placing data into the sink buffers directly (with no intermediate copies), and capable of limiting the host CPU involvement in the communication-related processing to a minimum.

The RDMA Protocol (RDMAP) [82] defines iWARP data transfer operations, including RDMA, as well as their completion semantics, ordering guarantees, synchronization, and interaction with data buffers. RDMAP also defines the message formats for the data transfer operations, as well as underlying transport stream setup and teardown, control message formats, and message exchange protocol.
Direct Data Placement (DDP) specification [86] defines data placement operations and their semantics, including the tagged (for RDMA) and untagged (for regular send) buffer models, data segmentation and reassembly, and DDP message ordering, completion and delivery semantics. Additionally, DDP defines its involvement in the underlying transport data stream teardown, error semantics, and message formats.

Marker PDU-Aligned (MPA) Framing specification [24] describes record framing markers to be used in cases when the underlying transport protocol does not support upper-layer protocol record framing (which is the case for TCP/IP). These markers are useful in conveying framing information to the network interface adapters for more efficient data processing with no intermediate copies and low CPU utilization.

Given the explicit goals of the iWARP specification to reduce communications demands for host memory bandwidth and host CPU utilization, iWARP clearly benefits from the transport stack offload. The protocol stack offloading allows for reduction the processing overhead of the transport layer utilized by iWARP and therefore compliments the benefits offered by iWARP in this respect.

2.1.4 Communication protocol stack offload

One of the proposed ways to alleviate the protocol stack burden on the host compute engine was to offload the protocol stack processing to a specialized intelligent protocol offload engine. This approach gained popularity as NICs were becoming increasingly more sophisticated in response to growing demand for higher communications performance.
Today, virtually all NICs include DMA engines for high-bandwidth data transfers between host and NIC memories, as well as to and from the network fabric. It is also common to find ASIC- and FPGA-based NIC support for offloading CPU-intensive parts of protocol processing such as checksum calculation and segmentation/re-assembly of transport-level packets [1, 3, 41, 52, 57]. Modern NIC drivers are capable of informing higher layers of the OS protocol stack of the NIC capabilities. The drivers also provide for various performance optimizations such as receive/transmit interrupt batching, distribution of network load among several NICs [40, 92], and high availability features based on redundancy of NICs and fabric connections [40]. As a consequence, the host CPU is freed to perform more application tasks, and host memory and I/O subsystems are often less burdened by network-related traffic.

Many modern NICs also include CPUs with instruction/data caches that execute firmware designed to schedule and control cooperating tasks performed by other NIC components. In effect, NICs become specialized embedded systems designated for high-performance communication protocol processing [14, 73, 88, 103]. As compared to ASIC- and FPGA-based solutions, these NICs are more flexible and customizable to specific customer requirements such as assisting the host CPU in processing multi-protocol stacks. These and other NICs with advanced support for transport protocol processing mentioned above are often referred to as *smart or intelligent NICs* (iNICs).

One of the factors that limits flexibility of protocol processing task assignment to iNICs is the design of the existing communication protocol stacks in commodity OSs. In this
respect, Linux OS imposes the least amount of such limitations because the protocol stack code is freely available for customization. However, even in the case of Linux, protocol stack design (as it is related to the overall OS design) imposes certain granularity of the task offload to iNICs. The \textit{I}_2\textit{O} Architectural Specification [38] adds flexibility in assigning protocol stack tasks to the host CPU and I/O processors (iNIC in case of networking). Many OSs, including Linux, support the \textit{I}_2\textit{O} architectural model.

The opportunities of offloading communication protocol stack processing has received significant attention in the context of the Network Attached Storage (NAS) technologies. TCP/IP offloading allows one to bypass the host compute engine and to limit the iSCSI (SCSI over IP) [85] and FCIP (Fiber Channel over IP) [77] traffic processing to iNICs and intelligent disk controllers that communicate over the I/O interconnect directly. The resulting performance gains have attracted significant interest in developing high-performance TCP/IP offload solutions [1, 3, 29, 52, 101].

Another approach to TCP offload applicable in System area Network (SAN) cluster environment was discussed in [10, 80, 81, 87]. The central idea of the approach was to move all processing of TCP/IP communication requests to a dedicated cluster node referred to as the TCP server node. The MemNet developers successfully used this approach to increase the throughput of a SAN-based Web server up to 30% [80]. They replaced the native TCP/IP socket implementation on the cluster nodes that ran the Web server processes with RPC requests to the TCP server. The RPC communication was performed over VIA-based SAN internal to the cluster.
The MemNet developers evaluated several TCP server design alternatives and several synchronous and asynchronous alternatives of the communication interfaces for their RPC implementation. They found that the most efficient TCP server implementation option was in-kernel VIA endpoint approach that avoided user-kernel data copies. They also found that asynchronous RPC offered superior performance as compared to both synchronous RPC and the regular TCP/IP option.

Several related research efforts evaluated design and implementation tradeoffs in SAN-based systems with TCP offload [10, 87]. The Communication Services Platform (CSP) project [87] also took the approach of moving TCP processing to dedicated network nodes and performing TCP service requests to theses nodes over a VIA-based cluster interconnect. Similar to the MemNet project, CSP focused on evaluation of the system design and implementation tradeoffs with the goal of optimizing throughput of a specialized application (SAN-based Web server) by means of offloading TCP processing to a dedicated cluster node. The TCP offload strategy was decided \textit{a priori}, and different offload options were not evaluated.

An interesting work with the goal of analyzing the impact of high-performance switched I/O interconnects in systems with active devices was presented in [18]. The system under study was an SMP-based high-performance web server with active (intelligent) disk and network devices capable of offloading file system and TCP stack processing. PCI, RapidIO, and InfiniBand were considered as the system I/O interconnect.
The authors proposed a pipelined web server model and, based on the timing characteristics of the system calculated using the model, they derived the web server throughput. They used this metric for comparison of the system design options with conventional (not active) devices and the system design options with active devices of three different processing speeds (relative to the host CPUs).

In order to obtain the processing demands the web server application imposed on the system components, the authors conducted a series of simulations on a MINT-based simulator [102]. They implemented the file system and TCP libraries and simulated their execution of the host CPUs as well as on the active devices. The authors validated their analytical model by comparing its predictions with the results of the corresponding simulation runs. They found a good qualitative agreement, but their quantitative results were between 9% and 22% apart. The authors attributed this disparity to the limitations of their simulation framework.

The main conclusions reached by Carrera et al in [18] was that the processing power of the active devices and the throughput of the I/O interconnect determined the application performance gains as compared to the case of the conventional system. The full potential of systems with active devices could be revealed if system software architectures allowed for offload of file I/O and TCP processing to intelligent devices, and high-performance switched system I/O interconnects provided for sufficient I/O bandwidth.

In this work, the authors used their models in order to estimate application performance gains for various system configurations based on an a priori idea of the file system and
TCP stack offload to the intelligent devices. The authors did not attempt to investigate the effects of different offload strategies on application performance. Therefore, this work is complimentary to the work discussed in this dissertation.

The work described in [10, 18, 80] was extended in [81]. In addition to evaluating the RPC-based TCP service request delegation over VIA-base SAN, the researchers used shared memory communication in an SMP system where they offloaded TCP processing to a dedicated CPU. Their main conclusion was that shared memory communication also allowed them to increase the Web server throughput up to 30%, and in addition to the performance advantages they have obtained, there was a clear opportunity to further optimize their system by dynamically balancing processing load between the active devices and the host CPU.

Based on the review of the research efforts in communication stack offload presented in this section, one can make several conclusions. The performance benefits of the communication stack offload are widely recognized, and many efforts in industry and academia are focused on evaluation of the design and implementation tradeoffs for the protocol stack offload solutions in specific application domains. The work on performance analysis in SAN-based and SMP-based distributed systems with TCP/IP offload in one such domain (distributed web server processing) focused on assessing potential benefits of using different system I/O interconnect options and intra-system transport implementation strategies in order to increase efficiency and performance of communication with the active devices that implement the protocol stack offload. The I/O interconnect options considered
included RapidIO and InfiniBand, and the transport implementation strategies included shared memory and VIA-based transports for communication with user and kernel space TCP offload agents.

The prior work did not attempt to quantify the system and application level factors that contributed to the observed CPSO performance effects. The system level effects were not characterized in terms of their interaction with application workloads. No application-independent metrics for quantifying CPSO performance effects were proposed.

In contrast to these research efforts, the study of the CPSO phenomena conducted in this dissertation focuses on the analysis of CPSO performance effects with the use of the system level metrics $D_o$, $D_{accs}$, and $S_{comm}$ and the application level metric $R_{app}$. These metrics are used in order to quantify performance effects of several CPSO scenarios. The interaction of the system and application level factors that contribute to the observed CPSO performance effects is discussed. This research is complimentary to the efforts reviewed above and therefore merits further consideration.

2.2 Networking subsystem performance optimizations

As discussed in Chapter I, the networking subsystem of a cluster node is a collection of interacting hardware and software components not limited to NIC hardware and firmware. Along with other software components, it includes the communication protocol stack and the network device driver. These components operate in the runtime environment offered by the operating systems of running on the host and iNIC CPUs.
The hardware components involved in processing networking tasks may include practically every system hardware component. Therefore, networking subsystem is traditionally tightly integrated with the rest of the computer system. While this organization has its advantages, such as low overhead of the networking subsystem interfaces, it also has some disadvantages. Sharing resources with other subsystems leads to complex interdependencies that complicate system analysis and programming.

From a purely engineering prospective, a system’s performance can be increased by increasing the servicing capacities of the system’s components while maintaining the component utilization in balance (no bottlenecks). The traditional approaches to increasing the servicing capacity of hardware system components include increasing CPU clock speeds and degree of instruction parallelism, increasing memory and I/O subsystem bandwidth by means of memory hierarchies and increased system interconnect signaling rates, and balancing data bandwidth and CPU instruction throughput [68]. Recent developments in networking technologies demonstrate a complimentary approach: elevating the hardware/software system interface to a higher-level of abstraction. This approach [16, 23] is characterized by virtualization of the hardware and offering the application programmers direct protected access to the hardware at little or no added cost in terms of the fraction of raw hardware performance. Additionally, system performance can also be increased by means of reducing the overhead incurred for support of the software abstractions that constitute application run-time environment.
2.2.1 Revising hardware/software system interfaces

Virtual Interconnect Architecture (VIA) [23] and InfiniBand [16] specifications are among the most interesting recent networking technologies. Both technologies are designed for System Area Network (SAN) environments characterized by very high signaling rates and very low data error rates approaching those of traditional memory subsystems. The hardware guarantees reliable and ordered data delivery without duplications. Therefore, these functions need not be performed in the communication protocol stacks.

Both technologies expect network devices — VIA NICs and InfiniBand Channel Adapters — to provide direct support for high-level abstractions of the communication endpoint. Each endpoint includes a pair of request queues for data transfer operations such as send, receive, and RDMA read and write operations. All the data buffers and request control data structures are accessed using their virtual addresses. The addresses are validated and translated by the hardware at runtime; this provides the degree of protection normally associated with private process address spaces.

The transfer requests are submitted by the service consumer and are carried out asynchronously and concurrently with other system activities, thereby allowing for overlap of communication and computation performed by the service consumer threads. The service consumer can choose to be asynchronously notified of some request completion events [16, 23]. Additionally, the service consumer can control generation of the asynchronous request completion notifications on the remote side [16]. This feature allows for tighter control over CPU usage by the communication subsystem.
Both technologies provide for delivery of request completion notifications from multiple endpoints associated with a single adapter to a single completion queue. InfiniBand defines a comprehensive set of attributes that describe quality of service (QoS) associated with a given data transfer. These parameters are enforced by the interconnect fabric and are specified per transfer request.

All the above-mentioned features are supported by the endpoint hardware directly. Therefore, they are available at little or no cost in terms of CPU overhead. The corresponding data structures that constitute software/hardware interfaces (request and completion queues, message headers, and data buffers) are made available to the hardware directly by means of registration of memory regions that contain these data structures with the hardware. The registration process ensures proper alignment and pinning of the regions in order to prevent them from being swapped out by the virtual memory subsystem. Direct hardware access to the request queues and data buffers allows one to avoid intermediate data copies and to eliminate the operating system involvement on performance critical execution paths.

Another important feature of modern I/O subsystems discussed in the InfiniBand specification is their departure from shared bus architectures in favor of switched interconnection technologies. Switched fabrics do not have the performance and scalability problems of shared busses because of the ease of modular expansion of and provision for scalable bisection bandwidth available for system I/O. For this reason, system I/O busses such as PCI are likely to be replaced by chip-to-chip memory-mapped cache-coherent Global Shared
Memory (cc-GSM) switched interconnects such as RapidIO [15]. The resulting system architecture presents a hierarchy of switched fabrics with cc-GSM-style connectivity inside cluster nodes, and message-passing-style switched fabrics for inter-node communication.

In summary, VIA and InfiniBand technologies address performance problems of legacy networking subsystems, such as outdated assumptions about low quality of physical media, high CPU overhead for execution of the protocol stacks, high memory bandwidth demands, and excessive involvement of the operating system on the performance critical execution paths. This is made possible by several advances in networking technology: improvements in physical link technology, support for high-level abstractions such as transfer request queue pairs in network interface hardware, and introduction of the memory-bandwidth-efficient RDMA operations. The improvements in performance of networking subsystem are directly relevant to the case of clusters, as the network is a primary medium for inter-task communication in such systems.

### 2.2.2 Minimizing overhead of runtime environments

Performance of the networking subsystem is determined both by its hardware and software components. On one hand, raw host and iNIC CPU speed, memory and I/O subsystem bandwidth constitute raw computing resources of the system. On the other hand, operating system software, protocol stacks, and middleware determine the degree of useful utilization of the computing resources.
Operating system software is designed to offer abstractions of computing resources that can significantly simplify the task of programming computer systems [98]. However, these abstractions also incur certain overheads that detract from system performance. For instance, virtual processor abstraction used by modern multi-tasking environments simplifies development of concurrent systems, but consumes a certain percentage of CPU and memory resources for task scheduling and storing task execution contexts. The use of a virtual memory subsystem presents a large, uniform, and protected address space for instructions and data, but introduces performance irregularities caused by TLB misses and page faults. File systems present file abstractions for structured data storage, but incur disk space, memory space, and CPU overhead for maintaining these abstractions and effectively decrease disk I/O performance. This list of examples can be extended to include other OS abstractions.

Transport level protocol stacks such as TCP/IP [42] provide applications with a convenient byte stream abstraction. The issues of ordered data delivery without loss and duplication are addressed by the protocol suit at the expense of a fraction of CPU performance, a fraction of memory space used for transport-level buffering, and a fraction of network bandwidth used for control message traffic.

Finally, message-passing middleware provides applications with a set of abstractions that support certain semantics of the runtime environment and message-passing primitives. These abstractions are usually built on top of the operating system and communication protocol stack software, and they also consume system resources. For example, support for
the communication contexts, process groups, and virtual topologies comes at the expense of maintaining a set of internal distributed objects that store relevant configuration information. The system memory required for the objects, the fraction of the nodes’ CPU time dedicated to execution of the internal middleware object management protocols, and the fraction of network throughput consumed by the object management traffic constitute the overhead for support of the convenient abstractions mentioned above.

2.2.3 Application-level latency hiding techniques

Application programs can also affect overall system performance by means of their choice of algorithms and their use of application-level techniques for efficient system resource utilization, such as concurrent execution and latency hiding. Several application and system characteristics are of importance when considering the degree to which an application can take advantage of the latency hiding techniques. These characteristics were discussed by R. Dimitrov in [26].

As was pointed out in [26], the system architecture has to exhibit concurrency in communications and computations. The communication and computation should be performed by separate concurrent subsystems (such as iNIC and host CPU) asynchronously, and these subsystems should not be in conflict for necessary system resources (such as memory and I/O bandwidth, buffer space, and CPU cycles).

Additionally, the application should be enabled to take advantage of the available means of concurrent communication and computation by proper design and implemen-
tation techniques. For instance, the structure of application programs should allow for issuing communication requests and execution of the application-specific computations while these communication requests are being satisfied by the communication subsystem concurrently with the application-specific computations [26].

R. Dimitrov has proposed a metric for quantifying the degree of concurrency afforded by the communications subsystem. The degree of overlapping $D_o$ is defined in [26] as a ratio of the overlapped communication time and the total communication time incurred by the application. The author has also defined a procedure for measurement of $D_o$ and discussed an analytical approach to determining a message size that would offer an optimal overlap of computation and communication in order to minimize total execution time, based on a given value of $D_o$.

The results published in [26] offer a convenient way of assessing the ability of a given communication subsystem to facilitate application-level overlap of communications and computations. The metric $D_o$ can either be modeled or measured experimentally, and it can be used in order to quantify the system’s ability to perform concurrent processing of application-specific computation and communication requests under the conditions of the protocol stack offload. The metric is used in this dissertation for this purpose.

2.2.4 End-to-end argument

The complexity of modern hardware and software makes system performance analysis a challenging task. However, some general principles, when applied consistently, allow
one to avoid excessive overheads for support of convenient runtime environments. A system design principle known as the “end-to-end argument” [84] suggests that in a layered system architecture that is to deliver a set of services on the top level, the mechanisms that deliver these services should be pushed to the higher system layers, as opposed to implementing these mechanisms in the lower system layers.

The rationale is that even though implementation of these services in the lower layers may allow for their use by intermediate system layers and for more modular design, any single implementation of these services is unlikely to meet the needs of multiple applications, and therefore, the services are likely to be extended and duplicated in the higher system layers. Examples that illustrate the problem include bit error recovery, acknowledgement delivery, and duplicate message suppression [84]. However, the argument itself is applicable in a far more general setting. It advocates simplification of lower-layer system services, such as operating system and middleware services, and provisions for extensibility and specialization of these services to meet specific needs of applications.

2.2.5 Exokernels

The ideas of flexibility and application-specific customization of the system-level mechanisms and abstractions were also considered in the framework of microkernel and exokernel approaches to operating system architecture. The microkernel approach emphasizes a minimal kernel that exports abstractions of protected execution contexts and IPC for communication with privileged user-mode subsystems that deliver the rest of familiar
runtime system services (file systems, networking, *etc*) [49]. The exokernel approach provides applications with protected access to system resources, as well as to all the low-level resource state information, while deferring resource management to applications [30, 46].

System resource management that is traditionally implemented by the operating system and trusted servers is performed by applications or by “library operating systems” linked with application code. This approach allows for a significant degree of extensibility and specialization of the system services by applications. Therefore, the approach provides for a high degree of control over the associated system performance price paid for the services. For example, a web server application can take advantage of specialized file system and TCP implementations optimized to perform specific tasks (finding a document file and sending it over a TCP connection) in order to increase application performance.

It is interesting to note that the exokernel and VIA/InfiniBand approaches attack the issue of minimization of system performance overheads from the opposite directions. While the former pushes the system interface closer to the hardware, VIA/InfiniBand elevates network hardware interfaces to provide an inherently simpler and easier to work with abstraction that can be directly exported to the user level. Similarly, interfaces to other subsystems, such as CPU and virtual memory, may be revised to better fit their user-level actions. As an example, Anderson, et al in [5], gives a system multi-threading interface that better addresses the needs of user-level concurrency management. Formulation of better semantics for system-level interfaces to other subsystems present a rich range of topics for further research.
2.2.6 Resource Kernels

Another approach to exporting control over system resource usage to applications is presented by Resource Kernels [67]. The central idea of the approach is to introduce a kernel-mode subsystem (resource kernel) that would export to the application a set of services for reserving shares of necessary resources. Therefore, the resource kernel subsystem, in cooperation with the host kernel, guarantees availability of the resources with the QoS parameters specified by applications.

System resources are presented in terms of resource reserves and resource sets. The former term denotes shares of specific system resources, such as CPU time, I/O subsystem bandwidth, network link bandwidth, physical memory for buffering, etc., requested by the application. The latter term denotes aggregations of the reserves. One or more applications can dynamically attach to and detach from a given resource set.

The resource kernel subsystem performs QoS provision and management functions [7] based on the requested resource sets. These functions include admission control upon resource set creation, scheduling and reservation of specific resources included in the sets, resource usage accounting, and enforcement of the resource management models characterized by their resource usage and replenishment policies. Hard, Firm, and Soft resource management models [67] differ in their strictness of enforcement of resource consumption within the specified QoS (that is, in the degree to which the resources are allowed to be over-consumed during a given recurrence period, and how this over-consumption is accounted for during the following periods).
Resource kernels present an approach to building real-time extensions of general-purpose kernels that provide explicit timing guarantees and enforce availability of system resources requested by the application [79]. Resource kernels are an example of a more general QoS-based approach to controlling system resource utilization for the purpose of providing better quality of application service to consumers [7]. For instance, a media application can deliver audio/video streams with better resolution and audio/video synchronization given control over the source-to-playback resource reservation in the system [78]. In the context of non-real-time applications, end-to-end QoS can be used for elimination of the system resource virtualization and multiplexing overheads incurred by commercial general-purpose kernels. Indeed, a requested share of resources can be effectively dedicated to a given application or a set of applications, and no further virtualization and multiplexing is necessary. Additionally, QoS-based resource control minimizes resource contention and therefore decreases synchronization overheads as discussed in the next section.

2.2.7 QoS-based resource management

The cost of inter-task synchronization in concurrent software systems is often increased because contention for local system resources adds random delays to task execution intervals between synchronization points. The local resources in question include CPU, memory, and network bandwidth [12, 13]. In contrast to precedence constraints or mutually exclusive access requirements imposed by the algorithms, the above-mentioned
variance in task execution times is caused by temporary unavailability of system resources due to the specifics of system resource scheduling policies and mechanisms (for instance, round-robin CPU scheduling or FIFO network subsystem scheduling).

An approach to the scheduling of tasks in concurrent systems that takes these issues into account is called gang scheduling [13, 32, 44]. Cooperating tasks are given access to necessary local resources (or scheduled) concurrently. Therefore, the time needed for each task to reach the next synchronization point is not affected by contention for the resources, and consequently, synchronization overheads are also minimized.

A natural extension of this approach is an integrated framework for task scheduling and QoS-based system resource management, where system resources are provided to the tasks based on their QoS requirements, and the system scheduler uses system resource reservation information for making task scheduling decisions [32]. This approach minimizes variances of task execution times due to resource contention and therefore decreases inter-task synchronization overheads. In addition to increasing performance, the approach also makes concurrent systems more amenable to modeling. Indeed, this approach minimizes random effects of synchronization costs due to resource contention, which has been a traditionally difficult phenomenon to model, as discussed below.

2.3 Performance analysis of computer systems

Although the approaches to increasing performance of computer systems mentioned in the earlier sections address various performance limitations of specific subsystems, they
do not contribute to an integral understanding of the system’s performance. One needs a system model in order to analyze the relative contribution of the system components in the overall system performance, as well as for understanding the scalability limits imposed by these components.

Performance analysis and modeling of computer systems may be of interest for a variety of reasons. Analysis of existing systems allows for better understanding of how to design high-performance systems. The analysis also allows for prediction of performance for system design alternatives and for selection of system configurations based on specific application requirements, while achieving optimal price-performance metrics.

Many factors such as resource contention, precedence constraints, and input-dependent execution time variability contribute to non-deterministic nature of models describing computations. Stochastic analytical models explicitly acknowledge this issue. The commonly used frameworks for stochastic modeling include Queueing Networks [50] and Stochastic Petri Nets [70]. These frameworks are briefly discussed below.

### 2.3.1 Queueing networks

Queueing models represent a well-established approach to performance analysis of computer systems [36, 43]. A queue is characterized by customer arrival process, service time distribution, number of servers, queue buffering capacity, customer population size, and queueing discipline. A Queueing Network (QN) is a directed graph with nodes representing queues (service centers) that model system resources.
Depending on the service semantics, the service centers are classified as single servers, multiple servers, or infinite servers (delay centers) [43]. Customers travel along the edges of the graph from one queue to another and compete for the resources (server capacity) modeled by the queues. The edges have routing probabilities assigned to them; along with the graph topology, these probabilities affect customer distribution patterns in the network.

Depending on the arrival rate and service rate of a particular queue in the network, a line of customers may develop waiting for the associated service. The customers are classified depending on whether they arrive from or depart to some external source, or constantly populate the system. Depending on the classes of customers that populate the network, the network may be closed, open, or mixed.

The state of the system is characterized by steady state queue lengths and customer arrival rate distributions at the nodes of the network for each class of customers. System performance metrics can be derived as follows. Resource utilization is a ratio of steady state arrival rate and service rate, service center throughput can be calculated as a ratio of the average steady state queue length and mean wait time (includes waiting in the queue and service time), and latency is defined as mean wait time in the service center queue [43].

A large class of queueing models has been shown to have reasonably straightforward solutions [64]. Individual queues in the network of this class can be treated independently, and the solution of the model, termed product form solution, can be efficiently computed [43]. Queueing modeling tools that automate the model solution process and provide the
user with requested performance metrics while hiding the mathematical complexity of the solution, are available [69].

The known limitations of the QN-based models include difficulties in modeling some system conditions, such as conditional synchronization [45], simultaneous resource possession, fork and join, blocking, and mutual exclusion [43]. Despite their limitations, QNs are widely used in performance modeling [50]. In this dissertation, QN-based models were chosen for the analysis of the CPSO performance effects as a compromise between the model’s predictive power and tractability. The author has implemented the Open, Closed, and Deterministic QN-based models in C programs and was able to experiment with various approaches to calculating the sought metrics. This flexibility was important in the study of the model’s applicability to the performance analysis of CPSO scenarios.

2.3.2 Stochastic Petri Nets

Petri Nets are a well-established tool for analysis of structural and behavioral properties of systems, as well as for system performance analysis [53, 63]. A Petri Net is a directed graph with nodes of two types, places and transitions [36]. Tokens travel between the nodes of the net according to the following rules. When all places connected to a transition have at least one token, the transition fires, which results in one token taken from each place connected to the transition, and one token moved to each place to which the transition connects. A given distribution of the tokens is called a marking of the net, and it characterizes the state of the system. The reachability set is formed by all possible mark-
ings that can be arrived at from initial net marking by applying the firing rules to propagate the tokens through the net.

Petri Net models allow for analysis of various system properties such as the possibility of deadlocks, system buffer overflows, and mutual exclusion. They are well suited for representing concurrency and synchronization aspects of system behavior, which accounts for their wide use in performance modeling and for various extensions developed by researchers in this area [37, 53]. Stochastic Petri nets (SPN) are extensions of the Petri Nets proposed by Molloy [60] where the firing rules of transitions are extended by associating firing rates with the transitions. When all the places connected to the transition have tokens, a random delay is introduced before the transition fires. If at the end of the delay the transition is still enabled, it fires. When the delays are exponentially distributed, the reachability set of the net is isomorphic to a Markov process [59], and therefore, SPNs can be analyzed using stochastic techniques (Markov process solutions). The possibly very large state space of the corresponding Markov process complicates use of SPNs in performance modeling.

A further refinement of SPNs are generalized SPNs (GSPNs) [53, 54, 55]. In GSPN, the transitions are partitioned into two subsets: timed and immediate. The timed transitions behave like transitions in SPNs, and immediate transitions fire as soon as they are enabled. Accordingly, the reachability graphs of GSPNs are partitioned into tangible and vanishing markings. It was shown that vanishing markings can be eliminated prior to the solution of the underlying Markov chain thereby simplifying the solution [53].
GSPNs can also include inhibitor arcs that disable a transition if a sufficient number of tokens is present in a place connected to the transition by the arc. These extensions allow for more concise expression of the model. It has been shown that the modeling power of GSPNs is equivalent to that of Turing machines [53]. GSPNs have been widely used for performance modeling [8, 9, 55].

GSPN-based performance evaluation and modeling tools that hide the mathematical complexity of model solutions are widely available [61]. Such tools appear to be a good match for performance analysis needed in this dissertation research. GSPN-based models do not have many limitations attributed to QN-based models such as difficulties in modeling many of the important aspects of concurrent architectures (simultaneous resource possession, blocking, and mutual exclusion) that may be important for correct representation of the system under investigation.

However, at the time of writing the dissertation, the author did not have the automated GSPN-based performance analysis tools at his disposal. Implementing a GSPN-based model from scratch required a significantly larger development effort as compared to the implementation of a QN-based model. As mentioned earlier, this was one of the reasons for selecting QN-based models for the analysis of the CPSO performance effects.

2.3.3 Hierarchical stochastic models

Stochastic models have previously been used in a hierarchical modeling framework. In his dissertation [2], Adve reports on several hierarchical stochastic models that share
common structure [47, 58, 100]. The models are designed for performance analysis of parallel programs and are based on task graph presentation of structure of parallel programs. The task graphs considered are fork-join and series-parallel, and arbitrary task graphs.

The models consist of two levels [2]. The higher-level model component deals with task-graph-related issues of the model such as task creation and termination, task scheduling, and synchronization. Given the task execution times, this model component allows for calculation of the overall program execution times and other metrics of interest. The lower-level model component is designed to represent such system-level effects as communication overheads, network contention, etc. This model component, often represented by a queueing network, allows for calculation of execution time of individual tasks used in the higher-level model component. Adve notes [2] that the most difficult part in solving these models is to find a way to perform the computation of the system performance metrics on the higher model level in a reasonably efficient way.

Adve reports on the following drawbacks of the above mentioned approaches [2]. The computationally efficient solutions are obtained for simple task graph structures that describe a restricted class of parallel programs. Models with more general task graphs consider exponentially distributed task execution times for reasons of tractability, which implies high variance that is not always justified. Finally, model state space that accounts for all possible combinations of tasks in execution often grows combinatorially and therefore may be computationally very expensive for a large number of tasks.
2.3.4 Deterministic models

As Adve showed in his dissertation [2], for some classes of shared-memory parallel programs and architectures, communication and resource contention delays primarily affect task mean execution times, while having insignificant effect on variances of task execution times. Therefore, task execution times can be considered deterministic quantities equal to the task CPU requirements plus mean communication and contention overhead. Adve developed a simple and computationally efficient deterministic higher-level model for such parallel programs. Combined with the system-level queueing network model for calculation of mean communication and contention overheads, the deterministic model was shown to provide reasonably accurate estimates of program performance metrics [2].

Other deterministic analytical approaches to modeling system performance consider high-level behavior and scaling of parallel applications [37]. These approaches are concerned with analysis of parallel speedup and efficiency of parallel applications [11, 28, 35, 96, 48].

The application is characterized by its parallelism profile that describes the application’s parallelism as a function of time, as well as by the notion of the average parallelism (parallel profile averaged over time) [28]. The speedup is studied under various scenarios of system scaling (fixed work, fixed time, memory bound, etc.) [34, 96]. The isoefficiency function [48] is introduced as a means of characterizing a combination of parallel algorithm and parallel computer system in terms of how fast one needs to grow the problem size with increasing number of processors in order to maintain constant parallel efficiency.
These approaches are complemented by *ad hoc* models for arriving at the measures of the total communication and synchronization overhead as functions of the problem size and number of processors. The models often use mean costs of basic system operations derived from some external sources. These approaches provide many useful insights into the behavior of parallel applications and advance understanding of fundamental laws that describe behavior of parallel applications.

### 2.3.5 Simulation

Simulation modeling is another technique used in performance analysis of computer systems. Whereas analytical methods describe a system in terms of analytical dependencies between the system’s parameters of interest, a simulation model represents the system as a collection of interacting mechanisms that mimic the system’s operation on a desired level of abstraction. Simulation models are often driven by recorded sequences of events generated during operation of the system under modeling. Alternatively, the event sequences may be synthetic, e.g. generated by other models.

Simulation is widely used for performance analysis of well-defined components of computer systems such as processor architectures [66], memory hierarchies, interconnection networks, etc. For example, simulation of program execution targeted for a given architecture by running a program that models this architecture on a host architecture is often referred to as execution-driven simulation or emulation [37]. Simulation of cache subsystems that use a program trace is referred to as trace-driven simulation, and sim-
ulation of network packet processing described by sequences of packet arrival/departure
events is referred to as event-driven simulation [37].

A simulation of a multiprocessor subsystem running a web server application was used
in [18] in order to assess the effects of high-performance I/O interconnects on the through-
put of the resulting system. In application to the performance analysis of networking
subsystems, an Emulated Network Device (END) framework was used for high-fidelity
evaluation of NIC architectural tradeoffs [39].

As the number of the simulated system components and their interaction increases,
building simulation models becomes increasingly difficult. The complexity of system
components that reflects the simulation abstraction level also contributes to the overall
complexity of a simulation, and therefore to the total execution time. Simulation models
are capable of providing accurate quantitative characteristics of systems under modeling,
but often require detailed modeling of complex system structures. This often accounts for
slow simulation runs. Research in Parallel Discrete Event Simulation (PDES) addresses
this and related issues [37].

2.3.6 Performance measurements

Measurement of performance of computer systems represents another performance
analysis technique. The objective is to measure system performance metrics of interest
(latency, throughput, and component utilization for instance) in a series of experiments
that exercise the system under workloads of interest[43]. The measurement tools should
ideally be minimally intrusive in order to avoid significant effects on system performance under measurements. Hardware-supported event counters [51] allow for low-overhead measurements that meet this requirement.

Since the performance measurement results may be dependent on a given workload, many standardized synthetic workloads (benchmarks) are widely used for performance characterization of computer systems [93]. The measurement technique complements analytical modeling and simulation in that it allows for validation of analytical models with direct measurements, as well as for collection of traces that drive simulations. In this dissertation, the measurements are used for the thesis validation and in the evaluation of the QN-based models for CPSO performance analysis.

Performance measurement of computer systems involves experimental design, measurement, and interpretation of experimental data [37, 43]. Although a good tool in many cases, measurements may be prohibitively expensive in terms of required resources, or impossible because of the lack of non-intrusive measurement instruments, or because of the destructive effects on the system under measurement and/or the environment.

2.4 Summary and conclusions

This chapter presented a review of relevant research materials published to date. The techniques for reducing the communication protocol processing burden on the compute subsystem, including communication protocol stack offload, were discussed. It was pointed out that although many research efforts have considered systems with protocol
stack offload capabilities, and some research was done in performance analysis and modeling of such systems, this research and models were not focused on the general evaluation of the effects of CPSO phenomena on the performance of a class of applications. It was noted that in contrast, this dissertation research was focused on such analysis of the performance effects of several CPSO scenarios.

After that, several performance analysis techniques were reviewed, including analytical modeling techniques, simulation, and performance measurements. It was mentioned that the performance measurements were chosen as an approach to the thesis validation, and the QN-based models were chosen as a tool for analytical modeling of the performance effects of CPSO scenarios.
CHAPTER III

APPLICATION AND SYSTEM LEVEL METRICS

This chapter introduces the terms and definitions that are used in the subsequent chapters of this dissertation. The chapter characterizes the target application domain and presents an approach to the analysis of application execution time. The chapter also discusses the extensions of the notion of the protocol stack offload that are instrumental in making the performance analysis applicable to a broader range of system optimizations.

The chapter offers definitions of the application and system level metrics that are used in the performance analysis. The application metrics are designed to characterize the relevant properties of the application programs that are considered in this work. The system-level metrics are designed to characterize the effects of the CPSO scenarios on the system properties that affect the application performance. Taken together, the application and system level metrics allow one to analyze the effects of the CPSO scenarios on the application performance. The discussion of the system-level metrics presented in this chapter is used as a basis for the dissertation thesis validation procedure discussed in Chapter IV. The chapter is concluded with the summary of the presented material.
3.1 Characterization of the target application domain

Let $\Omega = \{\omega\}$ represent a set of applications that comprise the application domain of interest in this work. An application $\omega \in \Omega$ consists of one or more consecutive execution phases. Each phase involves a certain (possibly zero) amount of application-specific computation and a certain (possibly zero) amount of communication over the network. The computation and communication activities performed in a given phase are independent from the activities performed in other application phases. Specifically, the effects of the phase interaction due to the overlap of communication and computation, as well as due to the contention for shared system resources, are insignificant and can be disregarded during the application performance analysis.

Even though the analysis developed in this chapter is applicable to zero-communication-time application phases, they are not of particular interest. The discussion below focuses on the execution phases with non-zero communication time since the primary goal of this work is to analyze the properties of networking subsystems and their effect on application performance.

The above-mentioned subdivision of the application execution time into the execution phases is somewhat arbitrary. The condition of negligible interaction between the phases is the only limitation to be considered when choosing a set of phases for the analysis of the application execution time. The primary goal of introduction of the execution phases is to reduce the complexity of the application performance analysis by narrowing the scope of this analysis, as discussed later in this section.
A reasonable approach to defining the boundaries of the execution phases is to use the scope of the data and control dependencies between the application-specific computation and communication portions activities. Indeed, the scope of these dependencies identifies the portions of the application where the effects of overlap of communication and computation might be significant. Therefore, the scope of the dependencies suggests the boundaries for the execution phases. These boundaries can be identified in program text; they often correspond to the networking subsystem interface calls to post communication requests and wait for their completion.

Nevertheless, the specific choice of boundaries between the execution phases is not particularly important. The application can always be thought of as a single execution phase. Once the choice is made, it defines the execution phases, and allows one to attribute a portion of the total application execution time to each phase. The total application execution time $T_{app}$ is a sum of the phase execution times, and $T_{app}$ is used as a measure of the application performance. $T_{app}$ is clearly independent of any specific way of conceptual subdivision of the application execution time into a set of phases. Since the phases are assumed to be executed sequentially, the total application execution time can be represented as a sum of the execution times of its $n$ phases,

$$T_{app} = \sum_{i=1}^{n} T_{\omega ph_i}$$  \hspace{1cm} (3.1)

and the analysis of the application execution time can be reduced to the analysis of each of the execution phases. Therefore, without loss of generality, one can concentrate on the analysis of one of the phases.
One can observe that the set of applications of interest includes practically any scientific cluster application with non-trivial amounts of communication. The details of the application-specific algorithms used in the computation are of no particular importance. The application-level parameters considered in this research characterize the workload on the compute and networking subsystems imposed by the application.

The parameters of the communication requests, including the rate of the communication requests and request types (e.g. synchronous vs. asynchronous requests), communication buffer size and alignment, buffer physical contiguity and caching state, as well as the total time the application spends performing communication-related processing during a given phase, are the parameters that characterize the networking subsystem workload. On the other hand, the memory bandwidth required by the application-specific computation and the total time the application spends performing this computation characterize the compute subsystem workload.

Since the compute and networking subsystems include overlapping sets of system components, the workloads that the application imposes on these subsystems interact. In many cases, the nature of this interaction is contention for the shared system resources. This interaction is affected both by the application-level parameters mentioned earlier, and by the system-level factors such as compute and networking subsystem processing capacities, as well as the resource demands on the compute subsystem imposed by the processing of the communication protocol stack. The latter factor is in direct relation with the networking subsystem configuration specifics described by CPSO scenarios.
The specifics of CPSO scenarios are independent of the application-level parameters. Therefore, in order to study CPSO performance effects, it is desirable to distinguish between the application-level factors and the system-level phenomena that can be directly related to the CPSO scenarios of interest. The approach taken in this research is to use the ranges of the relevant application-level parameters to organize the applications from the domain of interest $\Omega$ in subclasses (subsets). Within each subset, the application-level performance effects are considered fixed for a set of the application-level parameters. This approach allows one to concentrate on the system-level factors, as the influence of these factors on the resulting application performance could be clearly seen within a given application subset.

This research prospective can be advantageous in many settings. As an example, an application designer may consider using a range of message sizes and certain application-level techniques for increasing cache utilization and minimizing the rate of main memory accesses by the application. The designer may also have estimates for the total communication time and the total computation times of the application. Based on this data, the designer may wish to determine a CPSO scenario and a range message sizes that lead to the desired application performance.

The range of message sizes, the caching state of the application data, the memory access rate, and the total communication and computation times, can be used in order to define a subset of the application domain $\Omega$. The effects of the CPSO scenarios can be
studied for the message sizes of interest, either analytically or experimentally. The sought CPSO scenario and the message size can then be determined.

A short note on the notation used in the following sections. The Greek letters $\omega$ and $\Omega$ are used as superscripts to refer to a specific application $\omega$ from a specific application domain $\Omega$. The Latin subscripts $i, j, k$, as in $T^{\omega}_{ov_k}$, $D^{\omega}_{o_k}$ are used as subscripts to refer to a specific CPSO scenario realized in a specific system configuration $S_k$ with the networking subsystem $N_k$. If the Greek superscripts or the Latin subscripts are dropped, the discussion refers to an application from some application domain, or a CPSO scenario realized by some system configuration, unless explicitly stated otherwise.

### 3.2 Analysis of application execution time

As proposed earlier by R. Dimitrov in [26], the execution time $T^{\omega}_{ph}$ of an application phase with communication can be represented as

$$T^{\omega}_{ph} = T^{\omega}_{comp} + T^{\omega}_{comm} - T^{\omega}_{ov}$$

(3.2)

The $T^{\omega}_{comp}$ is the total time for the application-specific computation attributed to the execution phase and performed in one or more steps with no control or data dependencies on the other execution phases. The computation may, however, depend on the communication activities to be performed in the present application phase. $T^{\omega}_{comm}$ is the total communication time of the application phase performed in one or more steps with no control or data dependencies on the other execution phases. $T^{\omega}_{ov}$ represents the time of overlapped
(concurrent) processing of communication and computation. The $T_{\text{comp}}^\omega$, $T_{\text{comm}}^\omega$, and $T_{\text{ov}}^\omega$ are operational quantities, and they can be measured in a properly designed experiment.

In order to analyze the properties of networking subsystems, let us assume that $T_{\text{ov}}^\omega$ accounts for the following aspects of application processing. In addition to representing the time of concurrent processing of the application-specific computation and the application communication requests, let $T_{\text{ov}}^\omega$ also represent other aspects of the interaction between the compute and networking subsystems, including the contention for the shared system resources between the subsystems.

Consider two hypothetical systems $S_i$ and $S_j$ with identical compute subsystems $C$ and different networking subsystems $N_i$ and $N_j$. The networking subsystems $N_i$ and $N_j$ represent the two CPSO scenarios under comparison. Consider also an application phase characterized by certain amounts of application-specific computation and a certain set of communication requests with specific parameters. Suppose, the first system executes the phase in time $T_{\text{ph}}^\omega = T_{\text{comp}}^\omega + T_{\text{comm}}^\omega - T_{\text{ov}}^\omega$, and the second system executes the phase in time $T_{\text{ph}}^\omega = T_{\text{comp}}^\omega + T_{\text{comm}}^\omega - T_{\text{ov}}^\omega$.

Since the systems $S_i$ and $S_j$ have identical compute subsystems, the application-specific computation attributed to the execution phase is executed by these subsystem identically, and $T_{\text{comp}}^\omega = T_{\text{comp}}^\omega$. Therefore, the time difference $\Delta T_{\text{ph}}^\omega = T_{\text{ph}}^\omega - T_{\text{ph}}^\omega$ is due to the differences in the networking subsystem $N_i$ as compared to $N_j$, and to the difference in the interaction of the compute subsystem $C$ with the networking subsystems $N_i$ and $N_j$. 
The time difference $T^\omega_{gain} = \Delta T^\omega_{ph} = T^\omega_{ph,i} - T^\omega_{ph,j} = T^\omega_{comm,i} - T^\omega_{comm,j} + T^\omega_{ov,i} - T^\omega_{ov,j} = \Delta T^\omega_{comm} + \Delta T^\omega_{ov_{ij}} = T^\omega_{accs_{ij}} + \Delta T^\omega_{ov_{ij}}$ consists of two components. The first component $T^\omega_{accs_{ij}}$ represents the acceleration ($T^\omega_{accs_{ij}} > 0$) or deceleration ($T^\omega_{accs_{ij}} < 0$) of the communication protocol stack processing by the networking subsystem $N_j$ relative to the networking subsystem $N_i$. The second component $\Delta T^\omega_{ov_{ij}}$ characterizes the gain ($\Delta T^\omega_{ov_{ij}} > 0$) or loss ($\Delta T^\omega_{ov_{ij}} < 0$) of time that occurred as a result of changes in the degree of interaction between the compute subsystem $C$ and the networking subsystems $N_i$ and $N_j$. Specifically, $\Delta T^\omega_{ov_{ij}}$ accounts for the increased or decreased degree of the communication/computation overlap and contention for the shared resources in the system $S_j$ as compared to the system $S_i$. To summarize,

$$T^\omega_{gain_{ij}} = \Delta T^\omega_{ov_{ij}} + T^\omega_{accs_{ij}}, \quad \Delta T^\omega_{ov_{ij}} = T^\omega_{ov_i} - T^\omega_{ov_j}, \quad T^\omega_{accs_{ij}} = T^\omega_{comm_j} - T^\omega_{comm_i} \quad (3.3)$$

The following inequalities hold for the quantities discussed above.

$$T^\omega_{comm_i} > 0, \quad T^\omega_{comp_i} > 0, \quad T^\omega_{ov_i} \leq \min(T^\omega_{comp_i}, T^\omega_{comm_i}) \quad (3.4)$$

$$\Delta T^\omega_{ov_{ij}} \leq T^\omega_{ov_i}, \quad T^\omega_{accs_{ij}} \leq T^\omega_{comm_i} \quad (3.5)$$

The subdivision of $T^\omega_{gain_{ij}}$ into the $\Delta T^\omega_{ov_{ij}}$ and $T^\omega_{accs_{ij}}$ components is intended to capture two complimentary effects of CPSO scenarios. $\Delta T^\omega_{ov_{ij}}$, when positive, describes the change in the portion of the application execution phase attributed to the interaction between the compute and networking subsystems, including the concurrent communication and computation due to the offload. $T^\omega_{accs_{ij}}$, when positive, describes the portion of the communication time eliminated in a given CPSO scenario. These quantities are affected
by the application design and implementation techniques, as well as by the properties of
the underlying system. Indeed, an application can always force sequential execution of
the computation and communication steps during its execution phases. This observation
suggests a convenient way of assessing the $T_{ov_i}^\omega$ described in [26] using the sequential
execution of the application as a baseline.

The $T_{accs_{ij}}^\omega$ can also be isolated and measured by forcing the sequential execution of
the computation and communication steps and comparing the execution times of the ap-
lication for two alternative system configurations. Assuming that the interaction between
the compute and networking subsystems is mainly attributed to the overlap of communica-
tion and computation, $T_{accs_{ij}}$ can also be measured directly in a synthetic communication
benchmark of choice by comparing the communication subsystem response times for the
networking subsystem options of interest.

As mentioned earlier, the execution time of an application phase under two CPSO
scenarios $T_{ph_i}$ and $T_{ph_j}$ can be compared to each other, yielding the $\Delta T_{ov_{ij}}^\omega$ and $T_{accs}^\omega$. Al-
ternatively, the execution times $T_{ph_i}$ and $T_{ph_j}$ can also be compared to the phase execution
time $T_{base_{ph}}$ under some baseline networking subsystem configuration. This approach is
instrumental when dealing with more than two CPSO scenarios. When compared to the
$T_{base_{ph}}$ that represents the sequential execution of communication and computation under
the baseline configuration, $\Delta T_{ov_{ij}}^\omega$ becomes $T_{ov_{ij}}^\omega$, because the sequential baseline does not
exhibit any overlap of computation and communication.
Both $\Delta T_{ov}$ and $T_{accs}$ are functions of the parameters of the application communication requests. These parameters include message size, alignment, message buffer physical layout, and message buffer caching attributes. Additionally, the parameters of interest include the communication request type from the set of request types supported by the subject networking subsystem, such as synchronous, asynchronous, persistent, etc.

3.3 Extension of the notion of the protocol stack offload

It should be noted that $\Delta T_{ov}$ and $T_{accs}$ can also be used to quantify performance effects of the other optimizations of networking subsystems discussed in Chapter II. The effects of many end-system and protocol stack optimizations can be conveniently described with the use of $T_{accs}$. Alternative networking subsystems can also be evaluated with the use of $\Delta T_{ov}$ as an indicator of their asynchronous (with respect to the host compute engine) protocol stack processing capacity.

On the other hand, the notion of the protocol stack offload can also be extended to include these optimizations. By definition, the protocol stack offload is the process of moving a portion of a specific protocol stack off the host compute engine onto the iNIC processing engine. From this point of view, the discussion of the protocol stack offload phenomenon should be limited to the analysis of various offload scenarios of this specific protocol in the specific setting of a given system.

Alternatively, the notion of the protocol stack offload can be extended to include the end-system optimizations reviewed in Section 2.1.2. These optimizations lead to the re-
duction of the stack processing time and therefore can be accounted for with the use of $T_{accs}$. Conceptually, the portions of the stack processing that are eliminated as a consequence of a specific end-node optimization can be considered “offloaded” to a hypothetical processing engine with infinite processing capacity (and therefore processed in zero time).

Additionally, the discussion can be extended to include the protocol stack optimizations that suggest the use of alternative light-weight stacks on performance-critical execution paths reviewed in Section 2.1.3. These optimizations can also be accounted for with the notion of the hypothetical processing engine of infinite processing capacity. The original protocol stack can be extended to include the light-weight optimization in the corresponding layers of the stack in addition to the original functionality of the layers. The processing of the original stack is then “offloaded” to the infinite capacity engine.

These extensions of the strict notion of the protocol stack offload are useful in making the performance analysis of the protocol stack offload phenomenon applicable to a broader range of systems, and these extensions can be addressed by the performance analysis techniques and the thesis validation procedure described below. Therefore, when mentioned in the rest of the discussion, the term communication protocol stack offload will refer to the notion of the extended protocol stack offload phenomenon as discussed earlier in this section.
3.4 Application and system metrics

Consider an application execution phase with execution time $T_{\text{ph}i}$ defined in (3.2). The operational quantities $T_{\text{ov}i}$, $T_{\text{ov}j}$ and $T_{\text{accs}ij}$ chosen in (3.3) for the description of the performance effects of the CPSO scenarios, can be used to define the following set of application-specific metrics: degree of overlapping $D_{\text{ov}i}$, degree of acceleration $D_{\text{accs}ij}$, application latency hiding capacity $R_{\text{app}}$, and communication speedup $S_{\text{comm}ij}$.

\[
D_{\text{ov}i} = \frac{T_{\text{ov}i}}{T_{\text{comm}i}}, \quad D_{\text{accs}ij} = \frac{T_{\text{accs}ij}}{T_{\text{comm}i}}, \quad R_{\text{app}} = \frac{T_{\text{comp}i}}{T_{\text{comm}i}}, \quad S_{\text{comm}ij} = \frac{T_{\text{ph}i} - T_{\text{comp}i}}{T_{\text{ph}i} - T_{\text{comp}j}} \tag{3.6}
\]

Given that $T_{\text{ph}i} - T_{\text{comp}i} = T_{\text{comm}i} - T_{\text{ov}i} = T_{\text{comm}i}(1 - D_{\text{ov}i})$, and $\frac{T_{\text{comm}i}}{T_{\text{comm}j}} = \frac{1}{1 - D_{\text{accs}ij}}$, the expression for $S_{\text{comm}ij}$ can be re-written as follows:

\[
S_{\text{comm}ij} = \frac{1}{1 - D_{\text{accs}ij}} \frac{1 - D_{\text{ov}i}}{1 - D_{\text{ov}j}} \tag{3.8}
\]

The $D_{\text{ov}i}$, $D_{\text{ov}j}$ and $D_{\text{accs}ij}$ metrics characterize the degree of overlapping and degree of acceleration of the communication protocol stack processing that occurred during the execution of the subject application execution phase on systems $S_i$ and $S_j$. The $D_{\text{ov}i}$ metric was introduced in [26] and denoted as $d_o$ precisely for the purpose of evaluating the effects of overlapping communication and computation on application performance. The metrics $D_{\text{ov}i}$, $D_{\text{ov}j}$, and $D_{\text{accs}ij}$ can be used in order to characterize the relative effects of CPSO scenarios on $T_{\text{ph}i}$ and therefore, on the application performance metric $T_{\text{app}}$. 
As pointed out in [26], the $T_{\text{comp}}^\omega$ application-specific computation time imposes a limit on the $T_{\text{ov}}^\omega$ — the portion of the communication time $T_{\text{comm}}^\omega$ that can be hidden by the computation during the application execution phase. Therefore, $T_{\text{comp}}^\omega$ can be chosen as a measure of the communication latency hiding capacity characteristic of the execution phase in question. The ratio $R_{\text{app}}^\omega (3.6)$ is then a normalized measure of the communication latency hiding capacity for a given execution phase. $T_{\text{comp}}^\omega$ and $R_{\text{app}}^\omega$ identify the upper limits of the application latency hiding capacity. The latency hiding capacity is further limited by the control and data dependencies between the communication and computation activities within a given application execution phase.

The composite metric $S_{\text{comm}_{ij}}^\omega (D_{\text{oi}}, D_{\text{oj}}, D_{\text{accs}_{ij}})$ characterizes the speedup of the portion of the application execution time that is attributable to communication-related processing. It describes the combined effects of the acceleration of communication processing, as well as changes in the interaction of the communication and computation application activities.

Other composite metrics that are functions of the $D_{\text{oi}}^\omega$, $D_{\text{oj}}^\omega$, and $D_{\text{accs}_{ij}}^\omega$ parameters can be offered as well. The advantage of the communication speedup metric $S_{\text{comm}_{ij}}$ is that the body of research that deals with speedups of computations is directly applicable to the case of the communication speedup as well. The constant work communication speedup that describes the increase of communication performance under CPSO scenarios for the case of transferring a fixed amount of data split in $n$ segments for increasing $n$ is discussed in Chapter V. The extended treatment of $S_{\text{comm}_{ij}}$ is beyond the scope of this work. It is a
subject for future research by the author and his colleagues, and it is discussed further in Chapter VIII.

Based on (3.5), (3.6), and (3.6), one can observe that the following inequalities hold:

\[
D_{o_i} = \frac{T_{oe_i}}{T_{comm_i}} \leq \frac{\min(T_{comp_i}, T_{comm_i})}{T_{comm_i}} = \min(R_{app}^\omega, 1)
\]

\[
T_{accs_{ij}} = T_{comm_i} - T_{comm_j} \leq T_{comm_i}, \text{ and } D_{accs_{ij}} = \frac{T_{accs_{ij}}}{T_{comm_i}} \leq 1
\]

To summarize,

\[
D_{\omega_{accs_{ij}}}^\omega \leq 1, \quad D_{\omega_{o}}^\omega \leq \min(R_{app}^\omega, 1)
\]

(3.9)

This corresponds to the intuitive notion that the positive effects of the stack processing acceleration are limited by the communication time of the baseline scenario, and the positive effects of the concurrent communication and computation are limited by the application latency hiding capacity. At the same time, the negative effects of communication decelerations and an additional overhead described by negative \(T_{accs}\) and \(T_{ov}\) are potentially unbounded.

Consider application \(\omega\) from a subset of the target domain \(\Omega\) defined using a set of relevant application-level parameters as discussed earlier in Section 3.1. Let us denote this subset \(\Omega'\). Consider also a hypothetical system \(S\) that includes a compute engine \(C\) and a networking subsystem \(N\). Suppose, the system can be configured according to two CPSO scenarios described by the system configurations \(N_i\) and \(N_j\). As shown in (3.6) and (3.8), the effects of the CPSO scenarios on the application performance can be characterized with \(D_{\omega_{o}}^\omega\), \(D_{\omega_{accs_{ij}}}^\omega\) and \(S_{\omega_{comm_{ij}}}^\omega\).
The least upper bound of \( D_{\omega_{i}} \) over the subset \( \Omega' \) of the application domain \( \Omega \)

\[
D_{\omega_{i}}^{\Omega'} = \sup_{\omega \in \Omega'} D_{\omega_{i}}^{\omega}
\]  

(3.10)

can be chosen to represent the system-level characteristic that describes the ability of the system configuration \( S_{i} \) to offer a certain degree of overlapping over the subset \( \Omega' \) of the application domain \( \Omega \). The least upper bound

\[
D_{\text{accs}_{ij}}^{\Omega'} = \sup_{\omega \in \Omega'} D_{\text{accs}_{ij}}^{\omega}
\]  

(3.11)

can be chosen to represent the system-level characteristic that describes the ability of the CPSO scenario \( N_{j} \) to accelerate processing of the application communication requests as compared to the CPSO scenario \( N_{i} \). Finally, \( S_{\text{comm}_{ij}}^{\Omega'} \) calculated according to (3.8) using the \( D_{\omega_{i}}^{\Omega'} \), \( D_{\omega_{j}}^{\Omega'} \), and \( D_{\text{accs}_{ij}}^{\Omega'} \) can be used as a system metric that qualifies the CPSO scenarios \( N_{i} \) and \( N_{j} \) with respect to all the applications from the subset \( \Omega' \) of the application domain \( \Omega \).

The primary reason for referring to these metrics as to system-level metrics is that they are independent of any application parameters that characterize application \( \omega \in \Omega' \). The metrics describe the system’s behavior with respect to the set of applications as a whole. As mentioned earlier, for notational simplicity, \( D_{\omega_{k}} \), \( D_{\text{accs}_{ij}} \), and \( S_{\text{comm}_{ij}} \) will be used in order to denote \( D_{\omega_{k}}^{\Omega'} \), \( D_{\text{accs}_{ij}}^{\Omega'} \), and \( S_{\text{comm}_{ij}}^{\Omega'} \) respectively, unless otherwise specified.
3.5 Use of the metrics for studying CPSO performance effects

The system-level metrics $D_o$ and $D_{accs_{ij}}$ allow one to abstract from the application specifics and to study the potential effects of the CPSO on application performance that are applicable within the target application sub-domain. The application metric $R^\omega_{app}$ describes the application’s latency tolerance limits. Combined with the system metrics, it can be used in order to access the effects of the given $D_o$ and $D_{accs}$ on the application execution time. This, in turn, allows one to assess the application performance gains in a given CPSO scenario.

Indeed, based on the expressions (3.2) and (3.3), one can observe that $T_{ph}$ is minimized provided that $T_{gain}$, and therefore, $\Delta T_{ov_{ij}}$ and $T_{access_{ij}}$, are maximized. The latter quantities $\Delta T_{ov_{ij}}$ and $T_{access_{ij}}$ are maximized, provided that $D_{o_j}$ and $D_{access_{ij}}$ are maximized, which in turn, maximizes $S_{comm_{ij}}$, assuming constant $D_{o_i}$. This observation suggests a straightforward way of assessing the effects of CPSO scenarios on application execution time. A scenario that maximizes $D_o$, $D_{accs}$, and therefore, $S_{comm}$, also minimizes the application execution time, assuming constant $T_{comp}$.

The $R^\omega_{app}$ imposes a limit on the $D_{o_i}$ afforded by a CPSO scenario that is of practical use for application $\omega$. The performance effect of the CPSO scenario characterized by $D_{o_i}$ and $D_{access_{ij}}$ on $\omega$ is assessed by using $\min(R^\omega_{app}, D_{o_i})$ instead of $D_o$ in expression (3.8).

Therefore, a method for assessing the CPSO effects on the performance of a subset of applications $\Omega'$ from the application domain $\Omega$ characterized by a given $R_{app}$ can be summarized as follows. One needs to obtain $D_{o_i}$ of CPSO scenario $N_i$ used as a baseline sce-
nario, $D_{o_j}$ of CPSO scenario $N_j$, and $D_{accs_{ij}}$ for these scenarios. Then the communication speedup that characterizes the change of the application communication time attributable to the difference between CPSO scenarios $N_i$ and $N_j$ can be calculated as follows:

$$S_{comm_{ij}} = \frac{1}{1 - D_{accs_{ij}}} \frac{1 - \min(R_{app}, D_{o_i})}{1 - \min(R_{app}, D_{o_j})}$$  \hspace{1cm} (3.12)

This analysis can be used in order to guide optimization of networking subsystem characteristics for a given class of applications characterized by a certain $R_{app}$. In the design of many networking subsystems, there exists a tradeoff between the networking subsystem response time (latency) on one hand, and the subsystem bandwidth and the degree of asynchrony in the processing of communication requests, on the other hand. Using $R_{app}$ as a measure of the required asynchrony in the processing of communication requests, one can use the expression (3.12) in order to quantify the resulting performance effects of the networking subsystem optimization along the $D_{accs}$ dimension, while achieving $D_{o_j} \geq R_{app}$.

The method for assessing the effects of CPSO phenomena on application performance was listed as one of the contributions of the dissertation research in Chapter I. The method presented in this section accomplishes this goal under the condition that it can be shown that the system-level metrics $D_o$, $D_{accs}$, and $S_{comm}$ adequately describe the system-level CPSO performance effects. Therefore, the method is discussed further in Chapter V where the system-level metrics are studied experimentally.
3.6 Summary

This chapter introduces the terms and definitions that are used in the subsequent chapters of this dissertation. The chapter characterizes the target application domain, presents an approach to the analysis of application execution time, and discusses the extensions of the notion of the protocol stack offload.

The chapter defines the application and system level metrics that are used in the performance analysis. These metrics allow one to analyze the effects of the CPSO scenarios on the application performance. The chapter is concluded with the description of the method for quantifying the performance effects of CPSO scenarios on the performance of applications characterized by their latency hiding capacity $R_{app}$. 
CHAPTER IV
RESEARCH METHODOLOGY

This chapter discusses the methodology that is used in order to validate the dissertation thesis. As mentioned earlier, the dissertation thesis is validated empirically, based on the metrics for the performance effects of CPSO scenarios discussed in Chapter III. Therefore, the procedure for the thesis validation is discussed first. Then, the CPSO scenarios under consideration are discussed. Finally, the QN-based analytical models for CPSO performance effects developed and evaluated in the course of the dissertation research are briefly discussed.

4.1 The thesis validation procedure

To remind the reader, the thesis states that the performance effects of CPSO scenarios can be adequately described with the use of $D_o$, $D_{accs}$, and $S_{comm}(D_o, D_{accs})$ metrics discussed in detail in Chapter III. Some system configuration is chosen to represent a baseline for assessment of the relative CPSO performance effects. Each CPSO scenario is characterized by its $D_o$ and by the $D_{accs}$ as compared to the baseline scenario. The system metrics $D_o$, $D_{accs}$, and $S_{comm}$ are used in order to access the performance effects of CPSO scenarios on the application performance.
The dissertation thesis is validated empirically. The values of the metrics $D_o^{exp}$, $D_{accs}^{exp}$, and $S_{comm}^{exp}$ for the CPSO scenarios of interest are derived in the course of experiments. Additionally, the values of $S_{comm}^{calc}$ are calculated using expression (3.8). These $S_{comm}^{calc}$ values are then compared to the $S_{comm}^{exp}$ values derived directly from the measurements. A good agreement of the calculated and measured values of $S_{comm}$ indicates that the metrics adequately describe the performance effects of the CPSO scenarios of interest within the precision of the experiments.

The derivation of the metrics $D_o$, $D_{accs}$, and $S_{comm}$ is based on the measured communication times of the synthetic application benchmark DoBench discussed in detail in Chapter V. The benchmark allows one to measure the communication times $T_{ovl}$ and $T_{seq}$ of a parallel application with and without overlap with computation. The computation time $T_{comp}$ is also measured. These measurements are performed for all the CPSO scenarios of interest and for the baseline scenario.

The benchmark represents a specific subset of the application domain $\Omega$. The benchmark’s parameters that characterize the compute and networking subsystem workloads, as discussed earlier in Chapter III, are considered in the course of the benchmark design and are controlled during the experiments. The goal is to assure that the derived metrics $D_o$, $D_{accs}$, and $S_{comm}$ are sufficiently close to the corresponding system-level metrics defined as least upper bounds over the appropriate application sub-domain. The specific parameters and the techniques for their control are discussed further in Chapter V.
The sought metrics are derived as follows. Let us denote the measured baseline sequential and overlapped execution times $T_{\text{base}}^{\text{seq}}$ and $T_{\text{ov}}^{\text{base}}$, and the measured execution times under the CPSO scenario - $T_{\text{seq}}^{\text{pso}}$ and $T_{\text{ov}}^{\text{pso}}$. Then, $D_o$ and $D_{accs}$ can be calculated as follows:

\[
D_o^{\text{base}} = \frac{T_{\text{base}}^{\text{seq}} - T_{\text{base}}^{\text{ov}}}{T_{\text{seq}}^{\text{base}}}, \quad D_o^{\text{pso}} = \frac{T_{\text{seq}}^{\text{pso}} - T_{\text{ov}}^{\text{pso}}}{T_{\text{seq}}^{\text{pso}}}
\]

\[
D_{accs}^{\text{pso}} = \frac{T_{\text{base}}^{\text{seq}} - T_{\text{seq}}^{\text{pso}}}{T_{\text{base}}^{\text{comm}}}
\]  

(4.1)

(4.2)

Indeed, by definition, $T_{\text{seq}}^{\text{base}} = T_{\text{comp}}^{\text{base}} + T_{\text{comm}}^{\text{base}} - T_{\text{ov}}^{\text{base}}$. Since $T_{\text{seq}}^{\text{pso}}$ and $T_{\text{seq}}^{\text{base}}$ are measured in an experiment with communication and computation performed sequentially, $T_{\text{ov}}^{\text{pso}} = T_{\text{ov}}^{\text{base}} = 0$, and $T_{\text{seq}}^{\text{pso}} = T_{\text{comp}}^{\text{base}} + T_{\text{comm}}^{\text{base}} - T_{\text{gain}}^{\text{pso}} = T_{\text{comp}}^{\text{base}} + T_{\text{comm}}^{\text{base}} - T_{\text{accs}}^{\text{pso}}$. Similarly, in the experiment with overlapped communication and computation under the subject CPSO,

\[
T_{\text{ov}}^{\text{pso}} = T_{\text{comp}}^{\text{base}} + T_{\text{comm}}^{\text{base}} - T_{\text{gain}}^{\text{pso}} = T_{\text{comp}}^{\text{base}} + T_{\text{comm}}^{\text{base}} - T_{\text{accs}}^{\text{pso}} - T_{\text{ov}}^{\text{pso}}.
\]

Consequently, $T_{\text{seq}}^{\text{base}} - T_{\text{seq}}^{\text{pso}} = T_{\text{comp}}^{\text{base}} + T_{\text{comm}}^{\text{base}} - T_{\text{gain}}^{\text{pso}} - T_{\text{ov}}^{\text{pso}} - T_{\text{accs}}^{\text{pso}} - T_{\text{ov}}^{\text{pso}} = T_{\text{ov}}^{\text{pso}}$ and

\[
D_o^{\text{pso}} = \frac{T_{\text{ov}}^{\text{pso}}}{T_{\text{ov}}^{\text{pso}}}, \quad D_{accs}^{\text{pso}} = \frac{T_{\text{base}}^{\text{ov}} - T_{\text{base}}^{\text{comp}}}{T_{\text{base}}^{\text{ov}} - T_{\text{base}}^{\text{comp}}}
\]  

(4.3)

The metrics $D_o^{\text{base}}$, $D_o^{\text{pso}}$, and $D_{accs}^{\text{pso}}$ derived from the measured quantities according to expressions (4.1) and (4.2) can be used in order to calculate the values of the $S_{\text{comm}}^{\text{pso}}$ according to (3.8). Alternatively, $S_{\text{comm}}^{\text{pso}}$ can also be derived directly from the measured $T_{\text{ov}}^{\text{pso}}$ and $T_{\text{comp}}^{\text{pso}}$ as

\[
S_{\text{comm}}^{\text{pso}} = \frac{T_{\text{comp}}^{\text{ov}} - T_{\text{comp}}^{\text{base}}}{T_{\text{ov}}^{\text{pso}} - T_{\text{comp}}^{\text{pso}}}
\]  

(4.3)

according to the definition of $S_{\text{comm}}$ in (3.7).
The comparison of the values of $S_{comm}$ obtained in these two ways is the central point of the thesis validation procedure. A good agreement of these values within the measurements precision indicates that the metrics $D_o$ and $D_{accs}$ can be successfully used in order to quantify the performance effects of the CPSO scenarios of interest. Since the definition of the metrics is not dependent on the specifics of the scenarios, the conclusion about the fitness of the metrics for the characterization of CPSO scenarios can be generalized to include other scenarios not studied experimentally in this dissertation.

4.2 The CPSO scenarios of interest

Given the finite resources available for this work, it is necessary to define the CPSO scenarios of interest. The scenarios reflect the networking subsystem configurations of a PC-based cluster node. The nodes in question are SMP systems running Linux OS with MPI communication middleware and a stock TCP/IP communication protocol stack over a switched Gigabit Ethernet network fabric. More details on the system under test and on the networking subsystem configuration of each of the scenarios can be found in Chapter V.

4.2.1 Checksum offload

In the checksum offload scenario, the transmit and receive checksum calculation is offloaded to the iNIC processing engine. This allows for the reduction of CPU and memory subsystem utilization.
It should be mentioned that the transmit side data copy while computing the checksums on the host processor can be avoided by calculating the checksum inline with other data copies performed while processing the stack. This optimization is utilized in the modern Linux TCP/IP stack implementations. Therefore, this scenario is not expected to result in significant differences in the measured component utilization on the transmit side, as well as response times and throughput of streaming transfers, as compared to the baseline scenario. As discussed further in Chapter V, the experiments have shown that the effects of this CPSO scenario both on the networking performance and on the component utilization is insignificant. Therefore, the checksum offload optimization is considered as a component of other CPSO scenarios, but not as a separate scenario.

### 4.2.2 Data copy avoidance

In the *data copy avoidance* scenario, a copy of the data between user and system buffers on the sender side is eliminated. The user buffers are mapped into the system address space to allow for direct access. They are also marked copy-on-write in order to preserve the existing network I/O semantics of returning control to the user space without final completion notification from the receiver side.

This CPSO scenario allows for reduction in CPU and memory subsystem utilization, reduces response time, and increases networking subsystem throughput. These effects should be more pronounced in the case of streaming transfers.
4.2.3 Large MTU size

Another popular optimization available with Gigabit Ethernet networks is the increase of Ethernet MTU size. The increased MTU size allows one to decrease the number of Ethernet frames needed to transmit user messages of a given size. This allows for reduction of the overall per-packet overhead in the protocol stacks, and it may have a profound effect on the system CPU utilization.

The Ethernet MTU size can be controlled per iNIC with the use of the stock Linux network configuration tools, and it requires the iNIC driver and iNIC hardware support. The Gigabit Ethernet switches that comprise the fabric are also required to support large MTU sizes.

4.2.4 Complete CPSO

In the complete CPSO scenario, the processing of the application communication requests is performed on one of the host CPUs on an SMP system, while the application-specific computation is done in parallel on another processor (other processors). The host CPU engaged in the protocol stack processing plays the role of the iNIC processing engine with the processing capacity equal to that of the host compute engine.

Even though the use of host CPUs on iNICs may not be practical for several reasons, the scenario is interesting because it can be encountered in many SMP systems with middleware that allows for asynchronous processing of application communication requests.
The extended notion of the protocol stack offload discussed earlier allows for the analysis of this scenario in the performance analysis framework introduced in Chapter III.

4.2.5 Complete CPSO with data copy avoidance

In this scenario, the complete CPSO is augmented with the data copy avoidance technique thereby providing for an additional data point for the CPSO performance analysis. As compared to the data copy avoidance scenarios, this scenario increases the opportunities for asynchronous protocol stack processing that affects the $D_o$ metric. As compared to the complete CPSO scenario, this scenario describes the case of acceleration of the protocol stack processing by means of eliminating the send side data copy, and thereby increasing $D_{accs}$.

This scenario is expected to have the largest impact both on the networking performance and the system component utilization by the protocol stack. The performance effects of this and other CPSO scenarios of interest are analysed both empirically and with the use of QN-based analytical models discussed in the next section.

4.3 Models for CPSO performance effects

An additional goal of the dissertation research is to develop a set of QN-based analytical models for studying the performance effects of CPSO scenarios. The models are used in order to obtain the system metrics $D_o$, $D_{accs}$, and $S_{comm}$. 
In the course of modeling the system behavior in a CPSO scenario, the system is characterized in terms of the processing capacity of its components. The components in question include the cluster node compute engine, and the cluster node protocol offload engine. The CPSO scenarios are characterized by the service demands on these system components that result from the communication protocol stack processing by the system in this CPSO scenario.

The models are evaluated as to their complexity and precision based on the comparison of the modeling results with the values of the metrics obtained experimentally as described earlier in this chapter. Based on this comparison, the conclusions about the applicability of the models, the factors that limit the modeling precision, and the ways to improve the models are made. The detailed discussion of the models is presented in Chapter VI.

4.4 Summary

This chapter discusses the dissertation research methodology. The dissertation thesis is validated empirically, based on the metrics for the performance effects of CPSO scenarios discussed earlier. The chapter describes the procedure for the thesis validation, the experiments, and the CPSO scenarios under consideration.

The chapter also briefly discusses the analytical models for the performance effects of CPSO scenarios developed and evaluated in this dissertation. The synthetic application benchmarks designed in order to measure the metrics for the CPSO scenarios of interest and the experimental results are discussed in Chapter V.
CHAPTER V

EXPERIMENTAL WORK

This chapter discusses the experimental work performed in the course of the dissertation research. The details of the experimental platform (the test bed system) and the specifics of the experiments performed are discussed.

The experiments are organized in two groups. The data obtained in the first group of experiments is used in order to derive the system metrics $D_0$ and $D_{accs}$ as discussed in Chapter IV. Based on the results of these measurements, the thesis is validated. The derived metrics $D_0$ and $D_{accs}$ along with the experimental data obtained in the second group of experiments are used for the model verification and for the evaluation of the model complexity versus precision in Chapter VI.

5.1 Introduction

To remind the reader, the process of the dissertation thesis validation discussed in Chapter IV requires experimental derivation of the system metrics $D_0^{exp}$, $D_{accs}^{exp}$, and $S_{comm}^{exp}$. Then, $D_0^{exp}$ and $D_{accs}^{exp}$ are used in order to calculate $S_{comm}^{calc}$. The comparison of $S_{comm}^{exp}$ and $S_{comm}^{calc}$ allows one to make a conclusion about the fitness of the metrics $D_0$ and $D_{accs}$ for
the estimation of $S_{\text{comm}}$ and therefore for the quantification of the performance effects of CPSO scenarios.

The experiments designed to obtain $D_{o}^{\text{exp}}$ and $D_{\text{accs}}^{\text{exp}}$ are performed for the following set of CPSO scenarios discussed in Chapter IV:

- baseline
- large MTU size
- data copy avoidance
- complete CPSO
- complete CPSO with data copy avoidance.

The *baseline* networking subsystem configuration does not include any of the offload options discussed above. The transmit and receive checksum offload is turned off, and the data copy avoidance technique is not used.

The *large MTU size* scenario differs from the *baseline* scenario in that it uses larger Ethernet packet sizes. However, the checksum offload is still disabled, and the data copy avoidance technique is not utilized.

In the course of the experiments, it was determined that increasing MTU size had a significant effect on the communication performance and system resource utilization. Therefore, in the rest of the scenarios, their specific networking subsystem optimizations were combined with the effects of increasing the MTU size used in the *large MTU size* scenario.

As discussed later in this chapter, the operating system running on the cluster nodes of the experimental system is Linux 2.4.18. In the Linux TCP/IP implementation, the
transmit side checksum calculation has to be offloaded in order to enable the \textit{data copy avoidance} optimization. Even though the data copy avoidance technique could theoretically be applied with no transmit checksum offload, this would result in the elimination of only one data move across the memory bus (as opposed to the expected two moves in memory-to-memory copy) because of the in-line partial checksum optimization used in the Linux TCP/IP implementation. The isolation of the effects of the two offload scenarios would require modification of the Linux TCP/IP stack, and it was not undertaken for reasons of limited resources. In the rest of this discussion, the scenarios with data copy avoidance assume the checksum calculation offload option enabled.

The last two scenarios, \textit{complete CPSO} and \textit{complete CPSO with data copy avoidance}, need to be performed on an SMP cluster node. When run on a uniprocessor, they are identical to the \textit{large MTU size} scenario with checksum offload enabled, and the \textit{data copy avoidance} scenario respectively.

The discussion of the experimental results in the following sections refers to these scenarios using an abbreviated notation that encodes the corresponding system configurations. The notation encodes the following information: the number of processors, checksum offload on or off, data copy avoidance optimization on or off, and the MTU size. The reasons for choosing these specific configurations are discussed in more detail in Section 5.4.2.2; the list below presents the abbreviated notation:

- baseline scenario — uniprocessor, 1500 bytes MTU, no checksum offload, no copy avoidance — abbreviated notation: \textit{upxoff1500};
- large MTU size scenario — uniprocessor, 7500 bytes MTU, no checksum offload, no copy avoidance — abbreviated notation: \textit{upxoff7500};
• data copy avoidance — uniprocessor, 7500 bytes MTU, checksums offloaded, data copy avoidance — abbreviated notation: \textit{upzctcp7500};

• complete CPSO — dual SMP, 7500 bytes MTU, checksums offloaded, no data copy avoidance — abbreviated notation: \textit{xon7500};

• complete CPSO with data copy avoidance — dual SMP, 7500 bytes MTU, checksums offloaded, data copy avoidance on — abbreviated notation: \textit{zctcp7500}.

5.2 The experimental platform description

The bulk of the experimental work was performed on a 4-node PC-based cluster with Gigabit Ethernet as a cluster interconnect. The specifications of the main system components are described below. Interested readers are invited to consult the references for more information about the system used in the experiments.

The cluster nodes are Supermicro 1U server blades with P4DPR-6GM+ motherboard [97]. The motherboards include the Intel E7500 (Plumas) chipset, dual Intel Xeon 2.2 GHz CPUs with 12K bytes instruction L1 cache, 8K bytes L1 data cache, and 512K bytes L2 cache, 2G bytes DDR-200 two-way interleaved main memory, 400 MHz front side bus, 100MHz 64-bit PCI-X I/O bus, one Intel 82550 Fast Ethernet (100Mbit/s) controller, and one Intel 82546EB Gigabit Ethernet controller. The cluster network is a switched copper Gigabit Ethernet with an 8-port Gigabit Layer 2 switch Dlink DGS-100T [25]. The switching latency introduced by the switch is about 10 microseconds as measured in a separate experiment. The aggregate bandwidth of the four nodes communicating in pairs through the switch is twice that of a single pair. That is, the 8-port switch does not introduce a bisection bandwidth bottleneck for a 4-node cluster, as expected.
As it turns out, the switch does not support large (greater than 1500 bytes) MTU sizes. Since one of the offload scenarios requires large MTU size, the communication micro-benchmarks are performed on a pair of cluster nodes connected back-to-back by a crossover cable. The performance effects of passing data through the switch can be accounted for with a 10 microsecond increase in data transfer latency.

The nodes run Linux 2.4.18 OS with open-source Intel e1000 Ethernet drivers version 4.4.19 [41], stock TCP/IP stack, and MPI/Pro for Linux MPI implementation from MSTI [62]. The Linux kernel is configured in SMP and uniprocessor modes. The kernel configurations used in the experiments are as follows. First, a stock Linux 2.4.18 kernel in SMP and uniprocessor mode is used for the bulk of the experiments. Second, a Linux 2.4.18 kernel with the sender-side data copy avoidance patch [31] in SMP and uniprocessor mode is used in the CPSO scenarios with data copy avoidance. Third, a kernel with the Linux Trace Toolkit [104] patch for Linux 2.4.18 is used for “sanity checks” of the measured/estimated TCP/IP stack and the Ethernet driver timings.

Other relevant system settings are as follows. The Xeon CPU hyper-threading is turned off using the BIOS option. The Ethernet driver supports transmit and receive side checksum offload, interrupt coalescing, and has a scatter-gather-capable DMA engine for accessing non-contiguous regions in main memory. The segmentation offload capability is not enabled by the driver. The driver is modified in order to control the transmit side checksum offload capability.
5.3 The experimental work on the dissertation thesis validation

As discussed earlier in Chapter IV, the dissertation thesis validation procedure used in this work proceeds as follows. The sought metrics $D_o$ and $D_{accs}$ are obtained according to the procedure described earlier in Section 4.1 based on the measured communication times of a synthetic application-level benchmark DoBench developed by the author. The benchmark is run on the baseline system configuration, as well as under the CPSO scenarios of interest.

The benchmark represents a specific subset $\Omega'$ of the application domain $\Omega$. The benchmark’s parameters that characterize the compute and networking subsystem workloads, as discussed earlier in Chapter III, include the $T_{\text{comp}}$, $T_{\text{comm}}$, the benchmark’s communication request type, buffer size, alignment, and caching state.

In order to assure that the derived metrics $D_o$, $D_{accs}$, and $S_{\text{comm}}$ are sufficiently close to the system-level metrics defined as least upper bounds over the corresponding application sub-domain, the following issues are taken into account in the design of the benchmark, as well as during the experimentation. The benchmark’s $T_{\text{comp}}$ is greater than $T_{\text{comm}}$ by design in order to make sure there is a sufficient communication latency hiding capacity, and the measured values of $D_o$ are characteristic of the system, and not of the $R_{\text{app}}$ of the benchmark. The benchmark uses asynchronous communication requests in order to enable concurrent processing of communication and computation. The message buffer size is controlled in the course of the experiments within a specific range. The message buffers are eight byte aligned, and their size is a multiple of MTU sizes in order to facilitate
processing of the requests by the NIC DMA engine. The caching state of the buffers is controlled as discussed later in this chapter.

For each CPSO scenario of interest, the benchmark is run with communication and computation done sequentially, as well as with communication and computation overlapped. Based on the measurements, the $D_o$ and $D_{accs}$ are derived as discussed earlier in Section 4.1.

### 5.3.1 Benchmark description

The benchmark DoBench is an synthetic application benchmark developed as a part of this research. The benchmark is designed to measure the time that it takes to transfer a fixed total amount of data while performing a commensurate amount of computation for the CPSO scenarios of interest. These measurements are used in order to calculate the *constant work* communication speedup as a function of the number of segments (messages) used for the data transfer and the parameters of the system configuration characterized by a certain CPSO scenario.

The benchmark controls such aspects of interaction between the compute and networking subsystems as application latency hiding capacity, communication request type, buffer size, alignment, and caching state. The amount of computation is chosen such that the computation time is equal to or exceeds the communication time. This is achieved by measuring the average communication time for ping-pong style transfers in the first phase of the benchmark and then by calibrating the amount of computation performed in the sub-
sequent phases. The caching effects that are attributed to sharing communication buffers between the compute and communication subsystems are also considered and controlled.

The goal of controlling the caching effects is to maintain consistency between the groups of experiments performed in the course of the dissertation research, as well as to achieve the caching behavior of the synthetic benchmark that is characteristic of many real-life applications. The following “single user process” approximation of the stack caching behavior is considered for this purpose.

A single running process is likely to have a large portion of the processor cache at its disposal. On the transmit side, the user data buffers are likely to be recently used by the application, and therefore, their size determines whether or not they are in cache. If the buffers fit in the cache, they are likely to be present in the cache entirely; and if they do not fit in the cache, an arbitrary part of the buffer is cached, and therefore the time-averaged stack processing behavior is likely to be similar to the un-cached buffer processing case. On the receive side, the data received from the network is likely to be copied by the stack at least once, and therefore its size determines whether or not the buffer is in cache, analogously to the transmit side case.

This type of caching behavior can be reproduced in a benchmark in the following way. Before performing the operation under measurements, the whole buffer is touched sequentially. Then, the operation is performed, and the elapsed time is measured. This approach of controlling caching effects is applied to the rest of the benchmarks used in this work. This allows one to keep the caching behavior in the series of runs in a group of
benchmarks consistent, and therefore to ensure that they belong to the same sub-domain \( \Omega' \) of the target application domain \( \Omega \).

```plaintext
; first phase - do sequential
; experiment with no computation
ts=time();
if (sender) then
  for i in (0, N-1) do
    send(data)
    recv(data)
  done
else ; receiver
  for i in (0, N-1) do
    recv(data)
    send(data);
  done
endif

; second phase - do sequential
; experiment
if (sender) then
  ts=time();
  for i in (0, N-1) do
    compute(data)
    send(data)
  done
  t-time()-ts;
  tcomm=t/(2*N);
else ; receiver
  for i in (0, N-1) do
    recv(data)
  done
endif

; third phase - do overlapped
; experiment
if (sender) then
  ts=time();
  ; compute the first piece of
  ; data - sender side
  compute(data1)
  for i in (0, N-2) do
    post_send(data1)
    compute(data2)
    test_wait(data1)
    ; reassign the pointers
    switch(data1,data2)
  done
  ; send the last piece of data
  send(data1);
  t-time()-ts;
  tcomm=t/N;
else ; receiver
  for i in (0, N-1) do
    recv(data)
  done
endif

; degree of overlapping
D0 = (tseq-tcomm)/tcomm
```

Figure 5.1 DoBench pseudo-code

The benchmark consists of three phases. In the first phase, the communication time \( T_{comm} \) for the amount of data to be transferred with no computation is measured. The values of \( T_{comm} \) are used in order to calibrate the computational algorithm to compute for at least \( T_{comm} \) seconds. The computation is performed over the entire data segment, and the computational algorithm used in the benchmark performs floating point calculations with double precision characteristic of many signal processing applications that compute 1D
FFTs. In the second phase, the communication and computation activities are performed sequentially, and the time they take \( T_{\text{seq}} \) is measured. In the third phase, the computation and communication are overlapped, and the time \( T_{\text{ovl}} \) is also measured.

The three phases of the experiment deal with the same large buffer of data that exceeds the node's cache sizes. The buffer is divided in a number of segments that are used for storing the data that is exchanged in the process of communication, as well as for the computation. The number of segments and the segment size are recorded for further analysis of their effects on \( D_o \) and \( D_{\text{accs}} \). The benchmark’s pseudo-code is presented in Figure 5.1; \( N \) denotes the number of segments.

5.3.2 Experimental results

DoBench was run for all the CPSO scenarios of interest, including the baseline scenario. The measured \( D_o \) values are presented in Figure 5.2. The measured \( D_{\text{accs}} \) values are presented in Figure 5.3. The standard deviations of the measurements of \( D_o \) and \( D_{\text{accs}} \) metrics are shown in Figures 5.4 and 5.5. The measured \( S_{\text{comm}} \) values, as well as their comparison with the speedup figures calculated based on measured \( D_o \) and \( D_{\text{accs}} \) according to (3.8), are presented in Figure 5.6 and 5.7 respectively (the \( S_{\text{comm}} \) deltas are shown as fractions of the values of the metric). In each figure, the \( X \)-axis plots the exponentially increasing segment (message) size. All the experiments are run with the same set of message sizes. Each experiment is repeated 20 times, and the averages, standard deviations, and
coefficients of variation of the measured and derived quantities of interest are presented in the figures.

![Figure 5.2 $D_o$ measured in DoBench](image)

The measured values of $D_o$ are in line with the expected system behavior. The possibility of overlapping communication and computation increases as one moves from the baseline scenario to the uniprocessor data copy avoidance scenario, to the dual SMP data copy avoidance scenario. The decrease of $D_o$ for a small number of segments (large segment sizes) occurs because only $N - 1$ out of $N$ segments are overlapped according to Figure 5.1, and therefore, the function $D_o(N)$ is modulated by the factor $\frac{N-1}{N}$ that has a pronounced effect for a small $N$. 
The decrease of $D_o$ with the increasing $N$ can be attributed to the accumulation of the per-message processing costs as compared to the total communication processing overhead. Similar dependencies of $D_o$ on the segment size and the number of segments were also observed in [26]. The $D_o$ measurements are further analyzed and compared with the results of the modeling in Chapter VI.

![Figure 5.3 $D_{accs}$ measured during DoBench](image)

The values of $D_{accs}$ (Figure 5.3) are observed to increase with the growing segment size. This observation is consistent with the one that can be made based on the measurements of communication times with MpiBench benchmarks discussed in Section 5.4.2. The behavior of the metric for the CPSO scenarios of interest is somewhat non-uniform. This non-uniformity is attributed to the stochastic nature of the measured quantities.
However, as shown later in Figure 5.17, the values of $D_{accs}$ measured in the experiments with DoBench are not significantly different from the corresponding values measured with MpiBench. This behavior of $D_{accs}$ re-confirms the selection of the metric as one representing the effects of CPSO scenarios on communication performance. The metric does not reflect the impact of the communication processing on the node’s compute subsystem for a given subset of applications from the application domain $\Omega$.

Figure 5.4 Standard deviation of the $D_o$ measurements

The precision of the raw time measurements used to calculate the metrics is better than 4.8% for all the measurements performed in the course of the experiments with DoBench, and it is better than 2% for the majority of the measurements. The standard deviations of the measurements of $D_o$ and $D_{accs}$ metrics are shown in Figures 5.4 and 5.5. The standard
deviations of $D_o$ are relatively large for small values of $D_o$. This can be attributed to the nature of the metric: it is proportional to the difference of two large values - $T_{seq}$ and $T_{ovl}$. Even though the times $T_{seq}$ and $T_{ovl}$ are measured with a good precision, the difference may have a large variance if $T_{seq}$ and $T_{ovl}$ are close.

The coefficient of variation (COV) of $S_{comm}^{exp}$ is shown in Figure 5.8. The relatively large COV can also be attributed to the nature of the metric. According to expression (3.7), $S_{comm}^{exp}$ is a ratio of differences of the large values.

In the course of measurements of $S_{comm}^{exp}$, the computation time $T_{comp}$ was also measured. It was observed that for a given segment size, $T_{comp}$ did not vary significantly across the CPSO scenarios of interest. In fact, the $T_{comp}$ coefficient of variation was 2.4% for 59K segments, 3.9% for 119K segments, and less than 1.2% for the rest of the segment sizes.
Figure 5.6 $S_{comm}$ measured during DoBench

Figure 5.7 $S_{comm}$ delta between the measured and calculated values of the metric
These measurements indicate that the aspects of interaction between the node’s compute and communication subsystems that can be attributed to phenomena other than the overlap of communication and computation, are quite insignificant as compared to the effects of overlapping. The practically constant $T_{comp}$ across the CPSO scenarios of interest indicates that the system-level resource contention effects are insignificant. This is partially because the caching behavior is maintained consistent across the benchmark runs, as discussed earlier in this chapter. Another experimental result that leads to the same conclusion is that the values of $D_{accs}$ as measured with DoBench are very close to the $D_{accs}$ measurements obtained in the course of the experiments with MpiBench as discussed in Section 5.4.2. These observations suggest that values of $D_a$ and $D_{accs}$ measured with DoBench are reasonably close (within the experimental error) to the values of the system-
level metrics defined in (3.10) and (3.11) for the subset $\Omega'$ of applications from the target application domain $\Omega$ represented by DoBench.

5.3.3 The comparison of the calculated and measured values of $S_{comm}$

As shown in Figures 5.7 and 5.8, the values of the composite metric $S^{exp}_{comm}$ derived directly from the measurements according to (3.7) are in reasonable agreement with the values of $S^{calc}_{comm}$ calculated based on the measured $D_o$ and $D_{acces}$ according to (3.8). The difference between $S^{exp}_{comm}$ and $S^{calc}_{comm}$ is fully within the precision of the measurements.

Based on this observation, one can conclude that the proposed metrics $D_o$, $D_{acces}$, and $S_{comm}$ can be used in order to quantify the performance effects of the CPSO scenarios under consideration within the precision of the experiments. As discussed earlier in Chapter IV, this result can be generalized to include other CPSO scenarios not explicitly considered in this dissertation because the notions of $D_o$, $D_{acces}$, and $S_{comm}$ are sufficiently general and do not depend on any specifics of the CPSO scenarios considered in this dissertation. Naturally, the set of restrictions with respect to the target application domain $\Omega$ and the application subset $\Omega'$ mentioned earlier also holds for other CPSO scenarios if the above-mentioned results are to be generalized.

The influence of the benchmark-specific factors, including the buffer attributes and the caching behavior, on the values of $D_o$ and $D_{acces}$ measured in the experiments is controlled but is not explicitly quantified in the course of the experiments. Therefore, as mentioned earlier, the experimental results allow one to conclude that the measured values of these
metrics are within the experimental error from the values of the system-level metrics defined in (3.10) and (3.11) for the corresponding subset of applications $\Omega'$ of the application domain $\Omega$.

However, as discussed earlier in this chapter, the application-level parameters of the DoBench and MpiBench synthetic benchmarks are designed to be characteristic of many real-life message applications. Therefore, the values of the metrics can be considered reasonably close (within the experimental errors) to the values of the system-level metrics for a set of applications with the similar application-level parameters assumed to be characteristic of many message-passing applications.

Consequently, based on the discussion presented above, and the description of the thesis validation procedure in Chapter IV, the dissertation thesis formulated in Chapter I can be considered validated. The primary research goal of this dissertation is achieved, and the first contribution of the dissertation research is delivered.

As discussed in Chapter III, the method for assessing the effects of CPSO phenomenon on application performance is shown to accomplish its goal under the assumption that the system-level metrics $D_o$, $D_{acce}$, and $S_{comm}$ adequately describe the system-level CPSO performance effects. Since the latter is demonstrated in this chapter, the method can be considered adequate for the purposes of assessing the effects of CPSO on application performance. Therefore, the method delivers the second contribution of the dissertation research listed in Chapter I.
The remaining research goal to be discussed in the subsequent sections of this dissertation is the development and evaluation of the QN-based analytical models of CPSO performance effects. The experiments designed to gather the data for the models are discussed in the following sections of this chapter.

5.4 The experimental work on characterization of the networking subsystem

This section discusses the experiments performed in order to characterize the networking subsystem in terms of its throughput and response time (latency), as well as in terms of the protocol stack processing demands. Additionally, a set of experiments is performed in order to measure the system component processing capacities. These measurements are used in Chapter VI in conjunction with the analytical models for evaluation of the CPSO performance effects.

5.4.1 Component processing capacity measurements

The communication protocol processing service demands can be estimated by taking measurements of the component processing capacities and using them in conjunction with the protocol stack workload model discussed in detail in Chapter VI. For instance, the measurements of the host CPU checksumming throughput and the CPU-memory data copy throughput can be used in order to estimate the per-byte protocol processing demands for the CPU-memory service center. The NIC-to-memory and the NIC-to-fabric sustained
throughput measurements can be used to estimate the service demands for the NIC service center.

The CPU and memory subsystem parameters can be measured using a special-purpose micro-benchmark developed by the author, that performs MTU-size memory-to-memory copies and TCP checksum computation. The time measurements are performed by reading the CPU timestamp register in an inline assembly macro. The measurement resolution is equal to the CPU clock rate resolution of approximately 0.5 nanosecond, and the measurement overhead of less than 40 nanoseconds. This allows for a sufficiently non-intrusive time measurements.

The latencies of one-way memory copy, as well as latencies of inline and partial checksum calculation for various data sizes are presented in Figures 5.9 and 5.10. Significant changes of the measured quantities can be observed when data size exceeds the size of L1 cache. The sustained (out of the core) figures are measured for large data sizes.

The checksum calculation latencies are measured by benchmarking the Linux TCP stack assembly routines that perform the partial and inline checksumming. The routines are placed in the framework of a simple micro-benchmark program and are compiled with the (relevant) compiler options used by Linux when building TCP modules. The checksumming latency is measured with the use of the lightweight timing routines mentioned earlier. The two series of experiments in Figure 5.10 represent the checksum calculation inline with memory copy (series “inline Xsum”), as well as the checksum calculation with no memory copy (series “partial Xsum”).
Figure 5.9 Memory-to-memory copy latency

Figure 5.10 TCP checksum calculation latency
Although the experiments were performed for a large range of message sizes, the following quantities are of specific interest: the memory-to-memory copy latencies for transferring MTU-size data, as well as the sustained memory subsystem bandwidth. The latter is necessary to estimate the aggregate memory subsystem bandwidth available for both CPU and NIC.

The presented measurements allow one to estimate the per-byte protocol stack processing demands, as the checksum calculation and memory copy latencies are proportional to the data size. As discussed in Chapters II and VI, in addition to the per-byte service demands, the per-packet, and per-message service demands are also needed in order to correctly model the protocol service demands. These demands can be estimated by way of taking measurements of the average times needed in order to perform the per-packet and per-message processing. A conventional approach to obtaining these measurements is to use the communication stack processing traces.

However, the stack processing trace analysis is quite technically challenging, since most of the stack processing occurs in the kernel. The open source tools for Linux kernel tracing such as Linux Trace Toolkit (LTT) discussed in [104] are available, but a substantial development effort for the stack source code instrumentation is needed in order to use LTT for the stack tracing. Such an effort could not be accommodated within the schedule of this research work. Instead, the general-purpose kernel event profiling that could be done with LTT is used for general “sanity check” of the models, as it provides estimates for the duration of the system calls, interrupt service routines, etc..
Therefore, another approach to estimating the CPU-memory service demands by the communication protocol processing activities is used. The CPU utilization $U_{cpu}$ and the system throughput $X_{pkt}$ measured while running MpiBench communication benchmark discussed in Section 5.4.2 are used in the following expression for the sought service demands $D_{cpu}$:

$$D_{cpu} = \frac{U_{cpu}}{X_{pkt}}.$$

This approach allows one to avoid having to obtain the traces of the protocol stack processing and to measure the absolute CPU time spent on various stages of the stack processing. A disadvantage of this approach is the potential loss of precision of the protocol stack processing presentation. The approach is based on the assumption that the measured CPU utilization accounts for the per-byte, per-packet, and per-message overheads.

The NIC processing capacity is estimated based on the motherboard specifications [97]. According to the specifications, the Gigabit Ethernet controller is connected to the memory controller hub through a dedicated 1Gbit/s point-to-point connection. The raw Gigabit Ethernet throughput is also 1Gbit/s. The quality of the CAT-5e copper connection cables is verified with the traffic generator and protocol analyzer, and it is found to be capable of carrying 1Gbit/s traffic.

### 5.4.2 Networking subsystem throughput, response time and $D_{accs}$

The measurements of the throughput and response times of the node’s networking subsystem are performed with MpiBench, a special-purpose communication micro-
benchmark developed by the author. The benchmark is inspired by NetPIPE [90], a public
domain networking benchmark.

```fortran
! ping-pong-style experiment ; streaming experiment (large N)
if (sender) then
  ts=time();
  for i in (0, N-1) do
    send(data)
    recv(data)
  done
  t=time()-ts;
  tcomm=t/(2^N);
else ; receiver
  for i in (0, N-1)) do
    recv(data)
    send(data);
  done
endif
```

Figure 5.11 Pseudo-code of the MpiBench micro-benchmark

NetPIPE was used in the early stages of experimentation, but was later abandoned in
favor of MpiBench. The main reason for developing MpiBench was the complexity of
modification of NetPIPE code in the course of message passing experiments. NetPIPE
was designed to support a wide range of data transport options; this resulted in additional
complexity of the benchmark. At some point, the time spent for debugging the benchmark
after modifying the code to meet the author’s needs outweighed the benefits of the code
reuse. The MpiBench pseudo-code is presented in Figure 5.11.

The developed benchmark allows one to measure the throughput and latency of point-
to-point streaming and ping-pong-style data transfers. During the MpiBench runs, the
system CPU utilization and the NIC interrupt rate are logged with the use of the Linux
`sar` utility. The logging activity itself does not result in significant system resource con-
sumption, and is measured to incur 1% or less CPU utilization. The measurement was performed in a separate experiment; the logging overhead was measured by running two instances of `sar`, one logging the overhead of the other.

5.4.2.1 Description of the experiments

The goal of the experiments is to measure steady-state networking subsystem throughput, response time (latency), as well as CPU utilization and NIC interrupt rates, in a given CPSO scenario for a given message size. These measurements, coupled with an approximation of the protocol stack processing in a given CPSO scenario, allow one to estimate the service demands by the protocol stack. Given the service demands and data transfer parameters, one can estimate the system component utilization, component throughput, and component response time, as well as the networking subsystem response time and throughput. These estimates can be compared with the measured component utilization and the measured networking subsystem response time. The results of the comparison can be used in order to validate the approximation of the stack processing.

The measurements of the networking subsystem response times for a given CPSO scenario are used in order to calculate the $D_{accs}$ attributed to this scenario as $D_{accs} = \frac{R_{base} - R_{pso}}{R_{base}} = 1 - \frac{R_{pso}}{R_{base}}$ where $R_{pso}$ and $R_{base}$ are the response times (latencies) for the CPSO scenario under consideration and the baseline scenario. These values are used for the model validation later in Chapter VI.
The experiments are organized as follows. For each CPSO scenario, a series of the benchmark runs is performed for exponentially increasing (power of two) message sizes starting with MTU size less 40 bytes (the sufficient space for regular IP and TCP headers) up to a few megabytes. For each message size, the benchmark is run for about 15 seconds in order to achieve quasi-steady system state. The throughput and response times for each 15 second run are averaged and recorded. The average CPU utilization and interrupt generation rate are logged. The experiments are performed for two MTU sizes - 1500 and 7500 bytes. The results of the experiments are summarized below.

Please note that the “megabyte per second” quantities on the figures in this chapter, as well as in the rest of the text, actually denote “million bytes per second”, and not the corresponding power-of-two numbers. While stretching the terms, this notation, in the author’s opinion, is more convenient for calculations, and it is harmless, as long as all the measurements are recorded consistently in this fashion.

5.4.2.2 Experimental results

The experimental data were gathered for twelve different networking subsystem configurations. The following parameters define the system configuration space: number of CPUs supported (uniprocessor or dual-CPU SMP), MTU size (1500 or 7500), transmit/receive side checksums on/off (just two states - all checksums on or all checksums off), and data copy avoidance (zero-copy) optimization on/off. As mentioned earlier in Chapter IV, the
zero-copy optimization implies checksums on, hence the twelve configurations instead of sixteen possible with four binary parameters.

The bulk of the experiments were performed in order to evaluate the system’s balance and in order to understand which networking subsystem configurations have more significant impact on the networking performance. The amount of the experimental data is far too large to be included in full in this dissertation. A summary of the obtained data is presented below. The general trends and dependencies observed with the complete data set can also be seen in Figures 5.12 - 5.15.

Several interesting insights were obtained during the experimentation. It was observed that MTU size has significant effect on the behavior of the networking subsystem not only in terms of its performance, but also in terms of the impact by the other networking subsystem optimizations that are utilized in the configuration with a certain MTU size. For 1500 bytes MTU, there is a clear difference between the uniprocessor and SMP configurations in terms of the communication performance. SMP nodes achieve measurably higher throughput of about 120 Mbytes/sec, while uniprocessor nodes top out at about 85 Mbytes/sec. The transfer latencies are commensurately smaller for SMP nodes, as compared to the otherwise identical uniprocessor configurations.

On the contrary, for 7500 byte MTU, the SMP versus uniprocessor cluster node configurations do not have a noticeable effect on the achievable networking performance. Both achieve about 120 Mbytes/sec (million bytes per second) maximum throughput, and both have identical latencies.
The MTU size also has a profound effect on the protocol stack processing demands and therefore on the incurred system resource utilization. This effect is in line with the achievable communication performance. Specifically, for 1500 byte MTU, CPU utilization and interrupt rates are commensurately higher for the SMP configurations, as compared to the uniprocessor configurations. But for 7500 byte MTU, no significant differences in CPU utilization and interrupt dates is observed between the SMP and uniprocessor configurations.

The checksum offload optimization does not have significant impact on networking performance and system component utilization across MTU sizes and node configuration. The transmit side checksum offload is not expected to result in measurable performance/utilization differences because of the inline checksumming optimization mentioned earlier. However, no measurable effect is observed with the receive side checksum either. Based on these observations, the checksum offload CPSO scenario is not considered as a separate scenario, but as a component of other CPSO scenarios.

The zero-copy optimization results in an overall improvement of the CPU utilization and in networking performance improvement for large message sizes in case of both 1500 and 7500 byte MTU sizes. However, these effects were not as pronounced as the effects of the large MTU size optimization.

The observed effects of the CPSO scenarios can be explained as follows. The per-packet processing discussed in Chapter VI appears to be the most significant component of the CPU overhead under heavy networking loads. Among other components, this process-
ing overhead includes NIC interrupt processing. The per-byte overheads that are reduced by the checksum offload and data copy avoidance CPSO scenarios appear to constitute a smaller portion of the node’s CPU overhead.

Additionally, it is clear that the cluster nodes’ CPU/memory subsystem is not being utilized as heavily as the networking subsystem. Therefore, the fractional improvements that result in further unloading the CPU/memory subsystem under the checksum offload and data copy avoidance CPSO scenarios do not have significant effects on the networking performance, or on the system component utilization.

As an example, consider the following approximate breakdown of the message passing latency. Out of the 837 microseconds of one-way transfer latency for 59 Kbyte message under 1500 byte MTU uniprocessor configuration with checksum offload off, the CPU memory-to-memory copy and checksum calculation on transmit and receive side takes less than 12 microseconds at the conservative one-way sustained rate of 1.5 Gbyte/sec for the checksum calculation latencies in Figures 5.10. In the meantime, CPU utilization is around 70%, which means that the CPU is busy for about 585 microseconds out of 837 performing the protocol stack processing other than memory copies and checksums (presumably, performing the per-packet processing and delivering the data across the wire). Clearly, the per-byte processing constitutes an insignificant fraction of the total processing time. Another observation that can be made based on this example is that the 1500 byte MTU size is not a good choice for Gigabit Ethernet networks.
There are several architectural reasons for the observed system behavior. The fast CPU and the capable memory subsystem of the cluster nodes in question make the networking subsystem a clear bottleneck for all the 7500 MTU configuration with the utilization of the networking subsystem about 90%. Additional networking capacity such as more network links bonded together or a faster (fibre optic) networking medium would shift the component service rate balance and make the effects of the checksum offload and zero-copy optimization more noticeable.

Based on the analysis of the large experimental data set, the following CPSO scenarios were selected for further experimentation:

- the baseline scenario — uniprocessor, 1500 bytes MTU, no checksum offload, no copy avoidance — abbreviated notation: ppuxoff1500;
- large MTU size scenario — uniprocessor, 7500 bytes MTU, no checksum offload, no copy avoidance — abbreviated notation: ppuxoff7500;
- transmit side data copy avoidance — uniprocessor, 7500 bytes MTU, checksums offloaded, data copy avoidance — abbreviated notation: ppupzctcp7500;
- complete CPSO — dual SMP, 7500 bytes MTU, checksums offloaded, no data copy avoidance — abbreviated notation: ppxon7500;
- complete CPSO with data copy avoidance — dual SMP, 7500 bytes MTU, checksums offloaded, data copy avoidance on — abbreviated notation: ppzctcp7500.

The bandwidth, latency, CPU utilization and interrupt rates for these scenarios measured on the transmit side are shown in Figures 5.12, 5.13, 5.14, and 5.15. The transfer size ranges were selected based on the results of the measurements of the degree of overlapping $D_o$ discussed earlier.

It can be observed that the experimental results presented in these figures follow the trends characteristic of the larger experimental data set, as discussed earlier in this section.
Figure 5.12 CPU utilization under MpiBench test

Figure 5.13 NIC interrupt rates under MpiBench test
Figure 5.14 Communication bandwidth for the CPSO scenarios of interest

Figure 5.15 Communication latency for the CPSO scenarios of interest
One can notice that the CPU utilization patterns in Figure 5.12 follow the NIC interrupt rate dependencies in Figure 5.13. The approximate 50% drop of the CPU utilization for the SMP scenarios reflects the fact that the CPU utilization in the figure represents the average per-processor values. It can also be observed that the incremental differences in CPU utilization attributable to the checksum offload and zero-copy optimizations are fairly small as compared to the effects of the large MTU size optimization. Finally, it can be observed that the sustained communication performance for uniprocessor and SMP scenarios with large MTUs is quite similar, although it is achieved at the expense of different per-processor utilization.

![Figure 5.16 Degree of acceleration $D_{\text{acc}}$ measured in MpiBench](image)
The measurements of the networking subsystem response times are used to calculate the degree of acceleration $D_{accs}$ achieved by the CPSO scenarios as compared to the baseline scenario. The calculated quantities, as well as their comparison with the corresponding values presented in Figure 5.3, are presented in Figures 5.16 and 5.17. Figure 5.17 show the deltas as a fraction of the value of the metrics.

It can be seen that the values of $D_{accs}$ measured in the runs of DoBench (Figure 5.3) and MpiBench (Figure 5.16) are in reasonable agreement. The difference between the quantities of $D_{accs}$ obtained in these experiments can be attributed to the interaction between the communication and computation activities, other than their concurrent processing, as well as to experimental errors, notably in the case of 59Kbyte messages. It should be mentioned that even though the fractional value of $D_{accs}$ deviation seems large (up to 20%), the
absolute value is 0.03 or less, and the corresponding difference of the time measurements reflected in the coefficient of variation of $S_{comm}$ (3.8) is 3.7\% or less.

5.5 Summary

This chapter presented the experimental work performed in the course of the dissertation research. The benchmarks designed to gather the sought experimental data were described, and the experimental results were discussed.

The experiments were organized in two groups. The first group of experiments was performed in order to validate the dissertation thesis. The second group of experiments was performed in order to obtain the data used as input for the models.

Based on the experimental data and the dissertation thesis validation procedure, the thesis was validated. The primary research goal of this dissertation was achieved, and the first contribution of the dissertation research was delivered. The remaining research goal to be discussed in the subsequent sections of this dissertation is development and evaluation of the QN-based analytical models of CPSO performance effects.
CHAPTER VI

MODELS FOR PERFORMANCE ANALYSIS OF CPSO SCENARIOS

This chapter discusses the details of the analytical models for the cluster node networking subsystems briefly described in Chapter IV. The deliverables of the modeling process are the system-level metrics $D_o$ and $D_{acs}$ defined and discussed in detail in Chapter IV. The input parameters for the models, such as system component processing capacity, the protocol stack service demands, the networking subsystem response time (latency), and the networking subsystem throughput, are obtained in the set of experiments described in Section 5.4.2.

These models are based on the approximation of the MPI/TCP/IP communication protocol stack processing discussed in Section 6.1. The Open QN, Closed QN, and the Deterministic QN-based models are discussed in Section 6.2. The models are evaluated as to their agreement with the experimental data presented in Chapter V. The chapter is concluded with the discussion of the modeling results and their use for the analysis of the CPSO performance effects.
6.1 Approximation of the communication protocol stack processing

The processing of the MPI/TCP/IP communication protocol stack is a complex, data-dependent system activity. This processing can be considered on different levels of detail depending on the goals of the modeling process and the required precision. In other words, the processing may not need to be presented in all its complexity. Instead, it is important to capture the aspects of the processing that account for the significant portion of the system resource demands by the stack. Therefore, the discussion below is structured as follows. The general case of protocol stack processing is discussed first. The discussion of how this general case applies to the system under consideration is presented next.

Following and extending the approach taken in [19, 20], let us consider the protocol stack processing overheads to include the per-byte, per-packet, and per-message processing overheads. The per-byte overhead includes the costs of data movement and checksum computation that are incurred for every byte of data in transfer. The per-packet overhead includes packet buffer management, timer management, TCP, IP, and Ethernet address lookups, finalization of the checksum computation, and interrupt processing. The per-message overhead includes system call overhead, middleware overheads for asynchronous message completion notification and for the rendezvous protocols for receiver side buffer allocation for long messages [26]. The sum total of these overheads represents the protocol stack workload model that is used in conjunction with the analytical models discussed earlier in Chapter IV in order to derive the system metrics of interest $D_o$, $D_{acc}$, and $S_{comm}$. 
6.1.1 The general workload model

The initial approximation of the protocol stack processing for the baseline system and the CPSO scenarios discussed in Chapter IV is as follows. On the transmit side, some amount of processing is performed in order to lookup the TCP connection that corresponds to the MPI destination rank, the message send operation is posted, and context switch to the communication thread is performed. The communication thread makes a send system call; as a result, the data is copied from the user-space buffers to the system socket buffers with the partial checksum calculation done in-line, then the stack performs a certain amount of per-packet computation as described above, then the packets are brought from the main memory onto the NIC by the NIC DMA engine, and are sent over the fabric. The transmit completion interrupt is generated per a certain number of packets depending on the NIC driver configuration and the rate of the packets arrival to the NIC for transmission over the network.

On the receive side, a packet arrives from the network into the NIC, and it is placed in the main memory by the NIC DMA engine in a pre-allocated buffer. An interrupt is generated per a certain number of arrived packets, the host CPU validates the packet checksums, performs a certain amount of per-packet processing as discussed above, copies the data to the socket system buffers, and then, to the user-space buffers. The MPI communication thread waiting in the receive system call is unblocked, and it optionally copies the data to the destination user buffer.
In summary, the transmit-side TCP processing involves moving the message data over the memory bus three times (two times during the memory-to-memory copy and one time during the DMA to the NIC), performing the partial checksum calculation in-line with the memory-to-memory copy, per-packet processing, and servicing an interrupt possibly amortized over several packets. The worst case receive-side processing involves moving the message data over the memory bus six times (NIC DMA, checksumming, and two memory-to-memory copies), some amount of per-packet computation, and servicing an interrupt possibly amortized over several packets.

The large MTU size scenario reduces the number of Ethernet packets needed in order to pass a given amount of user data. Therefore, the aggregate per-packet processing overhead is expected to be reduced. As a result, the CPU utilization is expected to decrease.

The data copy avoidance scenario benefits the sender side by allowing it to avoid the initial memory copy to the system buffers. However, some additional processing for mapping the user buffers in kernel context is incurred instead. The zero-copy kernel patch that implements the optimization includes an adjustable threshold that defines the minimal data size that is processed by the zero-copy code path. The smaller messages are processed by the regular code path with the data copy. As mentioned earlier, this optimization also assumes the checksum offload is enabled.

The complete CPSO and complete CPSO with data copy avoidance scenarios are functionally identical to the large MTU size with checksum offload enabled and the data copy avoidance scenarios respectively. However, since the system is configured in the dual
SMP mode, the system component utilization and response times are significantly different. Therefore, the scenarios are considered separately.

The per-message overhead incurred in the MPI middleware layer is attributable to each of the above scenarios to an equal degree. It includes the inter-thread context switch to/from the MPI communication progress thread, the per-message system call overhead, and the rendezvous protocol overhead for long messages equivalent to the overhead of two short (less than the minimal Gigabit Ethernet frame size of 512 bytes) TCP messages exchanged between the sender and the receiver of the data message. Additionally, since MPI introduces an extra level of message de-multiplexing based on MPI communicators, another data copy on the receive side in the message de-multiplexor may be necessary.

6.1.2 Customization of the general workload model

The testbed system under consideration generally adheres to the protocol stack processing description provided in Section 6.1. However, some specifics in the system’s behavior allow for simplification of the general protocol stack processing case. As discussed further in Chapter V, it turns out that the per-packet processing overhead accounts for the major portion of the protocol stack service demands for the system under consideration. The primary reasons for that are the fast CPUs and the capable memory subsystem with abundant throughput that meet the demands of protocol stack processing at Gigabit speeds without saturation.
Therefore, the models discussed in the following sections of this chapter are focused on the packet processing by the protocol stack. The experimental data obtained in the ping-pong-style communication experiments are used as input parameters for the modeling process, as well as for the model verification. The primary reason behind this choice is that the synthetic application-level benchmark DoBench used for derivation of the $D_o$ and $D_{accs}$ metrics is structured to allow for the maximum overlapping of communication and computation. In order to achieve this goal, the benchmark assures that the per-message compute time exceeds the one-way communication latency measured in the first phase of the benchmark as discussed in Chapter V. Therefore, there is no overlap in processing of the consecutive messages.

Another important feature of the models under consideration is that they offer the per-packet system performance metrics for the case of an otherwise unloaded system. That is, the models assume that the communication processing is the only system activity in progress. This observation is used later in the discussion of approaches to increasing the precision of the modeling.

Based on the obtained per-packet component utilization and component response times, the per-message response times $R$, and the system metrics $D_o$, $D_{accs}$ and $S_{comm}$ are derived. When deriving the metrics, it is important to accurately account for the concurrency exhibited by the system components during the stack processing. The fidelity of the representation of the packet processing concurrency determines to a significant degree how well the modeling results fit the experimental data.
6.2 QN-based models for the analysis of CPSO performance effects

The analytical models discussed in this chapter are based on the approximation of the communication protocol stack processing described in Section 6.1. The models represent the case of a quasi-steady processing of network packets flowing through the cluster node’s compute and networking subsystems.

The queueing network in question consists of the following two service centers: the CPU-memory subsystem, and the NIC. A more detailed presentation of the system with more service centers that represent system components such as the memory subsystem and I/O subsystem with the corresponding busses and bridges is difficult to achieve for the following reason. QN-based models have difficulties modeling the multiple resource possession that occurs in models with finer granularity of component representation.

For instance, if the CPU and memory subsystem are presented as two separate components, then in order to accurately model memory-to-memory copy, both of these components need to be considered busy during the memory copy service request. This would be equivalent to having one job present in two queues at two service centers simultaneously, and, as discussed in [43], the performance analysis approaches used in this work would be inapplicable.

This problem can be avoided if the two components are merged. For the test bed system, this approach is also justified by the fact that the memory subsystem is loaded lightly, as compared to the CPU and NIC system components. If it is necessary to obtain
memory subsystem performance metrics, a separate model (that does not consider CPU component) can be used.

The compute service center is modeled as a finite capacity center with exponentially distributed service time and a per-packet service demand $D_{cpu} = \frac{U_{cpu}}{X_{pkt}}$. $U_{cpu}$ is the measured CPU utilization, and $X_{pkt}$ is the system throughput measured in packets per second. As discussed in Chapter IV, this approach allows one to avoid having to obtain the traces of the protocol stack processing and to measure the absolute CPU time spent on various stages of the stack processing. A disadvantage of this approach is the potential loss of precision of the protocol stack processing presentation. This approach is based on the assumption that the measured CPU utilization accounts for the per-byte, per-packet, and per-message overheads. This assumption is discussed later in this section.

An alternative approach to modeling the compute service center service demands is to measure the average times it takes to perform the per-byte, per-packet, and per-message processing. The processing times can be estimated with the use of the communication stack processing traces. This approach may offer better precision of the service demands estimates.

The dual CPU case is treated in a similar fashion, and the service demands per CPU are estimated as described above. The NIC is also modeled as a finite capacity single server with exponentially distributed service time $S_{nic}$ with mean $\frac{N_{pkt}}{X_{nic}}$; $N_{pkt}$ is the packet size, and $X_{nic}$ is the network link throughput in bytes per second (1 Gbit/sec).
The jobs (packets) enter the system and visit the compute and NIC centers once before leaving the system. As a result of modeling, the component response times $R_i$ are calculated. Based on the component metrics $R_i$, and the number of packets $n$ that constitute a message, the message response time $R$ and the system metric $D_o$ can be derived. The message response time is used for comparison with the measured message latency in the process of the model validation. The value of $D_o$ is compared to the one derived from the measurements as well. The value of $D_{accs}$ is calculated for each CPSO scenario and the corresponding message sizes relative to the baseline CPSO scenario. The $D_o$ and $D_{accs}$ are used in calculations of $S_{comm}$ that is subsequently used for comparison with the $S_{comm}$ values derived from the measurements.

The following pipelined packet processing approximation is used in order to calculate the per-message metrics $R$ and $D_o$.

\[
R = R_{cpu} + R_{nic} + \frac{n_{pkt} - 1}{X_{pkt}} \quad (6.1)
\]

\[
D_o = \frac{(n_{seg} - 1)}{n_{seg}} \frac{1}{1 - \frac{n_{pkt}D_{cpu} + R_{cpu} + R_{nic} + D_{msg}}{R}} \quad (6.2)
\]

The processing of each packet (the total of $n_{pkt}$ packets constitute a message) includes the processing by the compute subsystem, that takes $R_{cpu}$ time, and the processing by the networking subsystem, that takes $R_{nic}$ time. If packets are processed by the compute and networking subsystems in a pipelined fashion with throughput $X_{pkt}$ and latency $R_{cpu} + R_{nic}$, the time to process $n_{pkt}$ packets is equal to $R$ in (6.1).

The time when the compute subsystem is not busy is $T_{free} = (n_{seg} - 1)(R - n_{pkt}D_{cpu} - R_{cpu} - R_{nic} - D_{msg})$, where $n_{seg}$ is the total number of segments (messages) in the over-
lapping experiment, $D_{cpu}$ is the CPU time needed to process one packet, $R_{cpu} + R_{nic}$ is the initial latency, and $D_{msg}$ is the per-message overhead that includes the rendezvous protocol overhead and the thread switching overhead measured in seconds. The rendezvous negotiation overhead is equal to 125 microseconds - the time to exchange two small messages, and the thread switching overhead is equal to 17 microseconds as measured in a separate micro-benchmark.

Based on these observations, one can obtain the ratio of the time the compute system is available for the application-specific computation to the communication time as

$$\frac{T_{free}}{n_{seg}R} = \frac{n_{seg}-1}{n_{seg}}(1 - \frac{n_{pkt}D_{cpu}+R_{cpu}+R_{nic}+D_{msg}}{R})$$

As discussed in Chapter IV, this fraction can be used in order to estimate $D_o$. The factor of $\frac{n_{seg}-1}{n_{seg}}$ in (6.2) accounts for the fact that only $n_{seg} - 1$ out of $n_{seg}$ segments (messages) are overlapped with computation in DoBench, as discussed in Chapter V.

The strengths of the QN-based models include relative simplicity and computational efficiency. The models are fairly intuitive, and they can be used for initial estimates of the system metrics of interest. The model weaknesses include the assumption of exponential distributions of the component service time and job arrival times. These assumptions usually lead to over-estimation of the component response times and queue lengths that can affect the precision of the estimates for $D_o$ and $D_{accs}$. 

6.2.1 The Open QN model

The Open QN model represents the case of sender-side quasi-steady packet processing. The input parameters for the model are the system throughput, the service center processing capacity, and the job service demands per component. The output metrics of the model are the component utilization, the component response times, and the system response time per job.

The system throughput determines the rate of the end-to-end job flow through the system. The component processing capacities and the job service demands determine the service rates and the response times by the components. The per-job system response time, which represents the packet latency, is calculated based on the component response times and the number of visits of the component by the job.

The Open QN model of the stack processing is implemented by the author in a C program that accepts the input parameters in a tabular form arranged by the CPSO scenario and message size. The program calculates the output parameters using the following formulae from [43]:

\[ U_i = XS_iV_i, \quad X_i = XV_i, \quad R_i = \frac{S_i}{1 - U_i}, \quad i \in \{1, \ldots, M\}. \]

\( X \) is the system throughput, \( S_i \) - \( i \)th component service time per job visit, \( V_i \) - number of visits to the \( i \)th service center, \( M \) - number of devices in the system, \( R_i \) - response time of the \( i \)th device, and \( U_i \) - \( i \)th device utilization. The set of input parameters for the model is
gathered for each CPSO scenario and message size of interest based on the results of the experiments discussed in Chapter V.

![Figure 6.1 Open QN model: estimated $D_o$ vs. measured $D_o$](image)

The system metrics obtained with the use of the Open QN model are compared to the metrics obtained experimentally in Figures 6.1, 6.2, 6.4, and 6.5. For the reference, the corresponding metrics derived from the measurements are presented in Figures 5.2, 5.3, and 5.6. The legends in the figures follow a common naming convention for the CPSO scenarios: the $ms_-$ prefix denotes the measured quantities, and the scenario names without the prefix denote the quantities estimated by the model.

The model offers reasonable estimates of the $D_o$ for the some CPSO scenarios, such as data copy avoidance and complete CPSO scenarios. The $D_o$ for the complete CPSO
Figure 6.2 Open QN model: estimated $D_o$ vs. measured $D_o$, continued

with data copy avoidance scenario is generally underestimated, but the qualitative picture predicted by the model is reasonably close to the experimental results. However, the $D_o$ for the baseline scenario is significantly overestimated. For this scenario, the maximum of $D_o$ is predicted with about 22% accuracy, but the nature of the dependence of $D_o$ on the message size is not predicted correctly. Similarly, the values of $D_o$ for the large MTU size scenarios are overestimated for the small message sizes.

The model closely estimates the optimal message sizes for the two SMP scenarios and the data copy avoidance uniprocessor scenario. The model also predicts the maximum $D_o$ for the complete CPSO with data copy avoidance scenario with about 22% accuracy, and the maximal $D_o$ value of the complete CPSO scenario with about 12% accuracy. The
maxima of the data copy avoidance and large MTU size scenarios are predicted with 9% and 6% accuracy.

The model allows one to clearly distinguish between the uniprocessor and SMP scenarios, and it correctly indicates that the latter exhibits higher $D_o$. However, the model does not distinguish well between the scenarios within the uniprocessor and SMP groups.

The model also offers reasonably good estimates of the per-message latency $R$ (Figure 6.3) for all the scenarios, as compared to the values of $R$ measured in the MpiBench experiments. The values of $R$ are slightly overestimated. All estimates are within 12.7% from the measured quantities, with the majority of the estimates better than 8%.
Figure 6.4 Open QN model: estimated $D_{accs}$ vs. measured $D_{accs}$

Figure 6.5 Open QN model: estimated $S_{comm}$ vs. measured $S_{comm}$
The values of the $D_{accs}$ metric are predicted reasonably well based on the good estimates for $R$. However, the values of the $S_{comm}$ metric are generally significantly underestimated. The overestimated $D_o$ for the baseline scenario has affected the precision of the $S_{comm}$ estimates to a significant degree. The maximum of $S_{comm}$ for the complete CPSO with data copy avoidance scenario, message size 954,872 bytes, is predicted, but, as mentioned earlier, its value is underestimated.

In summary, the Open QN model can be used to predict message latencies and, consequently, the $D_{accs}$ metric, with reasonable precision. The model also predicts the values of the maxima of $D_o$ for the CPSO scenarios of interest, with better than 13% precision for 3 out of 4 CPSO scenarios. Additionally, the model offers a reasonable qualitative picture of $D_o$ vs. message size for the SMP and the data copy avoidance scenarios. The model also allows one to make a correct conclusion that the SMP CPSO scenarios lead to shorter application execution time as compared to the uniprocessor scenarios.

However, the model does not clearly indicate that the complete CPSO with data copy avoidance is a superior CPSO scenario as compared to the complete CPSO scenario. The model also mispredicts the nature of dependence of $D_o$ on the message size for the baseline and the large MTU size scenario, and it overestimates the $D_o$ for these scenarios for the majority of the message sizes. As a result, $S_{comm}$ is significantly underestimated.

Therefore, the model can be used for obtaining a general idea of the effects of the CPSO scenarios of interest in terms of the maximal achievable $D_o$ and $S_{comm}$, but it cannot be used for the detailed analysis of the behavior of the networking subsystem for various
message sizes. In order to perform such analysis, the modeling precision needs to be improved.

6.2.2 The Closed QN model

The input parameters for the Closed QN model include the number of jobs (packets for a given message size) in the system, the component processing capacities, and the component service demands. The model outputs include system throughput, component utilization, component response times, and system response time.

The Closed QN model is implemented by the author in a C program that accepts tabular input arranged by the CPSO scenario and message size, similar to the Open QN model. The output metrics are calculated using the Mean Value Analysis (MVA) algorithm [43]. This is an iterative procedure that calculates the metrics for an increasing number of jobs. The iteration is stopped when the number of jobs of interest is achieved. Figure 6.6 illustrates the algorithm adopted from [43].

The estimated values of $R$ along with the measured values of $R$ are shown in Figure 6.7. The system metrics obtained with the use of the Closed QN model are compared to the values obtained experimentally in Figures 6.8, 6.9, and 6.10. As earlier, the legends in the figures follow a common naming convention for the CPSO scenarios, with the ms_ prefix denoting the measured quantities.

In the process of experimentation with the Closed QN model, it was determined that the model overestimated message latency $R$ to a more significant degree as compared to
; inputs
; \( N \) - number of jobs
; \( M \) - number of service centers
; \( S_i \) - service time per visit to the \( i \)-th center
; \( V_i \) - number of visits to the \( i \)-th center

; the iterative part
for \( n \) in \((1, N)\) do
  \( R = 0 \)
  ; response time based on queue lengths
  for \( i \) in \((1, M)\) do
    \( R_i = S_i (1 + Q_i) \)
    \( R += R_i \cdot V_i \)
  done

; outputs
; \( X \) - system throughput
; \( Q_i \) - average number of jobs at the \( i \)-th center
; \( R_i \) - response time of the \( i \)-th center
; \( R \) - system response time
; \( U_i \) - utilization of the \( i \)-th center

; system throughput for \( n \) jobs
\( X = n / R \)

; re-calculated based on the throughput for \( i \) in \((1, M)\) do
  \( Q_i = X / V_i + R_i \)
  done

; algorithm start
; initialization
for \( i \) in \((1, M)\) do
  \( Q_i = 0 \)
  done

; device throughput and utilization
\( X_i = X / V_i \)
\( U_i = X / S_i + V_i \)

Figure 6.6 The MVA algorithm pseudo code

Figure 6.7 Per-message \( R \) in the Closed QN model vs. measured \( R \)
the Open QN model. The error grew with the message size, and it lied between 3% and 39% of the measured values of $R$. Given these imprecise estimates of $R$, it was desirable to exclude $R$, $R_{cpu}$, and $R_{nic}$ from expression (6.2). The following expression for $D_o$ was chosen instead:

$$D_o = \frac{n_{seg} - 1}{n_{seg}}(1 - U_{cpu})$$  \hspace{1cm} (6.3)

As indicated earlier, $n_{seg}$ represents the number of segments used in the DoBench for the measurement of $D_o$. $U_{cpu}$ is the CPU utilization predicted by the model. $U_{cpu}$ closely approximates the values of $\frac{n_{pkt}D_{cpu}}{R}$ used in (6.2).

![Figure 6.8 Closed QN model: estimated $D_o$ vs. measured $D_o$](image)

The predictions of $D_o$ by the Closed QN model are as follows. The *baseline* scenario is still mispredicted. However, instead of overestimating $D_o$ for the majority of the mes-
sage sizes, $D_o$ is under-estimated for the case of two largest sizes, and it is close to the measurements for the rest of the message sizes. The values of $D_o$ for the large MTU size scenario are significantly overestimated for all but the two largest message sizes.

The predictions for other scenarios offer a reasonable qualitative picture of $D_o$ vs. message size. The maxima of the predicted $D_o$ are closer to the measured quantities as compared to the Open QN modes. However, the maxima are shifted toward the smaller message sizes.

Similarly, the predicted maxima of $S_{comm}$ are closer to the measured values, but they are shifted toward the smaller message sizes. The qualitative picture of $S_{comm}$ vs. message size is not as close to the experimental data as it is in case of the Open QN model.
Similar to the Open QN model, the Closed QN can be used for rough estimates of the achievable maxima of $D_o$ and $S_{comm}$ metrics for the majority of the CPSO scenarios of interest. The over-estimation of $R$ does not result in a significant distortion of the picture for $S_{comm}$. The choice of (6.3) for $D_o$ facilitated better predictions of $S_{comm}$. The model shows a clear difference between the uniprocessor and SMP scenarios, but it still does not distinguish between the two SMP scenarios, or between the two uniprocessor scenarios.

6.2.3 The Deterministic QN-based model

It was pointed out earlier that the assumptions about exponential distributions of the inter-arrival and service times in the QN models result in over-estimation of the component and system response times. In the case of the Open QN model, the formula for calculation
of $R_i$, $R_i = \frac{S_i}{U_i}$ is based on that assumption [43]. It is interesting to eliminate this assumption from the model and to observe how the precision of the modeling is affected.

One approach to accomplishing this is to assume deterministic service demands and job arrival times. In this case, the component response times become equal to the per-visit service demands, $R_i = S_i$. This assumption results in the deterministic lengths of job queues at the service centers not exceeding one job. The remaining expressions used in the model, $U_i = XS_iV_i$ and $X_i = XV_i$, reflect the Operational Laws [43], and therefore, are valid regardless of the types of the service demands and inter-arrival times distributions. The implementation of this model requires a small change to the Open QN program.

Figure 6.11 Deterministic QN-based model: estimated $R$ vs. measured $R$
Similar to the Open QN model, the Deterministic model uses expressions (6.1) and (6.2) for derivation of the per-message $R$ and $D_o$. The comparison of $R$ as predicted by the model and as measured in MpiBench is presented in Figure 6.11. The system metrics obtained with the use of the Deterministic Open QN model, along with the corresponding measured values, are shown in Figure 6.12, 6.13, 6.14, and 6.15.

The predictions of $R$ fit the experimental data reasonably well, as expected with the use of expression (6.1). The values of $R$ are predicted with better than 5% accuracy for the majority of the message sizes. However, $R$ is underestimated for 59672 byte message size for all the scenarios except the baseline; the error is between 7% and 13%. The estimates of $D_{accs}$ shown in Figure 6.14 demonstrate good agreement with the experimental data for all data sizes except 59672 bytes.

![Figure 6.12 Deterministic QN-based model: estimated $D_o$ vs. measured $D_o$]

Figure 6.12 Deterministic QN-based model: estimated $D_o$ vs. measured $D_o$
The predictions of $D_o$ are very similar to the predictions of the Open QN model. In fact, the qualitative picture of $D_o$ vs. message size is practically the same with the $D_o$ maxima achieved at the same message sizes. However, the error of predicting the $D_o$ maxima is less than the error of $D_o$ prediction by the Open QN model. The baseline and large MTU size scenarios are still mispredicted in a fashion similar to the Open QN model. Finally, the predictions of $S_{comm}$ also offer a reasonably good qualitative picture that reflects the measured dependencies. The values of $S_{comm}$ maxima are closer to the measured values, but are still significantly underestimated. The optimal message sizes are also close to the experimental results.

In summary, the Deterministic model offers better prediction of the response time and $D_{accs}$ for the majority of message sizes, and the predictions of $D_o$ and $S_{comm}$ are closer
Figure 6.14 Deterministic QN-based model: estimated $D_{\text{accs}}$ vs. measured $D_{\text{accs}}$

Figure 6.15 Deterministic QN-based model: estimated $S_{\text{comm}}$ vs. measured $S_{\text{comm}}$
to the experimental data. However, the numerical precision still does not approach the precision of the experiments, and it is still inappropriate for quantitative analysis of CPSO performance effects.

6.2.4 Analysis of the modeling results

Several potential sources of insufficient modeling precision can be considered. One can observe, that the models represent the processing of the communication protocol stack on an unloaded system. Therefore, the model does not account for the service demands imposed by the application-specific computations. It should be mentioned however, that this approach of assessing $D_o$ is consistent with the definition of $D_o$ as a least upper bound over the class of applications of interest as discussed in Chapter IV.

Nevertheless, the additional load on the CPU-memory service center would result in delays in the communication protocol stack processing. In the case of the SMP scenarios, such delays would not necessarily result in a decrease of $D_o$ because of the possibility of concurrent processing of communication and computation on two CPUs. However, in the case of the uniprocessor scenarios, the delays would in fact result in a decrease of $D_o$ with decreasing message sizes.

In order to verify this hypothesis, an additional CPU load was introduced in the Open QN model. In the process of experimentation with the model, it was determined that an increase of the average CPU center queue length by 0.55 resulted in a significant improvement of the qualitative agreement with the experimental data in case of the baseline
scenario as shown in Figure 6.16 while still underestimating $D_o$ for the largest segment size.

The precision of the $D_o$ predictions for the large MTU size scenario can also be significantly improved with the introduction of an additional CPU load. However, the increase of the average CPU center queue length needed for better convergence with the experimental data is different from the one used for the baseline scenario. Therefore, the additional CPU load cannot be treated as a model calibration factor. Some modification of the model that results in the desired increase of the CPU center queue lengths supported by the proper justification is needed in order to improve the precision of the models.

Figure 6.16 $D_o$ in the adjusted Open QN model vs. measured $D_o$. 
The overestimation of $D_o$ for the *baseline* scenario for the majority of the segment sizes has affected the precision of $S_{comm}$ estimates. Perhaps the choice of another CPSO scenario as a *baseline* scenario would not reveal this shortcoming of the models. In the meantime, it seems very instructive to study this set of scenarios as they are modeled with the QN-based models in order to gain a better understanding of the limitation of these models.

One conclusion that can be made based on the experiments with the Deterministic model is that the Open QN model does not overestimate the service center response times to a significant degree. The values of $R$ are estimated with reasonable precision by both models. The $D_o$ estimates offered by the Deterministic model are closer to the measured values for the *data copy avoidance* and the SMP scenarios, but the the *baseline* and *large MTU size* scenarios are still mispredicted.

Two possible sources of overestimation of $D_o$ are the system resource depletion, as well as contention for the shared resources, that are not explicitly accounted for in the models discussed in this chapter. As mentioned earlier, these types of system phenomena are difficult to model with the use of queueing models. This intrinsic limitation of the modeling formalism makes it difficult to achieve high modeling precision with the models discussed in this chapter.

This line of reasoning is further strengthened by the observation that all the models misrepresent $D_o$ in the case of the *baseline* CPSO scenario, while the rest of the scenarios are represented with better precision. The parameter that distinguishes the *baseline* sce-
nario from the rest is the small MTU size of 1500 bytes. This results in a significantly higher packet traffic than in the rest of the scenarios. The high packet traffic possibly triggers some of the system-level mechanisms that lead to blocking, which results in less CPU time available for the application-level processing, and therefore lower $D_o$.

Yet another possible source of the unsatisfactory quantitative precision of the models is the error in estimating the CPU service demands using the expression $D_{cpu} = \frac{U_{cpu}}{X_{pkt}}$ discussed in Section 6.2. Tracing of the TCP/IP protocol stack in Linux kernel could allow for better understanding of the associated CPU and memory bandwidth demands and for the more precise modeling of the whole process.

6.3 Summary

The QN-based analytical models for the communication protocol stack processing by a PC-based cluster node were discussed in this chapter. It was determined that the models offered reasonable qualitative agreement with the experimental results. The Open QN model offered the best fitting estimates of the response times $R$ and therefore the $D_{accs}$ metric. The Deterministic QN-based model offered the best fitting estimates for the $D_o$ and $S_{comm}$ system metrics.

The models did not offer sufficient precision for the quantitative analysis of the performance effects of the CPSO scenarios with high accuracy. The composite metric $S_{comm}$ that accounted for the combined effects of accelerating communication and overlapping of communication and computation was underestimated to a significant degree by all the
models. The overestimated $D_o$ for the baseline CPSO scenario had a significant effect on the precision of the $S_{comm}$ estimates.

Finally, it was determined that the predictions by the models allowed one to clearly distinguish between the baseline scenario, the uniprocessor scenarios, and the SMP CPSO scenarios based on the predicted values of $S_{comm}$. Specifically, the models allowed one to identify the complete offload with data copy avoidance and the complete offload CPSO scenarios that minimise the application execution time. The measured values of $S_{comm}$ for these scenarios also indicated that the scenarios were very close in terms of their performance effects. Therefore, the models could be considered an acceptable tool for the qualitative analysis of the CPSO scenarios.
CHAPTER VII

CONCLUSIONS

This chapter concludes the description of the research work discussed in this dissertation. The dissertation research concentrated on the analysis of the performance metrics for the description of CPSO effects on performance in parallel applications in clusters. The work drew a distinction between system-level and application-level phenomena under conditions of communication protocol stack offload that affected application performance.

A set of metrics for description of these phenomena was proposed. The degree of acceleration $D_{accs}$ was used in conjunction with the degree of overlapping $D_o$ defined in the earlier research described in [26] in order to derive the communication speedup $S_{comm}$ defined in (3.7) in Chapter III. The latter metric was used in order to quantify the system-level phenomena that are responsible for the acceleration of the communication protocol stack processing and for the concurrent processing of the communication protocol stack and application-specific computations.

The application latency hiding capacity $R_{app}$ metric was used in order to quantify the application’s capacity to take advantage of the opportunities for the concurrent processing of the application communication requests and the application-specific computation described by the system-level metric $D_o$. A method for the assessment of the effects of
the CPSO phenomenon on the application performance was formulated and discussed in Chapter III. The method uses $R_{app}$, $D_o$, and $D_{accs}$ in expression (3.12) in order to calculate the communication speedup for a given class of applications characterized by $R_{app}$ that can be attributed to a given CPSO characterized by $D_o$ and $D_{accs}$.

Three QN-based analytical models for the performance effects of CPSO scenarios were developed and evaluated in Chapter VI. The models were shown to be adequate for qualitative analysis of most of the CPSO scenarios of interest. However, the modeling precision was found to be insufficient for a quantitative analysis of CPSO scenarios with the exception of a few cases.

The thesis of this dissertation stated that the effects of communication protocol stack offload (CPSO) on application execution time could be attributed to the following two complementary sources. First, the application-specific computation might be executed concurrently with the asynchronous communication performed by the communication protocol stack offload engine. Second, the protocol stack processing could be accelerated or decelerated by the offload engine. It was also stated that these two types of performance effects could be quantified with the use of the degree of overlapping $D_o$ and degree of acceleration $D_{accs}$ metrics. The composite communication speedup metric $S_{comm}(D_o, D_{accs})$ could be used in order to quantify the combined effects of the protocol stack offload.

This thesis was validated empirically based on the results of the experiments discussed in Chapter V. The experimental procedure for measuring the metrics was defined, the experimental results were presented and analyzed. The thesis validation procedure was
discussed in Chapter IV. It was shown that the communication speedup $S_{comm}$ under conditions of CPSO defined in (3.7) could be quantitatively described with the use of metrics $D_o$ and $D_{accs}$ in expression (3.8) with sufficient precision.

The primary contributions of this dissertation work were formulated in Chapter I as follows. The metric $D_{accs}$ and $S_{comm}(D_o, D_{accs})$ were to be introduced in order to quantify the effects of optimizations in the networking subsystem on parallel performance in clusters. These metrics were defined in Chapter III, and they were shown to adequately describe CPSO performance effects in the process of the dissertation thesis validation in Chapter V.

A method for assessing the effects of CPSO scenarios on application performance was to be developed. The method was discussed in Chapter III; it used $R_{app}$, $D_o$, and $D_{accs}$ in expression (3.12) for calculation of the communication speedup based on the metrics $R_{app}$, $D_o$ and $D_{accs}$.

A set of analytical models of cluster node networking subsystems with CPSO capability was to be developed. The models were to be characterized as to their complexity and precision of prediction of the $D_o$ and $D_{accs}$ metrics. This work was discussed in Chapter VI.

Therefore, the dissertation thesis is validated, and the dissertation contributions are delivered in full. The research work within the scope of this dissertation is thus concluded.
CHAPTER VIII

FUTURE WORK

In this chapter, logical extensions of the research work presented in this dissertation are discussed. The refinement of the communication speedup and the related metrics are discussed. The extensions of the QN-based models and the use of GSPN-based models for performance analysis of the cluster node networking subsystems are reviewed. The effects of CPSO on system scalability and adaptability are discussed. The related publications by the author, prior, as well as planned for the future, are also discussed.

8.1 Introduction

As mentioned in Chapter VII, the dissertation research work has resulted in introduction of the communication speedup metric \( S_{comm} \) defined as a function of the system-level metrics \( D_o \) and \( D_{accs} \) in (3.8). \( D_o \) was introduced earlier in [26] and \( D_{accs} \) was introduced in this dissertation. It was also shown that \( S_{comm} \) could be extended to include \( R_{app} \) in expression (3.12) in order to account for the application-level phenomena, specifically, the application’s ability to overlap its computation and communication requests.

These metrics were shown to be instrumental for the empirical quantitative analysis of the performance effects of CPSO scenarios of interest. These metrics were also obtained in
the course of analytical modeling with the use of QN-based models of CPSO performance effects. The following sections describe the directions for future work that may be taken in order to further develop and extend the results discussed in this dissertation.

8.2 The communication speedup and the related metric

The communication speedup metrics $S_{comm}$ defined in (3.7) is a fairly high-level metric in a sense that it describes the effects of optimizations in networking subsystems on application performance in terms of the relative decrease (or increase) of the portion of the application execution time attributed to the processing of the application communication requests, including the effects of concurrent processing of communication and computation.

In this dissertation, the metric is considered a function of $D_o$ and $D_{accs}$ metrics that describe two complimentary system-level effects. The concurrency in processing of the application-specific computation and the application communication requests as offered by the underlying computer system is described by $D_o$, and the acceleration of the processing of the application communication requests is described by $D_{accs}$. While serving the purpose of quantifying the performance effects of networking subsystem optimizations in general, the metrics do not offer an insight in the system internal structure to the degree that would allow one to attribute application performance effects for a given range of parameters of communication requests to specific system-level phenomena. However, such
an insight would be instrumental in the course of optimization of the system configuration in order to achieve the desirable values of $D_o$, $D_{accs}$, and $S_{comm}$.

Theoretically, these effects could be isolated with the use of the $D_o$ and $D_{accs}$ metrics by way of incremental variation of a given system parameter and observation of the corresponding changes in the values of the metrics. However, practical applications of this approach may be complicated by lack of resources or by unavailability of system components needed in order to achieve the values of the system parameters in the desired range.

One approach to addressing this issue is to offer less general performance metrics that account for the system-level phenomena with finer granularity. For example, the metric $D_o$ could be considered a function of several low-level metrics, each describing a specific subset of system-level phenomena that contribute to a certain value of $D_o$. One could offer a set of metrics for describing system-level contention for sets of shared resources, such as communication buffer space, message and packet queues, system and application heap, etc.. When $D_o$ is represented as a function of these metrics, the effects of each of the system-level phenomena that correspond to each of the metrics can be studied analytically. This is equivalent to offering a deterministic analytical model for $D_o$ and $D_{accs}$ as functions of the low-level metrics mentioned earlier. This line of reasoning is continued in Section 8.3.

Other directions for the extension of the presented research is to study the $S_{comm}$ metric under various scenarios of scaling the application computation and communication demands. This dissertation considered an example of studying $S_{comm}$ under the constant
work scaling scenario. This aspect of the experimentation and modeling is discussed in Chapter V.

Yet another direction for the extension of the research presented in this dissertation is to offer other composite metrics for the effects of networking subsystem optimizations on the application performance. They may be functions of \( D_o, D_{accs}, R_{app} \), and other metrics of interest whose optimization would result in desirable application performance effects.

### 8.3 Extentions of the QN-based models

The process of formulating models for the high-level system metrics can be extended to include other modeling approaches discussed in Chapter II. In this dissertation, the high-level deterministic model for \( S_{comm} \) as a function of \( D_o \) and \( D_{accs} \) is complemented with lower-level QN-based analytical models for \( D_o \) and \( D_{accs} \). However, the modeling granularity of the system component representation is still quite coarse: only two components - the compute subsystem and the networking subsystem are considered.

The reasons for this simplification were discussed in Chapter VI. First, the QN-based modeling formalism made it difficult to model simultaneous resource possession. Second, obtaining the protocol stack processing analysis detail needed in order to take advantage of the finer component representation granularity required more resources than was available for the protocol stack trace analysis.

Therefore, the obvious extensions of the presented research are to address these issues. The QN-based models could be studied further in order to increase the system component
representation granularity while avoiding the need to deal with the simultaneous resource possession issues. The protocol stack processing could be understood in finer detail with the use of kernel tracing tools such as LTT [104], and the results of the study could be combined with the finer component representation in order to refine the models and to obtain more precise quantitative estimates for the system metrics of interest.

Finally, in addition to the system-level resource demands, the application-level demands could also be represented in the models. It would be interesting to observe the effects of such model adjustments on the modeling precision.

8.4 GSPN-based models

As discussed in Chapter II, certain limitations of the QN-based models may account for the quantitative inaccuracies of the models discussed in this dissertation. It was also mentioned that GSPN-based models did not have those limitations. Therefore, it would be interesting to repeat the modeling experiments presented in this work with the use of GSPN-based models and to compare the results with the results of the QN-based models.

The author has developed several Petri Nets of different complexity for modeling the protocol stack processing. The lack of a robust automated analysis tool did not allow the author to conduct the study of these models and to present the modeling results in this dissertation. However, this work will be continued as a part of other research efforts by the author and his colleagues.
8.5 Other application and system characteristics

Performance effects of the communication protocol stack offload were studied in this dissertation. However, the stack processing offload also affects other system characteristics, such as system scalability and system’s adaptability to the changing application processing demands.

A simple example of CPSO effects on the system scalability is as follows. When a large portion of the stack processing is offloaded off the node compute subsystem, the processing capacity of the networking subsystem can be scaled without significant effects on the compute subsystem. For instance, doubling the networking subsystem processing capacity while keeping the networking subsystem utilization constant would result in the commensurate increase in the service demands associated with the protocol stack processing on the node compute subsystem. But, if the compute subsystem is not heavily utilized by the protocol stack processing (which is the case if a significant portion of the stack is offloaded), this increase would not result in significant changes of the system throughput and response time. In other words, the system service parameters, as well as the compute subsystem utilization by the stack processing, would not be affected to a significant degree.

Such system characteristics are very desirable. They simplify the system’s performance analysis and service capacity planning. They also make the system more adaptable to the bursts of processing demands. Therefore, it is desirable to study these system-level effects of the communication protocol offload.
8.6 Related publications

The author of this dissertation has been involved in studying the effects of concurrency on performance of communications subsystems since his early research at the Department of Computer Science and Engineering, Mississippi State University. His Master’s thesis “Concurrency, multi-threading, and message-passing” [76], considered the benefits of concurrency in processing of the application communication requests in message passing libraries implementing Message Passing Interface (MPI) specification [33]. As an extension of the thesis research in design and implementation techniques of message passing middleware, the author has published two papers in collaboration with Dr. Skjellum.

The first paper, “Shared-memory communication approaches for an MPI message passing library” [74] studied several approaches to the design and implementation of shared-memory communication protocol modules with implications on the achievable communication performance of these approaches. The second paper, “Multi-threaded Message Passing Interface (MPI) Architecture: Performance and Program Issues” [75] was a direct extension of the earlier work by the author as a part of the research group at MSU [89], as well as his thesis mentioned earlier. The paper discussed a thread-safe multi-threaded architecture for the MPI message-passing middleware that exploited the inherent concurrency in the processing of application communication requests. The architecture also allowed applications to take advantage of the concurrency in processing the application-specific computation and communication requests, as afforded by the system.
The latter work is closely related to the topic of this dissertation. The work on the quantification of the performance effects of CPSO scenarios is a logical extension of this prior work on studying the benefits of concurrency in message passing.

The selected topics for the future publications are discussed in the earlier sections of this chapter. All of them are equally interesting. At this point, the author plans to summarize his experiences with the modeling of the CPSO performance effects in a paper with the following tentative title: “Concurrency and Computation: Practice and Experience”. The paper will be developed in collaboration with Dr. Skjellum and Dr. Dimitrov, and it will include the material discussed in personal communications with these gentlemen.

The author also plans to continue refining the QN-based analytical models of the networking subsystems and to arrive at conclusions regarding the degree to which the modeling limitations, the granularity of the representation of the communication stack processing demands, and the system component representation granularity affect the modeling precision. Additionally, the author plans to complete the development of the GSPN-based models, to analyze their modeling results, and to compare them with the results of the QN-based models. These research efforts will be discussed in a series of publications whose title and exact contents are to be determined at a later date.

8.7 Summary

This chapter discussed the directions for future work that would extend the results presented in this dissertation. The extensions of the high-level system metrics $D_o$, $D_{acce}$, and
$S_{comm}$ were discussed in Section 8.2. The refinements of the QN-based analytical models were discussed in Section 8.3, and the development and application of the GSPN-based models was discussed in Section 8.4. The effects of the CPSO phenomena on system scalability and adaptability to the changing processing demands were discussed in Section 8.5.

The prior related publications were described in Section 8.6. Several topics for the future publications that would extend and refine the research efforts discussed in this dissertation were suggested in Section 8.6.
REFERENCES


